# Contents

1 Getting Started 3
2 Using PyPy 23
3 Development 57
4 Further resources 203
5 Contact 317
6 Indices and tables 319
Bibliography 321
Index 323
Welcome to the documentation for PyPy, a fast, compliant alternative implementation of the Python language.

- If you want to find out more about what PyPy is, have a look at our *What is PyPy?* or consult the PyPy website.
- If you’re interested in trying PyPy out, check out the installation instructions.
- If you want to help develop PyPy, please have a look at contributing and get in touch (Contact)!

All of the documentation and source code is available under the MIT license, unless otherwise specified. Consult LICENSE.
1.1 What is PyPy?

Historically, PyPy has been used to mean two things. The first is the RPython translation toolchain for generating interpreters for dynamic programming languages. And the second is one particular implementation of Python produced with it. Because RPython uses the same syntax as Python, this generated version became known as Python interpreter written in Python. It is designed to be flexible and easy to experiment with.

To make it more clear, we start with source code written in RPython, apply the RPython translation toolchain, and end up with PyPy as a binary executable. This executable is the Python interpreter.

Double usage has proven to be confusing, so we’ve moved away from using the word PyPy to mean both toolchain and generated interpreter. Now we use word PyPy to refer to the Python implementation, and explicitly mention RPython translation toolchain when we mean the framework.

Some older documents, presentations, papers and videos will still have the old usage. You are hereby warned.

We target a large variety of platforms, small and large, by providing a compiler toolsuite that can produce custom Python versions. Platform, memory and threading models, as well as the JIT compiler itself, are aspects of the translation process - as opposed to encoding low level details into the language implementation itself.

For more details, have a look at our architecture overview.

1.2 Goals and Architecture Overview

## Contents

- Goals and Architecture Overview
  - Mission statement
  - High Level Goals
This document gives an overview of the goals and architecture of PyPy. If you’re interested in using PyPy or hacking on it, have a look at our getting started section.

1.2.1 Mission statement

We aim to provide a compliant, flexible and fast implementation of the Python Language which uses the RPython toolchain to enable new advanced high-level features without having to encode the low-level details. We call this PyPy.

1.2.2 High Level Goals

Our main motivation for developing the translation framework is to provide a full featured, customizable, fast and very compliant Python implementation, working on and interacting with a large variety of platforms and allowing the quick introduction of new advanced language features.

This Python implementation is written in RPython as a relatively simple interpreter, in some respects easier to understand than CPython, the C reference implementation of Python. We are using its high level and flexibility to quickly experiment with features or implementation techniques in ways that would, in a traditional approach, require pervasive changes to the source code. For example, PyPy’s Python interpreter can optionally provide lazily computed objects - a small extension that would require global changes in CPython. Another example is the garbage collection technique: changing CPython to use a garbage collector not based on reference counting would be a major undertaking, whereas in PyPy it is an issue localized in the translation framework, and fully orthogonal to the interpreter source code.

1.2.3 PyPy Python Interpreter

PyPy’s Python Interpreter is written in RPython and implements the full Python language. This interpreter very closely emulates the behavior of CPython. It contains the following key components:

- a bytecode compiler responsible for producing Python code objects from the source code of a user application;
- a bytecode evaluator responsible for interpreting Python code objects;
- a standard object space, responsible for creating and manipulating the Python objects seen by the application.

The bytecode compiler is the preprocessing phase that produces a compact bytecode format via a chain of flexible passes (tokenizer, lexer, parser, abstract syntax tree builder, bytecode generator). The bytecode evaluator interprets this bytecode. It does most of its work by delegating all actual manipulations of user objects to the object space. The latter can be thought of as the library of built-in types. It defines the implementation of the user objects, like integers and lists, as well as the operations between them, like addition or truth-value-testing.

This division between bytecode evaluator and object space gives a lot of flexibility. One can plug in different object spaces to get different or enriched behaviours of the Python objects.
1.2.4 Layers

RPython

RPython is the language in which we write interpreters. Not the entire PyPy project is written in RPython, only the parts that are compiled in the translation process. The interesting point is that RPython has no parser, it’s compiled from the live python objects, which makes it possible to do all kinds of metaprogramming during import time. In short, Python is a meta programming language for RPython.

The RPython standard library is to be found in the rlib subdirectory. Consult Getting Started with RPython for further reading.

Translation

The translation toolchain - this is the part that takes care of translating RPython to flow graphs and then to C. There is more in the architecture document written about it.

It lives in the rpython directory: flowspace, annotator and rtyper.

PyPy Interpreter

This is in the pypy directory. pypy/interpreter is a standard interpreter for Python written in RPython. The fact that it is RPython is not apparent at first. Built-in modules are written in pypy/module/* . Some modules that CPython implements in C are simply written in pure Python; they are in the top-level lib_pypy directory. The standard library of Python (with a few changes to accommodate PyPy) is in lib-python.

JIT Compiler

Just-in-Time Compiler (JIT): we have a tracing JIT that traces the interpreter written in RPython, rather than the user program that it interprets. As a result it applies to any interpreter, i.e. any language. But getting it to work correctly is not trivial: it requires a small number of precise “hints” and possibly some small refactorings of the interpreter. The JIT itself also has several almost-independent parts: the tracer itself in rpython/jit/metainterp, the optimizer in rpython/jit/metainterp/optimizer that optimizes a list of residual operations, and the backend in rpython/jit/backend/<machine-name> that turns it into machine code. Writing a new backend is a traditional way to get into the project.

Garbage Collectors

Garbage Collectors (GC): as you may notice if you are used to CPython’s C code, there are no Py_INCREF/Py_DECREF equivalents in RPython code. Garbage Collection in RPython is inserted during translation. Moreover, this is not reference counting; it is a real GC written as more RPython code. The best one we have so far is in rpython/memory/gc/incminimark.py.

1.3 Downloading and Installing PyPy

1.3.1 Using a packaged PyPy

Some Linux distributions provide a pypy package. Note that in order to install additional modules that require compilation, you may need to install additional packages such as pypy-dev. This will manifest as an error about “missing
Python.h”. Distributions do not as of yet supply many pypy-ready packages, if you require additional modules we recommend creating a virtualenv and using pip.

1.3.2 Download a pre-built PyPy

The quickest way to start using PyPy is to download a prebuilt binary for your OS and architecture. You may be able to use either use the most recent release or one of our development nightly build. These builds depend on dynamically linked libraries that may not be available on your OS. See the section about Linux binaries for more info and alternatives that may work on your system.

Please note that the nightly builds are not guaranteed to be as stable as official releases, use them at your own risk.

1.3.3 Installing PyPy

PyPy is ready to be executed as soon as you unpack the tarball or the zip file, with no need to install it in any specific location:

```
$ tar xf pypy-x.y.z.tar.bz2
$ ./pypy-x.y.z/bin/pypy
Python 2.7.x (xxxxxxxxxxxx, Date, Time)
[PyPy x.y.z with GCC x.y.z] on linux2
Type "help", "copyright", "credits" or "license" for more information.
And now for something completely different: "PyPy is an exciting technology that lets you to write fast, portable, multi-platform interpreters with less effort"
```  

If you want to make PyPy available system-wide, you can put a symlink to the pypy executable in `/usr/local/bin`. It is important to put a symlink and not move the binary there, else PyPy would not be able to find its library.

1.3.4 Installing more modules

If you want to install 3rd party libraries, the most convenient way is to install pip using `ensurepip` (unless you want to install virtualenv as explained below; then you can directly use pip inside virtualenvs):

```
$ ./pypy-xxx/bin/pypy -m ensurepip
$ ./pypy-xxx/bin/pip install -U pip wheel # to upgrade to the latest versions
$ ./pypy-xxx/bin/pip install pygments # for example
```

Third party libraries will be installed in `pypy-xxx/site-packages`, and the scripts in `pypy-xxx/bin`.

1.3.5 Installing using virtualenv

It is often convenient to run pypy inside a virtualenv. To do this you need a version of virtualenv – 1.6.1 or greater. You can then install PyPy both from a precompiled tarball or from a mercurial checkout after translation:

```
# from a tarball
$ virtualenv -p /opt/pypy-xxx/bin/pypy my-pypy-env

# from the mercurial checkout
$ virtualenv -p /path/to/pypy/translator/goal/pypy-c my-pypy-env
```
# in any case activate it
$ source my-pypy-env/bin/activate

Note that my-pypy-env/bin/python is now a symlink to my-pypy-env/bin/pypy so you should be able to run pypy simply by typing:

$ python

You should still upgrade pip and wheel to the latest versions via:

$ my-pypy-env/bin/pip install -U pip wheel

## 1.3.6 Building PyPy yourself

If you’re interested in getting more involved, or doing something different with PyPy, consult the build instructions.

### 1.4 Building PyPy from Source

For building PyPy, we recommend installing a pre-built PyPy first (see Downloading and Installing PyPy). It is possible to build PyPy with CPython, but it will take a lot longer to run – depending on your architecture, between two and three times as long.

Even when using PyPy to build PyPy, translation is time-consuming – 30 minutes on a fast machine – and RAM-hungry. You will need at least 2 GB of memory on a 32-bit machine and 4GB on a 64-bit machine.

#### 1.4.1 Before you start

Our normal development workflow avoids a full translation by using test-driven development. You can read more about how to develop PyPy here, and latest translated (hopefully functional) binary packages are available on our buildbot’s nightly builds

You will need the build dependencies below to run the tests.

#### 1.4.2 Clone the repository

If you prefer to compile your own PyPy, or if you want to modify it, you will need to obtain a copy of the sources. This can be done either by downloading them from the download page or by checking them out from the repository using mercurial. We suggest using mercurial if you want to access the current development.

You must issue the following command on your command line, DOS box, or terminal:

```
hg clone http://bitbucket.org/pypy/pypy pypy
```

This will clone the repository and place it into a directory named pypy, and will get you the PyPy source in pypy/pypy and documentation files in pypy/pypy/doc. We try to ensure that the tip is always stable, but it might occasionally be broken. You may want to check out our nightly tests: find a revision (12-chars alphanumeric string, e.g. “963e808156b3”) that passed at least the (linux32) tests (corresponding to a + sign on the line success) and then, in your cloned repository, switch to this revision using:

```
hg up -r XXXXX
```
where XXXXX is the revision id.

### 1.4.3 Install build-time dependencies

(Note: for some hints on how to translate the Python interpreter under Windows, see the windows document. For hints on how to cross-compile in a chroot using scratchbox2, see the arm document in the RPython documentation)

The host Python needs to have CFFI installed. If translating on PyPy, CFFI is already installed. If translating on CPython, you need to install it, e.g. using pip install cffi.

To build PyPy on Unix using the C translation backend, you need at least a C compiler and make installed. Further, some optional modules have additional dependencies:

- **cffi**, **ctypes** libffi, pkg-config
- **zlib** libz
- **bz2** libbz2
- **pyexpat** libexpat1
- **_ssl** libssl
- **_vmprof** libunwind (optional, loaded dynamically at runtime)

Make sure to have these libraries (with development headers) installed before building PyPy, otherwise the resulting binary will not contain these modules. Furthermore, the following libraries should be present after building PyPy, otherwise the corresponding CFFI modules are not built (you can run or re-run pypy/tool/release/package.py to retry to build them; you don’t need to re-translate the whole PyPy):

- **sqlite3** libsqlite3
- **curses** libncurses-dev (for PyPy2) libncursesw-dev (for PyPy3)
- **gdbm** libgdbm-dev
- **tk** tk-dev
- **lzma (PyPy3 only)** liblzma

To run untranslated tests, you need the Boehm garbage collector libgc.

On recent Debian and Ubuntu (16.04 onwards), this is the command to install all build-time dependencies:

```
apt-get install gcc make libffi-dev pkg-config zlib1g-dev libbz2-dev \
libsqlite3-dev libncurses5-dev libexpat1-dev libssl1-dev libgdbm-dev \
tk-dev libgc-dev python-cffi \nliblzma-dev libncursesw5-dev # these two only needed on PyPy3
```

On older Debian and Ubuntu (12.04-14.04):

```
apt-get install gcc make libffi-dev pkg-config libz-dev libbz2-dev \
libsqlite3-dev libncurses-dev libexpat1-dev libssl-dev libgdbm-dev \
tk-dev libgc-dev python-cffi \nliblzma-dev libncursesw-dev # these two only needed on PyPy3
```

On Fedora:

```
dnf install gcc make libffi-devel pkgconfig zlib-devel bzip2-devel \
sqlite-devel ncurses-devel expat-devel openssl-devel tk-devel \
gdbm-devel python-cffi \
xz-devel # For lzma on PyPy3.
```
On SLES11:

```bash
zypper install gcc make python-devel pkg-config \  
zlib-devel libopenssl-devel libbz2-devel sqlite3-devel \  
libexpat-devel libffi-devel python-curses python-cffi \  
xz-devel # For Izma on PyPy3.
(XXX plus the SLES11 version of libgdbm-dev and tk-dev)
```

On Mac OS X:

Most of these build-time dependencies are installed alongside the Developer Tools. However, note that in order for the installation to find them you may need to run:

```bash
xcode-select --install
```

An exception is OpenSSL, which is no longer provided with the operating system. It can be obtained via Homebrew (with `$ brew install openssl`), but it will not be available on the system path by default. The easiest way to enable it for building pypy is to set an environment variable:

```bash
export PKG_CONFIG_PATH=$(brew --prefix)/opt/openssl/lib/pkgconfig
```

After setting this, translation (described next) will find the OpenSSL libs as expected.

## 1.4.4 Run the translation

We usually translate in the `pypy/goal` directory, so all the following commands assume your `$pwd` is there.

Translate with JIT:

```bash
pypy ../../rpython/bin/rpython --opt=jit
```

Translate without JIT:

```bash
pypy ../../rpython/bin/rpython --opt=2
```

Note this translates pypy via the `targetpypystandalone.py` file, so these are shorthand for:

```bash
pypy ../../rpython/bin/rpython <rpython options> targetpypystandalone.py <pypy options>
```

More help is available via `--help` at either option position, and more info can be found in the [Configuration Options for PyPy](#) section.

(You can use `python` instead of `pypy` here, which will take longer but works too.)

If everything works correctly this will:

1. Run the rpython translation chain, producing a database of the entire pypy interpreter. This step is currently single threaded, and RAM hungry. As part of this step, the chain creates a large number of C code files and a Makefile to compile them in a directory controlled by the `PYPY_USESSION_DIR` environment variable.
2. Create an executable `pypy-c` by running the Makefile. This step can utilize all possible cores on the machine.
3. Copy the needed binaries to the current directory.

The resulting executable behaves mostly like a normal Python interpreter (see [Differences between PyPy and CPython](#)), and is ready for testing, for use as a base interpreter for a new virtualenv, or for packaging into a binary suitable for installation on another machine running the same OS as the build machine.

### 1.4. Building PyPy from Source
Note that step 4 is merely done as a convenience, any of the steps may be rerun without rerunning the previous steps.

### 1.4.5 Making a debug build of PyPy

Rerun the `Makefile` with the `make lldebug` or `make lldebug0` target, which will build in a way that running under a debugger makes sense. Appropriate compilation flags are added to add debug info, and for `lldebug0` compiler optimizations are fully disabled. If you stop in a debugger, you will see the very wordy machine-generated C code from the rpython translation step, which takes a little bit of reading to relate back to the rpython code.

### 1.4.6 Build cffi import libraries for the stdlib

Various stdlib modules require a separate build step to create the cffi import libraries in the out-of-line API mode. This is done by the following command:

```
cd pypy/goal
PYTHONPATH=../.. ./pypy-c ../tool/build_cffi_imports.py
```

### 1.4.7 Packaging (preparing for installation)

Packaging is required if you want to install PyPy system-wide, even to install on the same machine. The reason is that doing so prepares a number of extra features that cannot be done lazily on a root-installed PyPy, because the normal users don’t have write access. This concerns mostly libraries that would normally be compiled if and when they are imported the first time.

```
cd pypy/tool/release
./package.py --archive-name=pypy-VER-PLATFORM
```

This creates a clean and prepared hierarchy, as well as a `.tar.bz2` with the same content; both are found by default in `/tmp/usession-YOURNAME/build/`. You can then either move the file hierarchy or unpack the `.tar.bz2` at the correct place.

It is recommended to use `package.py` because custom scripts will invariably become out-of-date. If you want to write custom scripts anyway, note an easy-to-miss point: some modules are written with CFFI, and require some compilation. If you install PyPy as root without pre-compiling them, normal users will get errors:

- **PyPy 2.5.1 or earlier:** normal users would see permission errors. Installers need to run `pypy -c "import gdbm"` and other similar commands at install time; the exact list is in `pypy/tool/release/package.py`. Users seeing a broken installation of PyPy can fix it after-the-fact if they have sudo rights, by running once e.g. `sudo pypy -c "import gdbm"`.

- **PyPy 2.6 and later:** anyone would get `ImportError: no module named _gdbm_cffi`. Installers need to run `pypy --_gdbm_build.py` in the `lib_pypy` directory during the installation process (plus others; see the exact list in `pypy/tool/release/package.py`). Users seeing a broken installation of PyPy can fix it after-the-fact, by running `pypy /path/to/lib_pypy/_gdbm_build.py`. This command produces a file called `_gdbm_cffi.pypy-41.so` locally, which is a C extension module for PyPy. You can move it at any place where modules are normally found: e.g. in your project’s main directory, or in a directory that you add to the env var PYTHONPATH.

### 1.4.8 Installation

PyPy dynamically finds the location of its libraries depending on the location of the executable. The directory hierarchy of a typical PyPy installation looks like this:
The hierarchy shown above is relative to a PREFIX directory. PREFIX is computed by starting from the directory
where the executable resides, and “walking up” the filesystem until we find a directory containing lib_pypy and
lib-python/2.7.

To install PyPy system wide on unix-like systems, it is recommended to put the whole hierarchy alone (e.g. in /opt/
pypy) and put a symlink to the pypy executable into /usr/bin or /usr/local/bin.

If the executable fails to find suitable libraries, it will report debug: WARNING: library path not
found, using compiled-in sys.path and then attempt to continue normally. If the default path is us-
able, most code will be fine. However, the sys.prefix will be unset and some existing libraries assume that this is
never the case.

1.5 Translating on Windows

RPython is supported on Windows platforms, starting with Windows 2000. The following text gives some hints about
how to translate a interpreter written in RPython, using PyPy as an example.

PyPy supports only being translated as a 32bit program, even on 64bit Windows. See at the end of this page for what
is missing for a full 64bit translation.

To build pypy-c you need a working python environment, and a C compiler. It is possible to translate with a CPython
2.6 or later, but this is not the preferred way, because it will take a lot longer to run – depending on your architecture,
between two and three times as long. So head to our downloads and get the latest stable version.

Microsoft Visual Studio is preferred as a compiler, but there are reports of success with the mingw32 port of gcc.

1.5.1 Installing Visual Compiler v9 (for Python 2.7)

This compiler, while the standard one for Python 2.7, is deprecated. Microsoft has made it available as the Micro-
soft Visual C++ Compiler for Python 2.7 (the link was checked in May 2018). Note that the compiler suite may
be installed in C:\Users\<user name>\AppData\Local\Programs\Common\Microsoft\Visual
C++ for Python or in C:\Program Files (x86)\Common Files\Microsoft\Visual C++ for
Python. A current version of setuptools will be able to find it there. Also, you must download and install the
.Net Framework 3.5, otherwise mt.exe will silently fail. Installation will begin automatically by running the
mt.exe command by hand from a DOS window (that is how the author discovered the problem).

1.5.2 Installing “Build Tools for Visual Studio 2015” (for Python 3)

As documented in the CPython Wiki, CPython recommends Visual C++ version 14.0 for python version 3.5. A
compact version of the compiler suite can be obtained from Microsoft downloads, search the page for “Microsoft
Build Tools 2015”.

You will need to reboot the computer for the installation to successfully install and run the mt.exe mainfest compiler.
The installation will set the VSI40COMNTOOLS environment variable, this is key to distutils/setuptools finding the compiler
1.5.3 Translating PyPy with Visual Studio

We routinely test translation of PyPy 2.7 using v9 and PyPy 3 with vc14. Other configurations may work as well.

The translation scripts will set up the appropriate environment variables for the compiler, so you do not need to run vcvars before translation. They will attempt to locate the same compiler version that was used to build the Python interpreter doing the translation. Failing that, they will pick the most recent Visual Studio compiler they can find. In addition, the target architecture (32 bits, 64 bits) is automatically selected. A 32 bit build can only be built using a 32 bit Python and vice versa. By default the interpreter is built using the Multi-threaded DLL (/MD) runtime environment.

If you wish to override this detection method to use a different compiler (mingw or a different version of MSVC):

- set up the PATH and other environment variables as needed
- set the CC environment variable to compiler exe to be used, for a different version of MSVC \texttt{SET CC=cl.exe}.

\textbf{Note:} The RPython translator does currently not support 64 bit Python, and translation will fail in this case.

Python and a C compiler are all you need to build pypy, but it will miss some modules that relies on third-party libraries. See below how to get and build them.

Please see the \textit{non-windows instructions} for more information, especially note that translation is RAM-hungry. A standard translation requires around 4GB, so special preparations are necessary, or you may want to use the method in the notes of the build instructions to reduce memory usage at the price of a slower translation:

\begin{verbatim}
set PYPY_GC_MAX_DELTA=200MB
pypy --jit loop_longevity=300 ../../../rpython/bin/rpython -Ojit targetpypystandalone
set PYPY_GC_MAX_DELTA=
PYTHONPATH=../.. ./pypy-c ../../../tool/build_cffi_imports.py
\end{verbatim}

1.5.4 Setting Up Visual Studio 9.0 for building SSL in Python3

\textbf{Note:} this is old information, left for historical reference. We recommend using Visual Studio 2015, which now seems to properly set this all up.

On Python3, the \texttt{ssl} module is based on \texttt{cffi}, and requires a build step after translation. However \texttt{distutils} does not support the Micorosft-provided Visual C compiler, and \texttt{cffi} depends on \texttt{distutils} to find the compiler. The traditional solution to this problem is to install the \texttt{setuptools} module via running \texttt{-m ensurepip} which installs \texttt{pip} and \texttt{setuptools}. However \texttt{pip} requires \texttt{ssl}. So we have a chicken-and-egg problem: \texttt{ssl} depends on \texttt{cffi} which depends on \texttt{setuptools}, which depends on \texttt{ensurepip}, which depends on \texttt{ssl}.

In order to solve this, the buildbot sets an environment variable that helps \texttt{distutils} find the compiler without \texttt{setuptools}:

\begin{verbatim}
set VS90COMNTOOLS=C:\Program Files (x86)\Common Files\Microsoft\Visual C++ 9.0\VC\bin
\end{verbatim}

or whatever is appropriate for your machine. Note that this is not enough, you must also copy the \texttt{vcvarsall.bat} file from the \texttt{\ldots\9.0} directory to the \texttt{\ldots\9.0\VC} directory, and edit it, changing the lines that set \texttt{VCINSTALLDIR} and \texttt{WindowsSdkDir}:

\begin{verbatim}
set VCINSTALLDIR=%~dp0\WindowsSdk\8.1
def VCVARSALL=%~dp0\VC\bin\vcvarsall.bat
\end{verbatim}
1.5.5 Preparing Windows for the large build

Normally 32bit programs are limited to 2GB of memory on Windows. It is possible to raise this limit, to 3GB on Windows 32bit, and almost 4GB on Windows 64bit.

On Windows 32bit, it is necessary to modify the system: follow http://usa.autodesk.com/adsk/servlet.ps/dl/item?siteID=123112&id=9583842&linkID=9240617 to enable the “3GB” feature, and reboot. This step is not necessary on Windows 64bit.

Then you need to execute:

```
<path-to-visual>\vc\vcvars.bat
editbin /largeaddressaware translator.exe
```

where `translator.exe` is the pypy.exe or cpython.exe you will use to translate with.

1.5.6 Installing external packages

We use a subrepository inside pypy to hold binary compiled versions of the build dependencies for windows. As part of the `rpython` setup stage, environment variables will be set to use these dependencies. The repository has a README file on how to replicate, and a branch for each supported platform. You may run the `get_externals.py` utility to checkout the proper branch for your platform and PyPy version.

1.5.7 Using the mingw compiler

You can compile an RPython program with the mingw compiler, using the `-cc=mingw32` option; gcc.exe must be on the PATH. If the -cc flag does not begin with “ming”, it should be the name of a valid gcc-derivative compiler, i.e. x86_64-w64-mingw32-gcc for the 64 bit compiler creating a 64 bit target.

You probably want to set the CPATH, LIBRARY_PATH, and PATH environment variables to the header files, lib or dlls, and dlls respectively of the locally installed packages if they are not in the mingw directory heirarchy.

**libffi for the mingw compiler**

To enable the `_rawffi` (and ctypes) module, you need to compile a mingw version of libffi. Here is one way to do this, which should allow you to try to build for win64 or win32:

1. Download and unzip a mingw32 build or mingw64 build, say into c:\mingw
2. If you do not use cygwin, you will need msys to provide make, autoconf tools and other goodies.
   1. Download and unzip a `msys for mingw`, say into c:\msys
   2. Edit the c:\msys\etc\fstab file to mount c:\mingw
3. Download and unzip the `libffi source files`, and extract them in the base directory.
4. Run c:\msys\msys\bat or a cygwin shell which should make you feel better since it is a shell prompt with shell tools.
5. From inside the shell, cd to the libffi directory and do:

   ```
   sh ./configure
   make
   cp .libs/libffi-5.dll <somewhere on the PATH>
   ```
If you can’t find the dll, and the libtool issued a warning about “undefined symbols not allowed”, you will need to edit the libffi Makefile in the toplevel directory. Add the flag -no-undefined to the definition of libffi_la_LDFLAGS

If you wish to experiment with win64, you must run configure with flags:

```
sh ./configure --build=x86_64-w64-mingw32 --host=x86_64-w64-mingw32
```

or such, depending on your mingw64 download.

**hacking on PyPy with the mingw compiler**

Since hacking on PyPy means running tests, you will need a way to specify the mingw compiler when hacking (as opposed to translating). As of March 2012, –cc is not a valid option for pytest.py. However if you set an environment variable CC to the compiler exe, testing will use it.

### 1.5.8 What is missing for a full 64-bit translation

The main blocker is that we assume that the integer type of RPython is large enough to (occasionally) contain a pointer value cast to an integer. The simplest fix is to make sure that it is so, but it will give the following incompatibility between CPython and PyPy on Win64:

**CPython:** sys.maxint == 2**31-1, sys.maxsize == 2**63-1

**PyPy:** sys.maxint == sys.maxsize == 2**63-1

... and, correspondingly, PyPy supports ints up to the larger value of sys.maxint before they are converted to long. The first decision that someone needs to make is if this incompatibility is reasonable.

Assuming that it is, the first thing to do is probably to hack CPython until it fits this model: replace the field in PyIntObject with a long long field, and change the value of sys.maxint. This might just work, even if half-brokenly: I’m sure you can crash it because of the precision loss that undoubtedly occurs everywhere, but try not to. :-)  

Such a hacked CPython is what you’ll use in the next steps. We’ll call it CPython64/64.

It is probably not too much work if the goal is only to get a translated PyPy executable, and to run all tests before translation. But you need to start somewhere, and you should start with some tests in rpython/translator/c/test/, like test_standalone.py and test_newgc.py: try to have them pass on top of CPython64/64.

Keep in mind that this runs small translations, and some details may go wrong. The most obvious one is to check that it produces C files that use the integer type Signed — but what is Signed defined to? It should be equal to long on every other platform, but on Win64 it should be something like long long.

What is more generally needed is to review all the C files in rpython/translator/c/src for the word long, because this means a 32-bit integer even on Win64. Replace it with Signed most of the times. You can replace one with the other without breaking anything on any other platform, so feel free to.

Then, these two C types have corresponding RPython types: rffi.LONG and lltype.Signed respectively. The first should really correspond to the C long. Add tests that check that integers cast to one type or the other really have 32 and 64 bits respectively, on Win64.

Once these basic tests work, you need to review rpython/rlib/ for uses of rffi.LONG versus lltype.Signed. The goal would be to fix some more long-versus-Signed issues, by fixing the tests — as always run on top of CPython64/64. Note that there was some early work done in rpython/rlib/rarithmetic with the goal of running all the tests on Win64 on the regular CPython, but I think by now that it’s a bad idea. Look only at CPython64/64.

The major intermediate goal is to get a translation of PyPy with --O2 with a minimal set of modules, starting with --no-allworkingmodules: you need to use CPython64/64 to run this translation too. Check carefully the
warnings of the C compiler at the end. By default, MSVC reports a lot of mismatches of integer sizes as warnings instead of errors.

Then you need to review `pypy/module/*` for LONG-versus-Signed issues. At some time during this review, we get a working translated PyPy on Windows 64 that includes all `--translationmodules`, i.e. everything needed to run translations. Once we have that, the hacked CPython64/64 becomes much less important, because we can run future translations on top of this translated PyPy. As soon as we get there, please distribute the translated PyPy. It’s an essential component for anyone else that wants to work on Win64! We end up with a strange kind of dependency — we need a translated PyPy in order to translate a PyPy —, but I believe it’s ok here, as Windows executables are supposed to never be broken by newer versions of Windows.

Happy hacking :-)
1.6.1 What is PyPy?

PyPy is a reimplementation of Python in Python, using the RPython translation toolchain.

PyPy tries to find new answers about ease of creation, flexibility, maintainability and speed trade-offs for language implementations. For further details see our goal and architecture document.

1.6.2 Is PyPy a drop in replacement for CPython?

Almost!

The most likely stumbling block for any given project is support for extension modules. PyPy supports a continually growing number of extension modules, but so far mostly only those found in the standard library.

The language features (including builtin types and functions) are very refined and well tested, so if your project doesn’t use many extension modules there is a good chance that it will work with PyPy.

We list the known differences in cpython differences.

1.6.3 Module xyz does not work with PyPy: ImportError

A module installed for CPython is not automatically available for PyPy — just like a module installed for CPython 2.6 is not automatically available for CPython 2.7 if you installed both. In other words, you need to install the module xyz specifically for PyPy.

On Linux, this means that you cannot use apt-get or some similar package manager: these tools are only meant for the version of CPython provided by the same package manager. So forget about them for now and read on.

It is quite common nowadays that xyz is available on PyPI and installable with pip install xyz. The simplest solution is to use virtualenv (as documented here). Then enter (activate) the virtualenv and type: pip install xyz.

If you don’t know or don’t want virtualenv, you can also install pip globally by saying pypy -m ensurepip.

If you get errors from the C compiler, the module is a CPython C Extension module using unsupported features. See below.

Alternatively, if either the module xyz is not available on PyPI or you don’t want to use virtualenv, then download the source code of xyz, decompress the zip/tarball, and run the standard command: pypy setup.py install. (Note: pypy here instead of python.) As usual you may need to run the command with sudo for a global installation. The other commands of setup.py are available too, like build.

1.6.4 Module xyz does not work in the sandboxed PyPy?

You cannot import any extension module in a sandboxed PyPy, sorry. Even the built-in modules available are very limited. Sandboxing in PyPy is a good proof of concept, and is without a doubt safe IMHO, however it is only a proof of concept. It currently requires some work from a motivated developer. However, until then it can only be used for “pure Python” example: programs that import mostly nothing (or only pure Python modules, recursively).
1.6.5 Do CPython Extension modules work with PyPy?

First note that some Linux distributions (e.g. Ubuntu, Debian) split PyPy into several packages. If you installed a package called “pypy”, then you may also need to install “pippy-dev” for the following to work.

We have experimental support for CPython extension modules, so they run with minor changes. This has been a part of PyPy since the 1.4 release, but support is still in beta phase. CPython extension modules in PyPy are often much slower than in CPython due to the need to emulate refcounting. It is often faster to take out your CPython extension and replace it with a pure python version that the JIT can see. If trying to install module xyz, and the module has both a C and a Python version of the same code, try first to disable the C version; this is usually easily done by changing some line in setup.py.

We fully support ctypes-based extensions. But for best performance, we recommend that you use the cffi module to interface with C code.

For information on which third party extensions work (or do not work) with PyPy see the compatibility wiki.

For more information about how we manage refcounting semamtics see rawrefcount

1.6.6 On which platforms does PyPy run?

PyPy currently supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD),
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

PyPy is regularly and extensively tested on Linux machines. It works on Mac and Windows: it is tested there, but most of us are running Linux so fixes may depend on 3rd-party contributions.

To bootstrap from sources, PyPy can use either CPython 2.7 or another (e.g. older) PyPy. Cross-translation is not really supported: e.g. to build a 32-bit PyPy, you need to have a 32-bit environment. Cross-translation is only explicitly supported between a 32-bit Intel Linux and ARM Linux (see here).

1.6.7 Which Python version (2.x?) does PyPy implement?

PyPy currently aims to be fully compatible with Python 2.7. That means that it contains the standard library of Python 2.7 and that it supports 2.7 features (such as set comprehensions).

1.6.8 Does PyPy have a GIL? Why?

Yes, PyPy has a GIL. Removing the GIL is very hard. On top of CPython, you have two problems: (1) GC, in this case reference counting; (2) the whole Python language.

For PyPy, the hard issue is (2): by that I mean issues like what occurs if a mutable object is changed from one thread and read from another concurrently. This is a problem for any mutable type: it needs careful review and fixes (fine-grained locks, mostly) through the whole Python interpreter. It is a major effort, although not completely impossible, as Jython/IronPython showed. This includes subtle decisions about whether some effects are ok or not for the user (i.e. the Python programmer).
CPython has additionally the problem (1) of reference counting. With PyPy, this sub-problem is simpler: we need to make our GC multithread-aware. This is easier to do efficiently in PyPy than in CPython. It doesn’t solve the issue (2), though.

Note that since 2012 there is work going on on a still very experimental Software Transactional Memory (STM) version of PyPy. This should give an alternative PyPy which works without a GIL, while at the same time continuing to give the Python programmer the complete illusion of having one. This work is currently a bit stalled because of its own technical difficulties.

1.6.9 What about numpy, numpypy, micronumpy?

Way back in 2011, the PyPy team started to reimplement numpy in PyPy. It has two pieces:

- the builtin module pypy/module/micronumpy: this is written in RPython and roughly covers the content of the numpy.core.multiarray module. Confusingly enough, this is available in PyPy under the name _numpypy. It is included by default in all the official releases of PyPy (but it might be dropped in the future).
- a fork of the official numpy repository maintained by us and informally called numpypy: even more confusing, the name of the repo on bitbucket is numpy. The main difference with the upstream numpy, is that it is based on the micronumpy module written in RPython, instead of of numpy.core.multiarray which is written in C.

Moreover, it is also possible to install the upstream version of numpy: its core is written in C and it runs on PyPy under the cpyext compatibility layer. This is what you get if you do `pypy -m pip install numpy`.

1.6.10 Should I install numpy or numpypy?

TL;DR version: you should use numpy. You can install it by doing `pypy -m pip install numpy`. You might also be interested in using the experimental PyPy binary wheels to save compilation time.

The upstream numpy is written in C, and runs under the cpyext compatibility layer. Nowadays, cpyext is mature enough that you can simply use the upstream numpy, since it passes 99.9% of the test suite. At the moment of writing (October 2017) the main drawback of numpy is that cpyext is infamously slow, and thus it has worse performance compared to numpypy. However, we are actively working on improving it, as we expect to reach the same speed, eventually.

On the other hand, numpypy is more JIT-friendly and very fast to call, since it is written in RPython: but it is a reimplemention, and it’s hard to be completely compatible: over the years the project slowly matured and eventually it was able to call out to the LAPACK and BLAS libraries to speed matrix calculations, and reached around an 80% parity with the upstream numpy. However, 80% is far from 100%. Since cpyext/numpy compatibility is progressing fast, we have discontinued support for numpypy.

1.6.11 Is PyPy more clever than CPython about Tail Calls?

No. PyPy follows the Python language design, including the built-in debugger features. This prevents tail calls, as summarized by Guido van Rossum in two blog posts. Moreover, neither the JIT nor Stackless change anything to that.

1.6.12 How do I write extension modules for PyPy?

See Writing extension modules for pypy.
1.6.13 How fast is PyPy?

This really depends on your code. For pure Python algorithmic code, it is very fast. For more typical Python programs we generally are 3 times the speed of CPython 2.7. You might be interested in our benchmarking site and our jit documentation.

Your tests are not a benchmark: tests tend to be slow under PyPy because they run exactly once; if they are good tests, they exercise various corner cases in your code. This is a bad case for JIT compilers. Note also that our JIT has a very high warm-up cost, meaning that any program is slow at the beginning. If you want to compare the timings with CPython, even relatively simple programs need to run at least one second, preferably at least a few seconds. Large, complicated programs need even more time to warm-up the JIT.

1.6.14 Couldn’t the JIT dump and reload already-compiled machine code?

No, we found no way of doing that. The JIT generates machine code containing a large number of constant addresses — constant at the time the machine code is generated. The vast majority is probably not at all constants that you find in the executable, with a nice link name. E.g. the addresses of Python classes are used all the time, but Python classes don’t come statically from the executable; they are created anew every time you restart your program. This makes saving and reloading machine code completely impossible without some very advanced way of mapping addresses in the old (now-dead) process to addresses in the new process, including checking that all the previous assumptions about the (now-dead) object are still true about the new object.

1.6.15 Would type annotations help PyPy’s performance?

Two examples of type annotations that are being proposed for improved performance are Cython types and PEP 484 - Type Hints.

Cython types are, by construction, similar to C declarations. For example, a local variable or an instance attribute can be declared "cdef int" to force a machine word to be used. This changes the usual Python semantics (e.g. no overflow checks, and errors when trying to write other types of objects there). It gives some extra performance, but the exact benefits are unclear: right now (January 2015) for example we are investigating a technique that would store machine-word integers directly on instances, giving part of the benefits without the user-supplied "cdef int".

PEP 484 - Type Hints, on the other hand, is almost entirely useless if you’re looking at performance. First, as the name implies, they are hints: they must still be checked at runtime, like PEP 484 says. Or maybe you’re fine with a mode in which you get very obscure crashes when the type annotations are wrong; but even in that case the speed benefits would be extremely minor.

There are several reasons for why. One of them is that annotations are at the wrong level (e.g. a PEP 484 “int” corresponds to Python 3’s int type, which does not necessarily fits inside one machine word; even worse, an “int” annotation allows arbitrary int subclasses). Another is that a lot more information is needed to produce good code (e.g. “this f() called here really means this function there, and will never be monkey-patched” – same with len() or list(), btw). The third reason is that some “guards” in PyPy’s JIT traces don’t really have an obvious corresponding type (e.g. “this dict is so far using keys which don’t override __hash__ so a more efficient implementation was used”). Many guards don’t even have any correspondence with types at all (“this class attribute was not modified”; “the loop counter did not reach zero so we don’t need to release the GIL”; and so on).

As PyPy works right now, it is able to derive far more useful information than can ever be given by PEP 484, and it works automatically. As far as we know, this is true even if we would add other techniques to PyPy, like a fast first-pass JIT.
1.6.16 Can I use PyPy's translation toolchain for other languages besides Python?

Yes. The toolsuite that translates the PyPy interpreter is quite general and can be used to create optimized versions of interpreters for any language, not just Python. Of course, these interpreters can make use of the same features that PyPy brings to Python: translation to various languages, stackless features, garbage collection, implementation of various things like arbitrarily long integers, etc.

Currently, we have Topaz, a Ruby interpreter; Hippy, a PHP interpreter; preliminary versions of a JavaScript interpreter (Leonardo Santagada as his Summer of PyPy project); a Prolog interpreter (Carl Friedrich Bolz as his Bachelor thesis); and a SmallTalk interpreter (produced during a sprint). On the PyPy bitbucket page there is also a Scheme and an Io implementation; both of these are unfinished at the moment.

1.6.17 How do I get into PyPy development? Can I come to sprints?

Certainly you can come to sprints! We always welcome newcomers and try to help them as much as possible to get started with the project. We provide tutorials and pair them with experienced PyPy developers. Newcomers should have some Python experience and read some of the PyPy documentation before coming to a sprint.

Coming to a sprint is usually the best way to get into PyPy development. If you get stuck or need advice, contact us. IRC is the most immediate way to get feedback (at least during some parts of the day; most PyPy developers are in Europe) and the mailing list is better for long discussions.

1.6.18 OSError: . . . cannot restore segment prot after reloc… Help?

On Linux, if SELinux is enabled, you may get errors along the lines of “OSError: externmod.so: cannot restore segment prot after reloc: Permission denied.” This is caused by a slight abuse of the C compiler during configuration, and can be disabled by running the following command with root privileges:

```
# setenforce 0
```

This will disable SELinux’s protection and allow PyPy to configure correctly. Be sure to enable it again if you need it!

1.6.19 How should I report a bug?

Our bug tracker is here: https://bitbucket.org/pypy/pypy/issues/

Missing features or incompatibilities with CPython are considered bugs, and they are welcome. (See also our list of known incompatibilities.)

For bugs of the kind “I’m getting a PyPy crash or a strange exception”, please note that: We can't do anything without reproducing the bug ourselves. We cannot do anything with traces from gdb, or core dumps. This is not only because the standard PyPy is compiled without debug symbols. The real reason is that a C-level traceback is usually of no help at all in PyPy. Debugging PyPy can be annoying.

This is a clear and useful bug report. (Admittedly, sometimes the problem is really hard to reproduce, but please try to.)

In more details:

- First, please give the exact PyPy version, and the OS.
- It might help focus our search if we know if the bug can be reproduced on a “pypy --jit off” or not. If “pypy --jit off” always works, then the problem might be in the JIT. Otherwise, we know we can ignore that part.
• If you got the bug using only Open Source components, please give a step-by-step guide that we can follow to reproduce the problem ourselves. Don’t assume we know anything about any program other than PyPy. We would like a guide that we can follow point by point (without guessing or having to figure things out) on a machine similar to yours, starting from a bare PyPy, until we see the same problem. (If you can, you can try to reduce the number of steps and the time it needs to run, but that is not mandatory.)

• If the bug involves Closed Source components, or just too many Open Source components to install them all ourselves, then maybe you can give us some temporary ssh access to a machine where the bug can be reproduced. Or, maybe we can download a VirtualBox or VMWare virtual machine where the problem occurs.

• If giving us access would require us to use tools other than ssh, make appointments, or sign a NDA, then we can consider a commerical support contract for a small sum of money.

• If even that is not possible for you, then sorry, we can’t help.

Of course, you can try to debug the problem yourself, and we can help you get started if you ask on the #pypy IRC channel, but be prepared: debugging an annoying PyPy problem usually involves quite a lot of gdb in auto-generated C code, and at least some knowledge about the various components involved, from PyPy’s own RPython source code to the GC and possibly the JIT.

1.6.20 Why doesn’t PyPy move to GitHub, Gitlab, …?

We’ve been quite happy with bitbucket.org. Moving version control systems and hosting is a lot of hard work: On the one hand, PyPy’s mercurial history is long and gnarly. On the other hand, all our infrastructure (buildbots, benchmarking, etc) would have to be adapted. So unless somebody steps up and volunteers to do all that work, it will likely not happen.

1.6.21 What is needed for Windows 64 support of PyPy?

First, please note that the Windows 32 PyPy binary works just fine on Windows 64. The only problem is that it only supports up to 4GB of heap per process.

As to real Windows 64 support: Currently we don’t have an active PyPy developer whose main development platform is Windows. So if you are interested in getting Windows 64 support, we encourage you to volunteer to make it happen! Another option would be to pay some PyPy developers to implement Windows 64 support, but so far there doesn’t seem to be an overwhelming commercial interest in it.

1.6.22 How long will PyPy support Python2?

Since RPython is built on top of Python2 and that is extremely unlikely to change, the Python2 version of PyPy will be around “forever”, i.e. as long as PyPy itself is around.
CHAPTER 2

Using PyPy

2.1 Differences between PyPy and CPython

This page documents the few differences and incompatibilities between the PyPy Python interpreter and CPython. Some of these differences are “by design”, since we think that there are cases in which the behaviour of CPython is buggy, and we do not want to copy bugs.

Differences that are not listed here should be considered bugs of PyPy.

2.1.1 Differences related to garbage collection strategies

The garbage collectors used or implemented by PyPy are not based on reference counting, so the objects are not freed instantly when they are no longer reachable. The most obvious effect of this is that files (and sockets, etc) are not promptly closed when they go out of scope. For files that are opened for writing, data can be left sitting in their output buffers for a while, making the on-disk file appear empty or truncated. Moreover, you might reach your OS’s limit on the number of concurrently opened files.

If you are debugging a case where a file in your program is not closed properly, you can use the -X track-resources command line option. If it is given, a ResourceWarning is produced for every file and socket that the garbage collector closes. The warning will contain the stack trace of the position where the file or socket was created, to make it easier to see which parts of the program don’t close files explicitly.

Fixing this difference to CPython is essentially impossible without forcing a reference-counting approach to garbage collection. The effect that you get in CPython has clearly been described as a side-effect of the implementation and not a language design decision: programs relying on this are basically bogus. It would be a too strong restriction to try to enforce CPython’s behavior in a language spec, given that it has no chance to be adopted by Jython or IronPython (or any other port of Python to Java or .NET).

Even the naive idea of forcing a full GC when we’re getting dangerously close to the OS’s limit can be very bad in some cases. If your program leaks open files heavily, then it would work, but force a complete GC cycle every n’th leaked file. The value of n is a constant, but the program can take an arbitrary amount of memory, which makes a complete GC cycle arbitrarily long. The end result is that PyPy would spend an arbitrarily large fraction of its run time in the GC — slowing down the actual execution, not by 10% nor 100% nor 1000% but by essentially any factor.
To the best of our knowledge this problem has no better solution than fixing the programs. If it occurs in 3rd-party code, this means going to the authors and explaining the problem to them: they need to close their open files in order to run on any non-CPython-based implementation of Python.

Here are some more technical details. This issue affects the precise time at which \_\_del\_\_ methods are called, which is not reliable or timely in PyPy (nor Jython nor IronPython). It also means that weak references may stay alive for a bit longer than expected. This makes “weak proxies” (as returned by weakref.proxy()) somewhat less useful: they will appear to stay alive for a bit longer in PyPy, and suddenly they will really be dead, raising a ReferenceError on the next access. Any code that uses weak proxies must carefully catch such ReferenceError at any place that uses them. (Or, better yet, don’t use weakref.proxy() at all; use weakref.ref().)

Note a detail in the documentation for weakref callbacks:

If callback is provided and not None, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized.

There are cases where, due to CPython’s refcount semantics, a weakref dies immediately before or after the objects it points to (typically with some circular reference). If it happens to die just after, then the callback will be invoked. In a similar case in PyPy, both the object and the weakref will be considered as dead at the same time, and the callback will not be invoked. (Issue #2030)

There are a few extra implications from the difference in the GC. Most notably, if an object has a \_\_del\_\_, the \_\_del\_\_ is never called more than once in PyPy; but CPython will call the same \_\_del\_\_ several times if the object is resurrected and dies again (at least it is reliably so in older CPythons; newer CPythons try to call destructors not more than once, but there are counter-examples). The \_\_del\_\_ methods are called in “the right” order if they are on objects pointing to each other, as in CPython, but unlike CPython, if there is a dead cycle of objects referencing each other, their \_\_del\_\_ methods are called anyway; CPython would instead put them into the list garbage of the gc module. More information is available on the blog [1][2].

Note that this difference might show up indirectly in some cases. For example, a generator left pending in the middle is — again — garbage-collected later in PyPy than in CPython. You can see the difference if the yield keyword it is suspended at is itself enclosed in a try: or a with: block. This shows up for example as issue 736.

Using the default GC (called minimark), the built-in function id() works like it does in CPython. With other GCs it returns numbers that are not real addresses (because an object can move around several times) and calling it a lot can lead to performance problem.

Note that if you have a long chain of objects, each with a reference to the next one, and each with a \_\_del\_\_, PyPy’s GC will perform badly. On the bright side, in most other cases, benchmarks have shown that PyPy’s GCs perform much better than CPython’s.

Another difference is that if you add a \_\_del\_\_ to an existing class it will not be called:

```python
>>> class A(object):
    .... pass
    ....
>>> A._del__ = lambda self: None
__main__:1: RuntimeWarning: a _del__ method added to an existing type will not be called
```

Even more obscure: the same is true, for old-style classes, if you attach the \_\_del\_\_ to an instance (even in CPython this does not work with new-style classes). You get a RuntimeWarning in PyPy. To fix these cases just make sure there is a \_\_del\_\_ method in the class to start with (even containing only pass; replacing or overriding it later works fine).
Last note: CPython tries to do a `gc.collect()` automatically when the program finishes; not PyPy. (It is possible in both CPython and PyPy to design a case where several `gc.collect()` are needed before all objects die. This makes CPython's approach only work “most of the time” anyway.)

### 2.1.2 Subclasses of built-in types

Officially, CPython has no rule at all for when exactly overridden method of subclasses of built-in types get implicitly called or not. As an approximation, these methods are never called by other built-in methods of the same object. For example, an overridden `__getitem__()` in a subclass of `dict` will not be called by e.g. the built-in `get()` method.

The above is true both in CPython and in PyPy. Differences can occur about whether a built-in function or method will call an overridden method of another object than `self`. In PyPy, they are often called in cases where CPython would not. Two examples:

```python
class D(dict):
    def __getitem__(self, key):
        return "%r from D" % (key,)

class A(object):
    pass

a = A()
a.__dict__ = D()
a.foo = "a's own foo"
print a.foo
# CPython => a's own foo
# PyPy => 'foo' from D

glob = D(foo="base item")
loc = {}
exec "print foo" in glob, loc
# CPython => base item
# PyPy => 'foo' from D
```

### 2.1.3 Mutating classes of objects which are already used as dictionary keys

Consider the following snippet of code:

```python
class X(object):
    pass

def __evil_eq__(self, other):
    print 'hello world'
    return False

def evil(y):
    d = {X(): 1}
    X.__eq__ = __evil_eq__
    d[y] # might trigger a call to __eq__?
```

In CPython, `__evil_eq__` might be called, although there is no way to write a test which reliably calls it. It happens if `y` is not `x` and `hash(y) == hash(x)`, where `hash(x)` is computed when `x` is inserted into the dictionary. If by chance the condition is satisfied, then `__evil_eq__` is called.
PyPy uses a special strategy to optimize dictionaries whose keys are instances of user-defined classes which do not override the default \_\_hash\_, \_\_eq\_ and \_\_cmp\_; when using this strategy, \_\_eq\_ and \_\_cmp\_ are never called, but instead the lookup is done by identity, so in the case above it is guaranteed that \_\_eq\_ won’t be called.

Note that in all other cases (e.g., if you have a custom \_\_hash\_ and \_\_eq\_ in y) the behavior is exactly the same as CPython.

### 2.1.4 Ignored exceptions

In many corner cases, CPython can silently swallow exceptions. The precise list of when this occurs is rather long, even though most cases are very uncommon. The most well-known places are custom rich comparison methods (like \_\_eq\_), dictionary lookup; calls to some built-in functions like isinstance().

Unless this behavior is clearly present by design and documented as such (as e.g. for hasattr()), in most cases PyPy lets the exception propagate instead.

### 2.1.5 Object Identity of Primitive Values, \_\_is\_ and \_\_id\_

Object identity of primitive values works by value equality, not by identity of the wrapper. This means that x + 1 \_\_is\_ x + 1 is always true, for arbitrary integers x. The rule applies for the following types:

- int
- float
- long
- complex
- \texttt{str} (empty or single-character strings only)
- \texttt{unicode} (empty or single-character strings only)
- \texttt{tuple} (empty tuples only)
- \texttt{frozenset} (empty frozenset only)
- unbound method objects (for Python 2 only)

This change requires some changes to \_\_id\_ as well. \_\_id\_ fulfills the following condition: x \_\_is\_ y \iff \_\_id\_ (x) == \_\_id\_ (y). Therefore \_\_id\_ of the above types will return a value that is computed from the argument, and can thus be larger than \texttt{sys.maxint} (i.e. it can be an arbitrary long).

Note that strings of length 2 or greater can be equal without being identical. Similarly, x \_\_is\_ (2,) is not necessarily true even if x contains a tuple and x \_\_==\_ (2,). The uniqueness rules apply only to the particular cases described above. The \texttt{str}, \texttt{unicode}, \texttt{tuple} and \texttt{frozenset} rules were added in PyPy 5.4; before that, a test like \texttt{if x \_\_is\_ ?} or \texttt{if x \_\_is\_ ()} could fail even if x was equal to "?" or (). The new behavior added in PyPy 5.4 is closer to CPython’s, which caches precisely the empty tuple/frozenset, and (generally but not always) the strings and unicodes of length \_\_<=\_ 1.

Note that for floats there “is” only one object per “bit pattern” of the float. So float(’nan’) \_\_is\_ float(’nan’) is true on PyPy, but not on CPython because they are two objects; but 0.0 \_\_is\_ -0.0 is always False, as the bit patterns are different. As usual, float(’nan’) \_\_==\_ float(’nan’) is always False. When used in containers (as list items or in sets for example), the exact rule of equality used is “if x \_\_is\_ y or x \_\_==\_ y” (on both CPython and PyPy); as a consequence, because all nans are identical in PyPy, you cannot have several of them in a set, unlike in CPython. (Issue \#1974). Another consequence is that \texttt{cmp}(float(’nan’), float(’nan’)) \_\_==\_ 0, because \texttt{cmp} checks with \_\_is\_ first whether the arguments are identical (there is no good value to return from this call to \texttt{cmp}, because \texttt{cmp} pretends that there is a total order on floats, but that is wrong for NaNs).
2.1.6 C-API Differences

The external C-API has been reimplemented in PyPy as an internal cpyext module. We support most of the documented C-API, but sometimes internal C-abstractions leak out on CPython and are abused, perhaps even unknowingly. For instance, assignment to a PyTupleObject is not supported after the tuple is used internally, even by another C-API function call. On CPython this will succeed as long as the refcount is 1. On PyPy this will always raise a SystemError('PyTuple_SetItem called on tuple after use of tuple') exception (explicitly listed here for search engines).

Another similar problem is assignment of a new function pointer to any of the tp_as_* structures after calling PyType_Ready. For instance, overriding tp_as_number.nb_int with a different function after calling PyType_Ready on CPython will result in the old function being called for x.__int__() (via class __dict__ lookup) and the new function being called for int(x) (via slot lookup). On PyPy we will always call the __new__ function, not the old, this quirky behaviour is unfortunately necessary to fully support NumPy.

2.1.7 Performance Differences

CPython has an optimization that can make repeated string concatenation not quadratic. For example, this kind of code runs in O(n) time:

```python
s = ''
for string in mylist:
    s += string
```

In PyPy, this code will always have quadratic complexity. Note also, that the CPython optimization is brittle and can break by having slight variations in your code anyway. So you should anyway replace the code with:

```python
parts = []
for string in mylist:
    parts.append(string)
s = ''.join(parts)
```

2.1.8 Miscellaneous

- Hash randomization (−R) is ignored in PyPy. In CPython before 3.4 it has little point. Both CPython >= 3.4 and PyPy3 implement the randomized SipHash algorithm and ignore −R.
- You can’t store non-string keys in type objects. For example:

```python
class A(object):
    locals()[42] = 3
```

won’t work.
- `sys.setrecursionlimit(n)` sets the limit only approximately, by setting the usable stack space to n * 768 bytes. On Linux, depending on the compiler settings, the default of 768KB is enough for about 1400 calls.
- since the implementation of dictionary is different, the exact number of times that __hash__ and __eq__ are called is different. Since CPython does not give any specific guarantees either, don’t rely on it.
- assignment to __class__ is limited to the cases where it works on CPython 2.5. On CPython 2.6 and 2.7 it works in a bit more cases, which are not supported by PyPy so far. (If needed, it could be supported, but then it will likely work in many more case on PyPy than on CPython 2.6/2.7.)
• the __builtins__ name is always referencing the __builtin__ module, never a dictionary as it sometimes is in CPython. Assigning to __builtins__ has no effect. (For usages of tools like RestrictedPython, see issue #2653.)

• directly calling the internal magic methods of a few built-in types with invalid arguments may have a slightly different result. For example, [].__add__(None) and (2).__add__(None) both return NotImplemented on PyPy; on CPython, only the latter does, and the former raises TypeError. (Of course, [1]+None and 2+None both raise TypeError everywhere.) This difference is an implementation detail that shows up because of internal C-level slots that PyPy does not have.

• on CPython, [].__add__ is a method-wrapper, and list.__add__ is a slot wrapper. On PyPy these are normal bound or unbound method objects. This can occasionally confuse some tools that inspect built-in types. For example, the standard library inspect module has a function ismethod() that returns True on unbound method objects but False on method-wrappers or slot wrappers. On PyPy we can’t tell the difference, so ismethod([].__add__) == ismethod(list.__add__) == True.

• in CPython, the built-in types have attributes that can be implemented in various ways. Depending on the way, if you try to write to (or delete) a read-only (or undeletable) attribute, you get either a TypeError or an AttributeError. PyPy tries to strike some middle ground between full consistency and full compatibility here. This means that a few corner cases don’t raise the same exception, like del (lambda:None).__closure__.

• in pure Python, if you write class A(object): def f(self): pass and have a subclass B which doesn’t override f(), then B.f(x) still checks that x is an instance of B. In CPython, types written in C use a different rule. If A is written in C, any instance of A will be accepted by B.f(x) (and actually, B.f is A.f in this case). Some code that work on CPython but not on PyPy includes: datetime.datetime.strptime(datetime.datetime.today(), ...) (here, datetime.datetime is the superclass of datetime.datetime). Anyway, the proper fix is arguably to use a regular method call in the first place: datetime.datetime.today().strftime(...) (here, datetime.datetime is the superclass of datetime.datetime). Anyway, the proper fix is arguably to use a regular method call in the first place: datetime.datetime.today().strftime(...)

• some functions and attributes of the gc module behave in a slightly different way: for example, gc.enable and gc.disable are supported, but “enabling and disabling the GC” has a different meaning in PyPy than in CPython. These functions actually enable and disable the major collections and the execution of finalizers.

• PyPy prints a random line from past #pypy IRC topics at startup in interactive mode. In a released version, this behaviour is suppressed, but setting the environment variable PYPY_IRC_TOPIC will bring it back. Note that downstream package providers have been known to totally disable this feature.

• PyPy’s readline module was rewritten from scratch: it is not GNU’s readline. It should be mostly compatible, and it adds multiline support (see multiline_input()). On the other hand, parse_and_bind() calls are ignored (issue #2072).

• sys.getsizeof() always raises TypeError. This is because a memory profiler using this function is most likely to give results inconsistent with reality on PyPy. It would be possible to have sys.getsizeof() return a number (with enough work), but that may or may not represent how much memory the object uses. It doesn’t even make really sense to ask how much one object uses, in isolation with the rest of the system. For example, instances have maps, which are often shared across many instances; in this case the maps would probably be ignored by an implementation of sys.getsizeof(), but their overhead is important in some cases if they are many instances with unique maps. Conversely, equal strings may share their internal string data even if they are different objects—or empty containers may share parts of their internals as long as they are empty. Even stranger, some lists create objects as you read them; if you try to estimate the size in memory of range(10**6) as the sum of all items’ size, that operation will by itself create one million integer objects that never existed in the first place. Note that some of these concerns also exist on CPython, just less so. For this reason we explicitly don’t implement sys.getsizeof().

• The timeit module behaves differently under PyPy: it prints the average time and the standard deviation, instead of the minimum, since the minimum is often misleading.
• The get_config_vars method of sysconfig and distutils.sysconfig are not complete. On POSIX platforms, CPython fishes configuration variables from the Makefile used to build the interpreter. PyPy should bake the values in during compilation, but does not do that yet.

• "%d" % x and "%x" % x and similar constructs, where x is an instance of a subclass of long that overrides the special methods __str__ or __hex__ or __oct__: PyPy doesn’t call the special methods; CPython does—but only if it is a subclass of long, not int. CPython’s behavior is really messy: e.g. for %x it calls __hex__(), which is supposed to return a string like -0x123L; then the 0x and the final L are removed, and the rest is kept. If you return an unexpected string from __hex__() you get an exception (or a crash before CPython 2.7.13).

• In PyPy, dictionaries passed as **kwargs can contain only string keys, even for dict() and dict.update(). CPython 2.7 allows non-string keys in these two cases (and only there, as far as we know). E.g. this code produces a TypeError, on CPython 3.x as well as on any PyPy: dict(**{1: 2}). (Note that dict(**dl) is equivalent to dict(dl).

• PyPy3: __class__ attribute assignment between heatypes and non heatypes. CPython allows that for module subtypes, but not for e.g. int or float subtypes. Currently PyPy does not support the __class__ attribute assignment for any non heapt-type subtype.

• In PyPy, module and class dictionaries are optimized under the assumption that deleting attributes from them are rare. Because of this, e.g. del foo.bar where foo is a module (or class) that contains the function bar, is significantly slower than CPython.

• Various built-in functions in CPython accept only positional arguments and not keyword arguments. That can be considered a long-running historical detail: newer functions tend to accept keyword arguments and older function are occasionally fixed to do so as well. In PyPy, most built-in functions accept keyword arguments (help() shows the argument names). But don’t rely on it too much because future versions of PyPy may have to rename the arguments if CPython starts accepting them too.

• PyPy3: distutils has been enhanced to allow finding VsDevCmd.bat in the directory pointed to by the VS%0.f0COMNTOOLS (typically VS140COMNTOOLS) environment variable. CPython searches for vcvarsall.bat somewhere above that value.

• SyntaxError s try harder to give details about the cause of the failure, so the error messages are not the same as in CPython

• Dictionaries and sets are ordered on PyPy. On CPython < 3.6 they are not; on CPython >= 3.6 dictionaries (but not sets) are ordered.

• PyPy2 refuses to load lone .pyc files, i.e. .pyc files that are still there after you deleted the .py file. PyPy3 instead behaves like CPython. We could be amenable to fix this difference in PyPy2: the current version reflects our annoyance with this detail of CPython, which bit us too often while developing PyPy. (It is as easy as passing the --lonepycfile flag when translating PyPy, if you really need it.)

2.1.9 Extension modules

List of extension modules that we support:

• Supported as built-in modules (in pypy/module/):

  __builtin__ __pypy__ _ast _codecs _collections _continuation _ffi _hashlib _io _locale _lsprof _md5 _minimal_curses _multiprocessing _random _rawffi _socket _sre _ssl _warnings _weakref _winreg array binascii bz2 cStringIO cmath cpyext crypt errno exceptions fcntl gc imp itertools marshal math mmap operator parser posix pyexpat select signal struct symbol sys termios thread time token unicodedata zipfile zlib

When translated on Windows, a few Unix-only modules are skipped, and the following module is built instead:

  _winreg

2.1. Differences between PyPy and CPython
• Supported by being rewritten in pure Python (possibly using cffi): see the lib_pypy/ directory. Examples of modules that we support this way: ctypes, cPickle,cmath, dbm, datetime... Note that some modules are both in there and in the list above; by default, the built-in module is used (but can be disabled at translation time).

The extension modules (i.e. modules written in C, in the standard CPython) that are neither mentioned above nor in lib_pypy/ are not available in PyPy. (You may have a chance to use them anyway with cpyext.)

2.2 Writing extension modules for pypy

This document tries to explain how to interface the PyPy python interpreter with any external library.

Right now, there are the following possibilities of providing third-party modules for the PyPy python interpreter (in order, from most directly useful to most messy to use with PyPy):

• Write them in pure Python and use CFFI.
• Write them in pure Python and use ctypes.
• Write them in C++ and bind them through cppyy using Cling.
• Write them as RPython mixed modules.

2.2.1 CFFI

CFFI is the recommended way. It is a way to write pure Python code that accesses C libraries. The idea is to support either ABI- or API-level access to C — so that you can sanely access C libraries without depending on details like the exact field order in the C structures or the numerical value of all the constants. It works on both CPython (as a separate pip install cffi) and on PyPy, where it is included by default.

PyPy’s JIT does a quite reasonable job on the Python code that call C functions or manipulate C pointers with CFFI. (As of PyPy 2.2.1, it could still be improved, but is already good.)

See the documentation here.

2.2.2 CTypes

The goal of the ctypes module of PyPy is to be as compatible as possible with the CPython ctypes version. It works for large examples, such as pyglet. PyPy’s implementation is not strictly 100% compatible with CPython, but close enough for most cases. More (but older) information is available here. Also, ctypes’ performance is not as good as CFFI’s.

PyPy implements ctypes as pure Python code around two built-in modules called _rawffi and _rawffi.alt, which give a very low-level binding to the C library libffi. Nowadays it is not recommended to use directly these two modules.

2.2.3 cppyy

For C++, _cppyy_ is an automated bindings generator available for both PyPy and CPython. _cppyy_ relies on declarations from C++ header files to dynamically construct Python equivalent classes, functions, variables, etc. It is designed for use by large scale programs and supports modern C++. With PyPy, it leverages the built-in _cppyy_ module, allowing the JIT to remove most of the cross-language overhead.

To install, run pip install cppyy. Further details are available in the full documentation.
2.2.4 RPython Mixed Modules

This is the internal way to write built-in extension modules in PyPy. It cannot be used by any 3rd-party module: the extension modules are built-in, not independently loadable DLLs.

This is reserved for special cases: it gives direct access to e.g. the details of the JIT, allowing us to tweak its interaction with user code. This is how the numpy module is being developed.

2.3 Garbage collector documentation and configuration

2.3.1 Incminimark

PyPy’s default garbage collector is called incminimark - it’s an incremental, generational moving collector. Here we hope to explain a bit how it works and how it can be tuned to suit the workload.

Incminimark first allocates objects in so called nursery - place for young objects, where allocation is very cheap, being just a pointer bump. The nursery size is a very crucial variable - depending on your workload (one or many processes) and cache sizes you might want to experiment with it via PYPY_GC_NURSERY environment variable. When the nursery is full, there is performed a minor collection. Freed objects are no longer referencable and just die, just by not being referenced any more; on the other hand, objects found to still be alive must survive and are copied from the nursery to the old generation. Either to arenas, which are collections of objects of the same size, or directly allocated with malloc if they’re larger. (A third category, the very large objects, are initially allocated outside the nursery and never move.)

Since Incminimark is an incremental GC, the major collection is incremental: the goal is not to have any pause longer than 1ms, but in practice it depends on the size and characteristics of the heap: occasionally, there can be pauses between 10-100ms.

2.3.2 Semi-manual GC management

If there are parts of the program where it is important to have a low latency, you might want to control precisely when the GC runs, to avoid unexpected pauses. Note that this has effect only on major collections, while minor collections continue to work as usual.

As explained above, a full major collection consists of N steps, where N depends on the size of the heap; generally speaking, it is not possible to predict how many steps will be needed to complete a collection.

gc.enable() and gc.disable() control whether the GC runs collection steps automatically. When the GC is disabled the memory usage will grow indefinitely, unless you manually call gc.collect() and gc.collect_step().

gc.collect() runs a full major collection.

gc.collect_step() runs a single collection step. It returns an object of type GcCollectStepStats, the same which is passed to the corresponding GC Hooks. The following code is roughly equivalent to gc.collect():

```python
while True:
    if gc.collect_step().major_is_done:
        break
```

For a real-world example of usage of this API, you can look at the 3rd-party module pypytools.gc.custom, which also provides a with customgc.nogc() context manager to mark sections where the GC is forbidden.
2.3.3 Fragmentation

Before we discuss issues of “fragmentation”, we need a bit of precision. There are two kinds of related but distinct issues:

- If the program allocates a lot of memory, and then frees it all by dropping all references to it, then we might expect to see the RSS to drop. (RSS = Resident Set Size on Linux, as seen by “top”; it is an approximation of the actual memory usage from the OS’s point of view.) This might not occur: the RSS may remain at its highest value. This issue is more precisely caused by the process not returning “free” memory to the OS. We call this case “unreturned memory”.

- After doing the above, if the RSS didn’t go down, then at least future allocations should not cause the RSS to grow more. That is, the process should reuse unreturned memory as long as it has got some left. If this does not occur, the RSS grows even larger and we have real fragmentation issues.

2.3.4 gc.get_stats

There is a special function in the gc module called get_stats(memory_pressure=False).

memory_pressure controls whether or not to report memory pressure from objects allocated outside of the GC, which requires walking the entire heap, so it’s disabled by default due to its cost. Enable it when debugging mysterious memory disappearance.

Example call looks like that:

```python
>>> gc.get_stats(True)
Total memory consumed:
GC used: 4.2MB (peak: 4.2MB)
  in arenas: 763.7kB
  rawmalloced: 383.1kB
  nursery: 3.1MB
raw assembler used: 0.0kB
memory pressure: 0.0kB
----------------------------
Total: 4.2MB

Total memory allocated:
GC allocated: 4.5MB (peak: 4.5MB)
  in arenas: 763.7kB
  rawmalloced: 383.1kB
  nursery: 3.1MB
raw assembler allocated: 0.0kB
memory pressure: 0.0kB
----------------------------
Total: 4.5MB
```

In this particular case, which is just at startup, GC consumes relatively little memory and there is even less unused, but allocated memory. In case there is a lot of unreturned memory or actual fragmentation, the “allocated” can be much higher than “used”. Generally speaking, “peak” will more closely resemble the actual memory consumed as reported by RSS. Indeed, returning memory to the OS is a hard and not solved problem. In PyPy, it occurs only if an arena is entirely free—a contiguous block of 64 pages of 4 or 8 KB each. It is also rare for the “rawmalloced” category, at least for common system implementations of malloc().

The details of various fields:

- GC in arenas - small old objects held in arenas. If the amount “allocated” is much higher than the amount “used”, we have unreturned memory. It is possible but unlikely that we have internal fragmentation here. However,
this unreturned memory cannot be reused for any `malloc()`, including the memory from the “rawmalloced” section.

- GC rawmalloced - large objects allocated with malloc. This is gives the current (first block of text) and peak (second block of text) memory allocated with `malloc()`. The amount of unreturned memory or fragmentation caused by `malloc()` cannot easily be reported. Usually you can guess there is some if the RSS is much larger than the total memory reported for “GC allocated”, but do keep in mind that this total does not include malloc’ed memory not known to PyPy’s GC at all. If you guess there is some, consider using jemalloc as opposed to system malloc.

- nursery - amount of memory allocated for nursery, fixed at startup, controlled via an environment variable
- raw assembler allocated - amount of assembler memory that JIT feels responsible for
- memory pressure, if asked for - amount of memory we think got allocated via external malloc (e.g., loading cert store in SSL contexts) that is kept alive by GC objects, but not accounted in the GC

### 2.3.5 GC Hooks

GC hooks are user-defined functions which are called whenever a specific GC event occur, and can be used to monitor GC activity and pauses. You can install the hooks by setting the following attributes:

- `gc.hook.on_gc_minor` Called whenever a minor collection occurs. It corresponds to `gc-minor` sections inside PYPYLOG.

- `gc.hook.on_gc_collect_step` Called whenever an incremental step of a major collection occurs. It corresponds to `gc-collect-step` sections inside PYPYLOG.

- `gc.hook.on_gc_collect` Called after the last incremental step, when a major collection is fully done. It corresponds to `gc-collect-done` sections inside PYPYLOG.

To uninstall a hook, simply set the corresponding attribute to `None`. To install all hooks at once, you can call `gc.hooks.set(obj)`, which will look for methods `on_gc_*` on `obj`. To uninstall all the hooks at once, you can call `gc.hooks.reset()`.

The functions called by the hooks receive a single `stats` argument, which contains various statistics about the event.

Note that PyPy cannot call the hooks immediately after a GC event, but it has to wait until it reaches a point in which the interpreter is in a known state and calling user-defined code is harmless. It might happen that multiple events occur before the hook is invoked: in this case, you can inspect the value `stats.count` to know how many times the event occurred since the last time the hook was called. Similarly, `stats.duration` contains the total time spent by the GC for this specific event since the last time the hook was called.

On the other hand, all the other fields of the `stats` object are relative only to the last event of the series.

The attributes for `GcMinorStats` are:

- `count` The number of minor collections occurred since the last hook call.
- `duration` The total time spent inside minor collections since the last hook call, in seconds.
- `duration_min` The duration of the fastest minor collection since the last hook call.
- `duration_max` The duration of the slowest minor collection since the last hook call.
- `total_memory_used` The amount of memory used at the end of the minor collection, in bytes. This include the memory used in arenas (for GC-managed memory) and raw-malloced memory (e.g., the content of numpy arrays).
- `pinned_objects` the number of pinned objects.
The attributes for `GcCollectStepStats` are:

- `count`, `duration`, `duration_min`, `duration_max` See above.
- `oldstate`, `newstate` Integers which indicate the state of the GC before and after the step.
- `major_is_done` Boolean which indicate whether this was the last step of the major collection.

The value of `oldstate` and `newstate` is one of these constants, defined inside `gc.GcCollectStepStats`: `STATE_SCANNING`, `STATE_MARKING`, `STATE_SWEEPING`, `STATE_FINALIZING`, `STATE_USERDEL`. It is possible to get a string representation of it by indexing the `GC_STATES` tuple.

The attributes for `GcCollectStats` are:

- `count` See above.
- `num_major_collects` The total number of major collections which have been done since the start. Contrarily to `count`, this is an always-growing counter and it’s not reset between invocations.
- `arenas_count_before`, `arenas_count_after` Number of arenas used before and after the major collection.
- `arenas_bytes` Total number of bytes used by GC-managed objects.
- `rawmalloc_bytes_before`, `rawmalloc_bytes_after` Total number of bytes used by raw-malloced objects, before and after the major collection.

Note that `GcCollectStats` has **not** got a `duration` field. This is because all the GC work is done inside `gc-collect-step: gc-collect-done` is used only to give additional stats, but doesn’t do any actual work.

Here is an example of GC hooks in use:

```python
class MyHooks(object):
    done = False

    def on_gc_minor(self, stats):
        print 'gc-minor: count = %02d, duration = %d' % (stats.count, stats.duration)

    def on_gc_collect_step(self, stats):
        old = gc.GcCollectStepStats.GC_STATES[stats.oldstate]
        new = gc.GcCollectStepStats.GC_STATES[stats.newstate]
        print 'gc-collect-step: %s --> %s' % (old, new)
        print ' count = %02d, duration = %d' % (stats.count, stats.duration)

    def on_gc_collect(self, stats):
        print 'gc-collect-done: count = %02d' % stats.count
        self.done = True

hooks = MyHooks()
gc.hooks.set(hooks)

# simulate some GC activity
lst = []
while not hooks.done:
    lst = [lst, 1, 2, 3]
```

```python
import sys
import gc

class MyHooks(object):
    done = False

    def on_gc_minor(self, stats):
        print 'gc-minor: count = %02d, duration = %d' % (stats.count, stats.duration)

    def on_gc_collect_step(self, stats):
        old = gc.GcCollectStepStats.GC_STATES[stats.oldstate]
        new = gc.GcCollectStepStats.GC_STATES[stats.newstate]
        print 'gc-collect-step: %s --> %s' % (old, new)
        print ' count = %02d, duration = %d' % (stats.count, stats.duration)

    def on_gc_collect(self, stats):
        print 'gc-collect-done: count = %02d' % stats.count
        self.done = True

hooks = MyHooks()
gc.hooks.set(hooks)

# simulate some GC activity
lst = []
while not hooks.done:
    lst = [lst, 1, 2, 3]
```
2.3.6 Environment variables

PyPy's default incminimark garbage collector is configurable through several environment variables:

**PYPY_GC_NURSERY** The nursery size. Defaults to 1/2 of your last-level cache, or 4M if unknown. Small values (like 1 or 1KB) are useful for debugging.

**PYPY_GC_NURSERY_DEBUG** If set to non-zero, will fill nursery with garbage, to help debugging.

**PYPY_GC_INCREMENT_STEP** The size of memory marked during the marking step. Default is size of nursery times 2. If you mark it too high your GC is not incremental at all. The minimum is set to size that survives minor collection times 1.5 so we reclaim anything all the time.

**PYPY_GC_MAJOR_COLLECT** Major collection memory factor. Default is 1.82, which means trigger a major collection when the memory consumed equals 1.82 times the memory really used at the end of the previous major collection.

**PYPY_GC_GROWTH** Major collection threshold’s max growth rate. Default is 1.4. Useful to collect more often than normally on sudden memory growth, e.g. when there is a temporary peak in memory usage.

**PYPY_GC_MAX** The max heap size. If coming near this limit, it will first collect more often, then raise an RPython MemoryError, and if that is not enough, crash the program with a fatal error. Try values like 1.6GB.

**PYPY_GC_MAX_DELTA** The major collection threshold will never be set to more than PYPY_GC_MAX_DELTA the amount really used after a collection. Defaults to 1/8th of the total RAM size (which is constrained to be at most 2/3/4GB on 32-bit systems). Try values like 200MB.

**PYPY_GC_MIN** Don’t collect while the memory size is below this limit. Useful to avoid spending all the time in the GC in very small programs. Defaults to 8 times the nursery.

**PYPY_GC_DEBUG** Enable extra checks around collections that are too slow for normal use. Values are 0 (off), 1 (on major collections) or 2 (also on minor collections).

**PYPY_GC_MAX_PINNED** The maximal number of pinned objects at any point in time. Defaults to a conservative value depending on nursery size and maximum object size inside the nursery. Useful for debugging by setting it to 0.

2.4 JIT hooks

There are several hooks in the pypyjit module that may help you with understanding what’s pypy’s JIT doing while running your program. There are three functions related to that coming from the pypyjit module:

**set_compile_hook** *(callable, operations=True)*

Set a compiling hook that will be called each time a loop is compiled.

The callable will be called with the pypyjit.JitLoopInfo object. Refer to it's documentation for details.

Note that jit hook is not reentrant. It means that if the code inside the jit hook is itself jitted, it will get compiled, but the jit hook won’t be called for that.

if operations=False, no list of operations will be available. Useful if the hook is supposed to be very lightweight.

**set_abort_hook** *(hook)*

Set a hook (callable) that will be called each time there is tracing aborted due to some reason.

The hook will be invoked with the signature: hook(jitdriver_name, greenkey, reason, oplist)

Reason is a string, the meaning of other arguments is the same as attributes on JitLoopInfo object.
enable_debug()

Start recording debugging counters for get_stats_snapshot

disable_debug()

Stop recording debugging counters for get_stats_snapshot

get_stats_snapshot()

Get the jit status in the specific moment in time. Note that this is eager - the attribute access is not lazy, if you need new stats you need to call this function again. You might want to call enable_debug to get more information. It returns an instance of JitInfoSnapshot

class JitInfoSnapshot

A class describing current snapshot. Usable attributes:

- counters - internal JIT integer counters
- counter_times - internal JIT float counters, notably time spent TRACING and in the JIT BACKEND
- loop_run_times - counters for number of times loops are run, only works when enable_debug is called.

class JitLoopInfo

A class containing information about the compiled loop. Usable attributes:

- operations - list of operations, if requested
- jitdriver_name - the name of jitdriver associated with this loop
- greenkey - a key at which the loop got compiled (e.g. code position, is_being_profiled, pycode tuple for python jitdriver)
- loop_no - loop cardinal number
- bridge_no - id of the fail descr
- type - “entry bridge”, “loop” or “bridge”
- asmaddr - an address in raw memory where assembler resides
- asmlen - length of raw memory with assembler associated

2.5 Application-level Stackless features

2.5.1 Introduction

PyPy can expose to its user language features similar to the ones present in Stackless Python: the ability to write code in a massively concurrent style. (It does not (any more) offer the ability to run with no recursion depth limit, but the same effect can be achieved indirectly.)

This feature is based on a custom primitive called a continulet. Continulets can be directly used by application code, or it is possible to write (entirely at app-level) more user-friendly interfaces.

Currently PyPy implements greenlets on top of continulets. It also implements (an approximation of) tasklets and channels, emulating the model of Stackless Python.

Continulets are extremely light-weight, which means that PyPy should be able to handle programs containing large amounts of them. However, due to an implementation restriction, a PyPy compiled with --grootfinder=shadowstack consumes at least one page of physical memory (4KB) per live continulet, and half a megabyte of virtual memory on 32-bit or a complete megabyte on 64-bit. Moreover, the feature is only available (so far) on x86 and x86-64 CPUs; for other CPUs you need to add a short page of custom assembler to rpython/translator/c/src/stacklet/.
2.5.2 Theory

The fundamental idea is that, at any point in time, the program happens to run one stack of frames (or one per thread, in case of multi-threading). To see the stack, start at the top frame and follow the chain of \texttt{f\_back} until you reach the bottom frame. From the point of view of one of these frames, it has a \texttt{f\_back} pointing to another frame (unless it is the bottom frame), and it is itself being pointed to by another frame (unless it is the top frame).

The theory behind continulets is to literally take the previous sentence as definition of “an O.K. situation”. The trick is that there are O.K. situations that are more complex than just one stack: you will always have one stack, but you can also have in addition one or more detached cycles of frames, such that by following the \texttt{f\_back} chain you run in a circle. But note that these cycles are indeed completely detached: the top frame (the currently running one) is always the one which is not the \texttt{f\_back} of anybody else, and it is always the top of a stack that ends with the bottom frame, never a part of these extra cycles.

How do you create such cycles? The fundamental operation to do so is to take two frames and permute their \texttt{f\_back} — i.e. exchange them. You can permute any two \texttt{f\_back} without breaking the rule of “an O.K. situation”. Say for example that \(f\) is some frame halfway down the stack, and you permute its \texttt{f\_back} with the \texttt{f\_back} of the top frame. Then you have removed from the normal stack all intermediate frames, and turned them into one stand-alone cycle. By doing the same permutation again you restore the original situation.

In practice, in PyPy, you cannot change the \texttt{f\_back} of an arbitrary frame, but only of frames stored in continulets. Continulets are internally implemented using stacklets. Stacklets are a bit more primitive (they are really one-shot continuations), but that idea only works in C, not in Python. The basic idea of continulets is to have at any point in time a complete valid stack; this is important e.g. to correctly propagate exceptions (and it seems to give meaningful tracebacks too).

2.5.3 Application level interface

Continulets

A translated PyPy contains by default a module called \_continuation exporting the type \texttt{continulet}. A \texttt{continulet} object from this module is a container that stores a “one-shot continuation”. It plays the role of an extra frame you can insert in the stack, and whose \texttt{f\_back} can be changed.

To make a continulet object, call \texttt{continulet()} with a callable and optional extra arguments.

Later, the first time you \texttt{switch()} to the continulet, the callable is invoked with the same continulet object as the extra first argument. At that point, the one-shot continuation stored in the continulet points to the caller of \texttt{switch()}. In other words you have a perfectly normal-looking stack of frames. But when \texttt{switch()} is called again, this stored one-shot continuation is exchanged with the current one; it means that the caller of \texttt{switch()} is suspended with its continuation stored in the container, and the old continuation from the continulet object is resumed.

The most primitive API is actually ‘permute()’, which just permutes the one-shot continuation stored in two (or more) continulets.

In more details:

- \texttt{continulet(callable, *args, **kwds)}: make a new continulet. Like a generator, this only creates it; the callable is only actually called the first time it is switched to. It will be called as follows:

\begin{verbatim}
callable(cont, *args, **kwds)
\end{verbatim}

where \texttt{cont} is the same continulet object.

Note that it is actually \texttt{cont.__init__()} that binds the continulet. It is also possible to create a not-bound-yet continulet by calling explicitly \texttt{continulet.__new__()}, and only bind it later by calling explicitly \texttt{cont.__init__()}.
• cont.switch(value=None, to=None): start the continulet if it was not started yet. Otherwise, store
the current continuation in cont, and activate the target continuation, which is the one that was previously stored
in cont. Note that the target continuation was itself previously suspended by another call to switch(); this
older switch() will now appear to return. The value argument is any object that is carried to the target and
returned by the target’s switch().

If to is given, it must be another continulet object. In that case, performs a “double switch”: it switches as
described above to cont, and then immediately switches again to to. This is different from switching directly
to: the current continuation gets stored in cont, the old continuation from cont gets stored in to, and only
then we resume the execution from the old continuation out of to.

• cont.throw(type, value=None, tb=None, to=None): similar to switch(), except that imme-
diately after the switch is done, raise the given exception in the target.

• cont.is_pending(): return True if the continulet is pending. This is False when it is not initialized
(because we called __new__ and not __init__) or when it is finished (because the callable() returned).
When it is False, the continulet object is empty and cannot be switch()-ed to.

• permute(*continulets): a global function that permutes the continuations stored in the given continulets
arguments. Mostly theoretical. In practice, using cont.switch() is easier and more efficient than using
permute(); the latter does not on its own change the currently running frame.

Genlets

The _continuation module also exposes the generator decorator:

```python
@generator
def f(cont, a, b):
    cont.switch(a + b)
    cont.switch(a + b + 1)

for i in f(10, 20):
    print i
```

This example prints 30 and 31. The only advantage over using regular generators is that the generator itself is not
limited to yield statements that must all occur syntactically in the same function. Instead, we can pass around
cont, e.g. to nested sub-functions, and call cont.switch(x) from there.

The generator decorator can also be applied to methods:

```python
class X:
    @generator
    def f(self, cont, a, b):
        ...
```

Greenlets

Greenlets are implemented on top of continulets in lib_pypy/greenlet.py. See the official documentation of the green-
lets.

Note that unlike the CPython greenlets, this version does not suffer from GC issues: if the program “forgets” an
unfinished greenlet, it will always be collected at the next garbage collection.

Unimplemented features

The following features (present in some past Stackless version of PyPy) are for the time being not supported any more:
• Coroutines (could be rewritten at app-level)
• Continuing execution of a continulet in a different thread (but if it is “simple enough”, you can pickle it and unpickle it in the other thread).
• Automatic unlimited stack (must be *emulated* so far)
• Support for other CPUs than x86 and x86-64

We also do not include any of the recent API additions to Stackless Python, like `set_atomic()`. Contributions welcome.

### Recursion depth limit

You can use continulets to emulate the infinite recursion depth present in Stackless Python and in stackless-enabled older versions of PyPy.

The trick is to start a continulet “early”, i.e. when the recursion depth is very low, and switch to it “later”, i.e. when the recursion depth is high. Example:

```python
from _continuation import continulet

def invoke(_, callable, arg):
    return callable(arg)

def bootstrap(c):
    # this loop runs forever, at a very low recursion depth
    callable, arg = c.switch()
    while True:
        # start a new continulet from here, and switch to
        # it using an "exchange", i.e. a switch with to=.  
        to = continulet(involve, callable, arg)
        callable, arg = c.switch(to=to)

c = continulet(bootstrap)
c.switch()

def recursive(n):
    if n == 0:
        return ("ok", n)
    if n % 200 == 0:
        prev = c.switch((recursive, n - 1))
    else:
        prev = recursive(n - 1)
    return (prev[0], prev[1] + 1)

print recursive(999999)  # prints ("ok", 999999)
```

Note that if you press Ctrl-C while running this example, the traceback will be built with *all* recursive() calls so far, even if this is more than the number that can possibly fit in the C stack. These frames are “overlapping” each other in the sense of the C stack; more precisely, they are copied out of and into the C stack as needed.

(The example above also makes use of the following general “guideline” to help newcomers write continulets: in `bootstrap(c)`, only call methods on `c`, not on another continulet object. That’s why we wrote `c.switch(to=to)` and not `to.switch()`, which would mess up the state. This is however just a guideline; in general we would recommend to use other interfaces like genlets and greenlets.)
Stacklets

Continules are internally implemented using stacklets, which is the generic RPython-level building block for “one-shot continuations”. For more information about them please see the documentation in the C source at rpython/translator/c/src/stacklet/stacklet.h.

The module rpython.rlib.rstacklet is a thin wrapper around the above functions. The key point is that new() and switch() always return a fresh stacklet handle (or an empty one), and switch() additionally consumes one. It makes no sense to have code in which the returned handle is ignored, or used more than once. Note that stacklet.c is written assuming that the user knows that, and so no additional checking occurs; this can easily lead to obscure crashes if you don’t use a wrapper like PyPy’s `_continuation` module.

Theory of composability

Although the concept of coroutines is far from new, they have not been generally integrated into mainstream languages, or only in limited form (like generators in Python and iterators in C#). We can argue that a possible reason for that is that they do not scale well when a program’s complexity increases: they look attractive in small examples, but the models that require explicit switching, for example by naming the target coroutine, do not compose naturally. This means that a program that uses coroutines for two unrelated purposes may run into conflicts caused by unexpected interactions.

To illustrate the problem, consider the following example (simplified code using a theoretical coroutine class). First, a simple usage of coroutine:

```python
main_coro = coroutine.getcurrent()   # the main (outer) coroutine
data = []

def data_producer():
    for i in range(10):
        # add some numbers to the list 'data' ...
data.append(i)
data.append(i * 5)
data.append(i * 25)
    # and then switch back to main to continue processing
    main_coro.switch()

producer_coro = coroutine()
producer_coro.bind(data_producer)

def grab_next_value():
    if not data:
        # put some more numbers in the 'data' list if needed
        producer_coro.switch()
    # then grab the next value from the list
    return data.pop(0)
```

Every call to grab_next_value() returns a single value, but if necessary it switches into the producer function (and back) to give it a chance to put some more numbers in it.

Now consider a simple reimplementation of Python’s generators in term of coroutines:

```python
def generator(f):
    """Wrap a function 'f' so that it behaves like a generator."""
    def wrappedfunc(*args, **kwds):
        g = generator_iterator()
        g.bind(f, *args, **kwds)
        return g
```

(continues on next page)
Both these examples are attractively elegant. However, they cannot be composed. If we try to write the following generator:

```python
def grab_values(n):
    for i in range(n):
        Yield(grab_next_value())
grab_values = generator(grab_values)
```

then the program does not behave as expected. The reason is the following. The generator coroutine that executes `grab_values()` calls `grab_next_value()`, which may switch to the `producer_coro` coroutine. This works so far, but the switching back from `data_producer()` to `main_coro` lands in the wrong coroutine: it resumes execution in the main coroutine, which is not the one from which it comes. We expect `data_producer()` to switch back to the `grab_next_values()` call, but the latter lives in the generator coroutine `g` created in `wrappedfunc`, which is totally unknown to the `data_producer()` code. Instead, we really switch back to the main coroutine, which confuses the `generator_iterator.next()` method (it gets resumed, but not as a result of a call to `Yield()`).

Thus the notion of coroutine is not composable. By opposition, the primitive notion of continulets is composable: if you build two different interfaces on top of it, or have a program that uses twice the same interface in two parts, then assuming that both parts independently work, the composition of the two parts still works.

A full proof of that claim would require careful definitions, but let us just claim that this fact is true because of the following observation: the API of continulets is such that, when doing a `switch()`, it requires the program to have some continulet to explicitly operate on. It shuffles the current continuation with the continuation stored in that continulet, but has no effect outside. So if a part of a program has a continulet object, and does not expose it as a global, then the rest of the program cannot accidentally influence the continuation stored in that continulet object.

In other words, if we regard the continulet object as being essentially a modifiable `f_back`, then it is just a link between the frame of `callable()` and the parent frame — and it cannot be arbitrarily changed by unrelated code, as long as they don’t explicitly manipulate the continulet object. Typically, both the frame of `callable()` (commonly a local function) and its parent frame (which is the frame that switched to it) belong to the same class or module; so from that point of view the continulet is a purely local link between two local frames. It doesn’t make sense to have a
concept that allows this link to be manipulated from outside.

2.6 The __pypy__ module

The __pypy__ module is the main entry point to special features provided by PyPy’s standard interpreter. Its content depends on configuration options which may add new functionality and functions whose existence or non-existence indicates the presence of such features. These are generally used for compatibility when writing pure python modules that in CPython are written in C. Not available in CPython, and so must be used inside a if platform.

python_implementation == 'PyPy' block or otherwise hidden from the CPython interpreter.

2.6.1 Generally available functionality

- internal_repr(obj): return the interpreter-level representation of an object.
- bytebuffer(length): return a new read-write buffer of the given length. It works like a simplified array of characters (actually, depending on the configuration the array module internally uses this).
- attach_gdb(): start a GDB at the interpreter-level (or a PDB before translation).
- newmemoryview(buffer, itemsize, format, shape=None, strides=None): create a memoryview instance with the data from buffer and the specified itemsize, format, and optional shape and strides.
- bufferable: a base class that provides a __buffer__(self, flags) method for subclasses to override. This method should return a memoryview instance of the class instance. It is called by the C-API's tp_as_buffer. bf_getbuffer.
- builtinify(func): To implement at app-level modules that are, in CPython, implemented in C: this decorator protects a function from being ever bound like a method. Useful because some tests do things like put a “built-in” function on a class and access it via the instance.
- hidden_applevel(func): Decorator that hides a function’s frame from app-level
- get_hidden_tb(): Return the traceback of the current exception being handled by a frame hidden from applevel.
- lookup_special(obj, meth): Lookup up a special method on an object.
- do_what_I_mean
- resizelist_hint(...): Reallocate the underlying storage of the argument list to sizehint
- newlist_hint(...): Create a new empty list that has an underlying storage of length sizehint
- add_memory_pressure(bytes): Add memory pressure of estimate bytes. Useful when calling a C function that internally allocates a big chunk of memory. This instructs the GC to garbage collect sooner than it would otherwise.
- newdict(type): Create a normal dict with a special implementation strategy. type is a string and can be:
  - "module" - equivalent to some_module.__dict__
  - "instance" - equivalent to an instance dict with a not-changing-much set of keys
  - "kwargs" - keyword args dict equivalent of what you get from **kwargs in a function, optimized for passing around
  - "strdict" - string-key only dict. This one should be chosen automatically
• **reversed_dict**: Enumerate the keys in a dictionary object in reversed order. This is a __pypy__ function instead of being simply done by calling reversed(), for CPython compatibility: dictionaries are ordered in PyPy but not in CPython 2.7. You should use the collections.OrderedDict class for cases where ordering is important. That class implements __reversed__ by calling __pypy__.reversed_dict().

• **dict_popitem_first**: Interp-level implementation of OrderedDict. popitem(last=False).

• **delitem_if_value_is** Atomic equivalent to: if dict.get(key) is value: del dict[key].

SPECIAL USE CASES ONLY! Avoid using on dicts which are specialized, e.g. to int or str keys, because it switches to the object strategy. Also, the is operation is really pointer equality, so avoid using it if value is an immutable object like int or str.

• **move_to_end**: Move the key in a dictionary object into the first or last position. This is used in Python 3.x to implement OrderedDict.move_to_end().

• **strategy(dict or list or set)**: Return the underlying strategy currently used by the object

• **specialized_zip_2_lists**

• **locals_to_fast**

• **set_code_callback**

• **save_module_content_for_future_reload**

• **decode_long**

• **side_effects_ok**: For use with the reverse-debugger: this function normally returns True, but will return False if we are evaluating a debugging command like a watchpoint. You are responsible for not doing any side effect at all (including no caching) when evaluating watchpoints. This function is meant to help a bit—you can write:

```python
if not __pypy__.side_effects_ok():
    skip the caching logic
```

inside getter methods or properties, to make them usable from watchpoints. Note that you need to re-run REVD=.. pypy after changing the Python code.

• **stack_almost_full**: Return True if the stack is more than 15/16th full.

• **pyos_inputhook**: Call PyOS_InputHook() from the CPython C API

• **os.real_getenv (...) gets OS environment variables skipping python code**

• **_pypydatetime** provides base classes with correct C API interactions for the pure-python datetime stdlib module

### 2.6.2 Fast String Concatenation

Rather than in-place concatenation +=, use these to enable fast, minimal copy, string building.

• **builders.StringBuilder**

• **builders.UnicodeBuilder**
2.6.3 Interacting with the PyPy debug log

The following functions can be used to write your own content to the PYPYLOG.

- `debug_start(category, timestamp=False)`: open a new section; if `timestamp` is True, also return the timestamp which was written to the log.
- `debug_stop(category, timestamp=False)`: close a section opened by `debug_start`.
- `debug_print(...)` : print arbitrary text to the log.
- `debug_print_once(category, ...)`: equivalent to `debug_start + debug_print + debug_stop`.
- `debug_flush`: flush the log.
- `debug_read_timestamp()`: read the timestamp from the same timer used by the log.
- `debug_get_timestamp_unit()`: get the unit of the value returned by `debug_read_timestamp()`.

Depending on the architecture and operating system, PyPy uses different ways to read timestamps, so the timestamps used in the log file are expressed in varying units. It is possible to know which by calling `debug_get_timestamp_unit()`, which can be one of the following values:

- **tsc** The default on x86 machines: timestamps are expressed in CPU ticks, as read by the Time Stamp Counter.
- **ns** Timestamps are expressed in nanoseconds.

**QueryPerformanceCounter** On Windows, in case for some reason tsc is not available: timestamps are read using the win API `QueryPerformanceCounter()`.

Unfortunately, there does not seem to be a reliable standard way for converting tsc ticks into nanoseconds, although in practice on modern CPUs it is enough to divide the ticks by the maximum nominal frequency of the CPU. For this reason, PyPy gives the raw value, and leaves the job of doing the conversion to external libraries.

2.6.4 Transparent Proxy Functionality

If transparent proxies are enabled (with –objspace-std-withtproxy) the following functions are put into __pypy__:

- `tproxy(typ, controller)`: Return something that looks like it is of type typ. Its behaviour is completely controlled by the controller. See the docs about transparent proxies for detail.
- `get_tproxy_controller(obj)`: If obj is really a transparent proxy, return its controller. Otherwise return None.

2.6.5 Additional Clocks for Timing

The time submodule exposes the platform-dependent clock types such as CLOCK_BOOTTIME, CLOCK_MONOTONIC, CLOCK_MONOTONIC_COARSE, CLOCK_MONOTONIC_RAW and two functions:

- `clock_gettime(m)` which returns the clock type time in seconds and
- `clock_getres(m)` which returns the clock resolution in seconds.

2.6.6 Extended Signal Handling

`thread.signals_enabled` is a context manager to use in non-main threads. enables receiving signals in a “with” statement. More precisely, if a signal is received by the process, then the signal handler might be called either in the main thread (as usual) or within another thread that is within a “with signals_enabled:”. This
other thread should be ready to handle unexpected exceptions that the signal handler might raise — notably KeyboardInterrupt.

2.6.7 Integer Operations with Overflow

• intop provides a module with integer operations that have two-complement overflow behaviour instead of overflowing to longs

2.6.8 Functionality available on py.py (not after translation)

• isfake(obj): returns True if obj is faked.

2.7 PyPy’s sandboxing features

Warning: This is not actively maintained. You will likely have to fix some issues yourself, or otherwise play around on your own. We provide this documentation for historical reasons, it will not translate or run on the latest PyPy code base.

2.7.1 Introduction

PyPy offers sandboxing at a level similar to OS-level sandboxing (e.g. SECCOMP on Linux), but implemented in a fully portable way. To use it, a (regular, trusted) program launches a subprocess that is a special sandboxed version of PyPy. This subprocess can run arbitrary untrusted Python code, but all its input/output is serialized to a stdin/stdout pipe instead of being directly performed. The outer process reads the pipe and decides which commands are allowed or not (sandboxing), or even reinterprets them differently (virtualization). A potential attacker can have arbitrary code run in the subprocess, but cannot actually do any input/output not controlled by the outer process. Additional barriers are put to limit the amount of RAM and CPU time used.

Note that this is very different from sandboxing at the Python language level, i.e. placing restrictions on what kind of Python code the attacker is allowed to run (why? read about pysandbox).

Another point of comparison: if we were instead to try to plug CPython into a special virtualizing C library, we would get a result that is not only OS-specific, but unsafe, because CPython can be segfaulted (in many ways, all of them really, really obscure). Given enough efforts, an attacker can turn almost any segfault into a vulnerability. The C code generated by PyPy is not segfaultable, as long as our code generators are correct - that’s a lower number of lines of code to trust. For the paranoid, PyPy translated with sandboxing also contains systematic run-time checks (against buffer overflows for example) that are normally only present in debugging versions.

Warning: The hard work from the PyPy side is done — you get a fully secure version. What is only experimental and unpolished is the library to use this sandboxed PyPy from a regular Python interpreter (CPython, or an unsandboxed PyPy). Contributions welcome.

2.7.2 Overview

One of PyPy’s translation aspects is a sandboxing feature. It’s “sandboxing” as in “full virtualization”, but done in normal C with no OS support at all. It’s a two-processes model: we can translate PyPy to a special “pypy-c-sandbox” executable, which is safe in the sense that it doesn’t do any library or system calls - instead, whenever it would like to perform such an operation, it marshals the operation name and the arguments to its stdout and it waits for the marshalled result on its stdin. This pypy-c-sandbox process is meant to be run by an outer “controller” program that answers these operation requests.

The pypy-c-sandbox program is obtained by adding a transformation during translation, which turns all RPython-level external function calls into stubs that do the marshalling/waiting/unmarshalling. An attacker that tries to escape the sandbox is stuck within a C program that contains no external function calls at all except for writing to stdout and reading from stdin. (It’s still attackable in theory, e.g. by exploiting segfault-like situations, but as explained in the introduction we think that PyPy is rather safe against such attacks.)

The outer controller is a plain Python program that can run in CPython or a regular PyPy. It can perform any virtualization it likes, by giving the subprocess any custom view on its world. For example, while the subprocess thinks it’s using file handles, in reality the numbers are created by the controller process and so they need not be (and probably should not be) real OS-level file handles at all. In the demo controller I’ve implemented there is simply a mapping from numbers to file-like objects. The controller answers to the “os_open” operation by translating the requested path to some file or file-like object in some virtual and completely custom directory hierarchy. The file-like object is put in the mapping with any unused number >= 3 as a key, and the latter is returned to the subprocess. The “os_read” operation works by mapping the pseudo file handle given by the subprocess back to a file-like object in the controller, and reading from the file-like object.

Translating an RPython program with sandboxing enabled also uses a special flag that enables all sorts of C-level assertions against index-out-of-bounds accesses.

By the way, as you should have realized, it’s really independent from the fact that it’s PyPy that we are translating. Any RPython program should do. I’ve successfully tried it on the JS interpreter. The controller is only called “pypy_interact” because it emulates a file hierarchy that makes pypy-c-sandbox happy - it contains (read-only) virtual directories like /bin/lib/pypy1.2/lib-python and /bin/lib/pypy1.2/lib_pypy and it pretends that the executable is /bin/pypy-c.

2.7.3 Howto

Grab a copy of the pypy repository. In the directory pypy/goal, run:

```
../../../rpython/bin/rpython -O2 --sandbox targetpypystandalone.py
```

If you don’t have a regular PyPy installed, you should, because it’s faster to translate; but you can also run the same line with python in front.

To run it, use the tools in the pypy/sandbox directory:

```
./pypy_interact.py /some/path/pypy-c-sandbox [args...]
```

Just like with pppy-c, if you pass no argument you get the interactive prompt. In theory it’s impossible to do anything bad or read a random file on the machine from this prompt. To pass a script as an argument you need to put it in a directory along with all its dependencies, and ask pypy_interact to export this directory (read-only) to the subprocess’ virtual /tmp directory with the --tmp=DIR option. Example:

```
mkdir myexported
cp script.py myexported/
./pypy_interact.py --tmp=myexported /some/path/pypy-c-sandbox /tmp/script.py
```
This is safe to do even if script.py comes from some random untrusted source, e.g. if it is done by an HTTP server.
To limit the used heapsize, use the \texttt{--heapsize=N} option to \texttt{pypy_interact.py}. You can also give a limit to the CPU time (real time) by using the \texttt{--timeout=N} option.

Not all operations are supported; e.g. if you type \texttt{os.readlink(‘...’)}, the controller crashes with an exception and the subprocess is killed. Other operations make the subprocess die directly with a “Fatal RPython error”. None of this is a security hole. More importantly, most other built-in modules are not enabled. Please read all the warnings in this page before complaining about this. Contributions welcome.

2.8 Software Transactional Memory

This page is about \texttt{pypy-stm}, a special in-development version of PyPy which can run multiple independent CPU-hungry threads in the same process in parallel. It is a solution to what is known in the Python world as the “global interpreter lock (GIL)” problem — it is an implementation of Python without the GIL.

“STM” stands for Software Transactional Memory, the technique used internally. This page describes \texttt{pypy-stm} from the perspective of a user, describes work in progress, and finally gives references to more implementation details.

This work was done by Remi Meier and Armin Rigo. Thanks to all donors for crowd-funding the work so far! Please have a look at the 2nd call for donation.
2.8.1 What pypy-stm is for

pypy-stm is a variant of the regular PyPy interpreter. (This version supports Python 2.7; see below for Python 3, CPython, and others.) With caveats listed below, it should be in theory within 20%-50% slower than a regular PyPy, comparing the JIT version in both cases (but see below!). It is called STM for Software Transactional Memory, which is the internal technique used (see Reference to implementation details).

The benefit is that the resulting pypy-stm can execute multiple threads of Python code in parallel. Programs running two threads or more in parallel should ideally run faster than in a regular PyPy (either now, or soon as bugs are fixed).

- pypy-stm is fully compatible with a GIL-based PyPy; you can use it as a drop-in replacement and multi-threaded programs will run on multiple cores.
- pypy-stm provides (but does not impose) a special API to the user in the pure Python module transaction. This module is based on the lower-level module pypystm, but also provides some compatibility with non-STM PyPy’s or CPython’s.
- Building on top of the way the GIL is removed, we will talk about How to write multithreaded programs: the 10’000-feet view and transaction.TransactionQueue.

... and what pypy-stm is not for

pypy-stm gives a Python without the GIL. This means that it is useful in situations where the GIL is the problem in the first place. (This includes cases where the program can easily be modified to run in multiple threads; often, we don’t consider doing that precisely because of the GIL.)

However, there are plenty of cases where the GIL is not the problem. Do not hope pypy-stm to be helpful in these cases! This includes all programs that use multiple threads but don’t actually spend a lot of time running Python code. For example, it may be spending all its time waiting for I/O to occur, or performing some long computation on a huge matrix. These are cases where the CPU is either idle, or in some C/Fortran library anyway; in both cases, the interpreter (either CPython or the regular PyPy) should release the GIL around the external calls. The threads will thus not end up fighting for the GIL.

2.8.2 Getting Started

pypy-stm requires 64-bit Linux for now.

Development is done in the branch stmgc-c8. If you are only interested in trying it out, please pester us until we upload a recent prebuilt binary. The current version supports four “segments”, which means that it will run up to four threads in parallel.

To build a version from sources, you first need to compile a custom version of gcc(!). See the instructions here: https://bitbucket.org/pypy/stmgc/src/default/gcc-seg-gs/ (Note that these patches are being incorporated into gcc. It is likely that future versions of gcc will not need to be patched any more.)

Then get the branch stmgc-c8 of PyPy and run:

```
cd pypy/goal
../../../../rpython/bin/rpython -Ojit --stm
```

At the end, this will try to compile the generated C code by calling gcc-seg-gs, which must be the script you installed in the instructions above.
Current status (stmgc-c7)

**Warning:** THIS PAGE IS OLD, THE REST IS ABOUT STMGC-C7 WHEREAS THE CURRENT DEVELOPMENT WORK IS DONE ON STMGC-C8

- **NEW:** It seems to work fine, without crashing any more. Please report any crash you find (or other bugs).

- It runs with an overhead as low as 20% on examples like “richards”. There are also other examples with higher overheads – currently up to 2x for “translate.py” – which we are still trying to understand. One suspect is our partial GC implementation, see below.

- **NEW:** the `PYPYSTM` environment variable and the `pypy/stm/print_stm_log.py` script let you know exactly which “conflicts” occurred. This is described in the section `transaction.TransactionQueue` below.

- **NEW:** special transaction-friendly APIs (like `stmdict`), described in the section `transaction.TransactionQueue` below. The old API changed again, mostly moving to different modules. Sorry about that. I feel it’s a better idea to change the API early instead of being stuck with a bad one later...

- Currently limited to 1.5 GB of RAM (this is just a parameter in `core.h` – theoretically. In practice, increase it too much and clang crashes again). Memory overflows are not correctly handled; they cause segfaults.

- **NEW:** The JIT warm-up time improved again, but is still relatively large. In order to produce machine code, the JIT needs to enter “inevitable” mode. This means that you will get bad performance results if your program doesn’t run for several seconds, where `several` can mean many. When trying benchmarks, be sure to check that you have reached the warmed state, i.e. the performance is not improving any more.

- The GC is new; although clearly inspired by PyPy’s regular GC, it misses a number of optimizations for now. Programs allocating large numbers of small objects that don’t immediately die (surely a common situation) suffer from these missing optimizations. (The bleeding edge `stmgc-c8` is better at that.)

- Weakrefs might appear to work a bit strangely for now, sometimes staying alive through `gc.collect()`, or even dying but then un-dying for a short time before dying again. A similar problem can show up occasionally elsewhere with accesses to some external resources, where the (apparent) serialized order doesn’t match the underlying (multithreading) order. These are bugs (partially fixed already in `stmgc-c8`). Also, debugging helpers like `weakref.getweakrefcount()` might give wrong answers.

- The STM system is based on very efficient read/write barriers, which are mostly done (their placement could be improved a bit in JIT-generated machine code).

- Forking the process is slow because the complete memory needs to be copied manually. A warning is printed to this effect.

- Very long-running processes (on the order of days) will eventually crash on an assertion error because of a non-implemented overflow of an internal 28-bit counter.

- The recursion detection code was not reimplemented. Infinite recursion just segfaults for now.

### 2.8.3 Python 3, CPython, and others

In this document I describe “pypy-stm”, which is based on PyPy’s Python 2.7 interpreter. Supporting Python 3 should take about half an afternoon of work. Obviously, what I don’t mean is that by tomorrow you can have a finished and polished “pypy3-stm” product. General py3k work is still missing; and general stm work is also still missing. But they are rather independent from each other, as usual in PyPy. The required afternoon of work will certainly be done one of these days now that the internal interfaces seem to stabilize.

The same is true for other languages implemented in the RPython framework, although the amount of work to put there might vary, because the STM framework within RPython is currently targeting the PyPy interpreter and other
ones might have slightly different needs. But in general, all the tedious transformations are done by RPython and you’re only left with the (hopefully few) hard and interesting bits.

The core of STM works as a library written in C (see reference to implementation details below). It means that it can be used on other interpreters than the ones produced by RPython. Duhton is an early example of that. At this point, you might think about adapting this library for CPython. You’re warned, though: as far as I can tell, it is a doomed idea. I had a hard time debugging Duhton, and that’s infinitely simpler than CPython. Even ignoring that, you can see in the C sources of Duhton that many core design decisions are different than in CPython: no refcounting; limited support for prebuilt “static” objects; \texttt{stm\_read()} and \texttt{stm\_write()} macro calls everywhere (and getting very rare and very obscure bugs if you forget one); and so on. You could imagine some custom special-purpose extension of the C language, which you would preprocess to regular C. In my opinion that’s starting to look a lot like RPython itself, but maybe you’d prefer this approach. Of course you still have to worry about each and every C extension module you need, but maybe you’d have a way forward.

\section*{2.8.4 User Guide}

\textbf{How to write multithreaded programs: the 10’000-feet view}

PyPy-STM offers two ways to write multithreaded programs:

- the traditional way, using the thread or threading modules, described \textit{first}.

- using TransactionQueue, described \textit{next}, as a way to hide the low-level notion of threads.

The issues with low-level threads are well known (particularly in other languages that don’t have GIL-based interpreters): memory corruption, deadlocks, livelocks, and so on. There are alternative approaches to dealing directly with threads, like OpenMP. These approaches typically enforce some structure on your code. TransactionQueue is in part similar: your program needs to have “some chances” of parallelization before you can apply it. But I believe that the scope of applicability is much larger with TransactionQueue than with other approaches. It usually works without forcing a complete reorganization of your existing code, and it works on any Python program which has got latent and imperfect parallelism. Ideally, it only requires that the end programmer identifies where this parallelism is likely to be found, and communicates it to the system using a simple API.

\textbf{Drop-in replacement}

Multithreaded, CPU-intensive Python programs should work unchanged on pypy-stm. They will run using multiple CPU cores in parallel.

The existing semantics of the GIL (Global Interpreter Lock) are unchanged: although running on multiple cores in parallel, pypy-stm gives the illusion that threads are run serially, with switches only occurring between bytecodes, not in the middle of them. Programs can rely on this: using \texttt{shared\_list.append()/pop()} or \texttt{shared\_dict.setdefault()} as synchronization mechanisms continues to work as expected.

This works by internally considering the points where a standard PyPy or CPython would release the GIL, and replacing them with the boundaries of “transactions”. Like their database equivalent, multiple transactions can execute in parallel, but will commit in some serial order. They appear to behave as if they were completely run in this serialization order.

\textbf{transaction.TransactionQueue}

In CPU-hungry programs, we can often easily identify outermost loops over some data structure, or other repetitive algorithm, where each “block” consists of processing a non-trivial amount of data, and where the blocks “have a good chance” to be independent from each other. We don’t need to prove that they are actually independent: it is enough if they are often independent — or, more precisely, if we think they should be often independent.
One typical example would look like this, where the function `func()` typically invokes a large amount of code:

```python
for key, value in bigdict.items():
    func(key, value)
```

Then you simply replace the loop with:

```python
from transaction import TransactionQueue

tr = TransactionQueue()
for key, value in bigdict.items():
    tr.add(func, key, value)
tr.run()
```

This code’s behavior is equivalent. Internally, the `TransactionQueue` object will start N threads and try to run the `func(key, value)` calls on all threads in parallel. But note the difference with a regular thread-pooling library, as found in many lower-level languages than Python: the function calls are not randomly interleaved with each other just because they run in parallel. The behavior did not change because we are using `TransactionQueue`. All the calls still appear to execute in some serial order.

A typical usage of `TransactionQueue` goes like that: at first, the performance does not increase. In fact, it is likely to be worse. Typically, this is indicated by the total CPU usage, which remains low (closer to 1 than N cores). First note that it is expected that the CPU usage should not go much higher than 1 in the JIT warm-up phase: you must run a program for several seconds, or for larger programs at least one minute, to give the JIT a chance to warm up enough. But if CPU usage remains low even afterwards, then the `PYPYSTM` environment variable can be used to track what is going on.

Run your program with `PYPYSTM=logfile` to produce a log file called `logfile`. Afterwards, use the `pypy/stm/print_stm_log.py` utility to inspect the content of this log file. It produces output like this (sorted by amount of time lost, largest first):

```
10.5s lost in aborts, 1.25s paused (12412x STM_CONTENTION_WRITE_WRITE)
File "foo.py", line 10, in f
    someobj.stuff = 5
File "bar.py", line 20, in g
    someobj.other = 10
```

This means that 10.5 seconds were lost running transactions that were aborted (which caused another 1.25 seconds of lost time by pausing), because of the reason shown in the two independent single-entry tracebacks: one thread ran the line `someobj.stuff = 5`, whereas another thread concurrently ran the line `someobj.other = 10` on the same object. These two writes are done to the same object. This causes a conflict, which aborts one of the two transactions. In the example above this occurred 12412 times.

The two other conflict sources are `STM_CONTENTION_INEVITABLE`, which means that two transactions both tried to do an external operation, like printing or reading from a socket or accessing an external array of raw data; and `STM_CONTENTION_WRITE_READ`, which means that one transaction wrote to an object but the other one merely read it, not wrote to it (in that case only the writing transaction is reported; the location for the reads is not recorded because doing so is not possible without a very large performance impact).

Common causes of conflicts:

- First of all, any I/O or raw manipulation of memory turns the transaction inevitable (“must not abort”). There can be only one inevitable transaction running at any time. A common case is if each transaction starts with sending data to a log file. You should refactor this case so that it occurs either near the end of the transaction (which can then mostly run in non-inevitable mode), or delegate it to a separate transaction or even a separate thread.
- Writing to a list or a dictionary conflicts with any read from the same list or dictionary, even one done with a different key. For dictionaries and sets, you can try the types `transaction.stmdict` and `transaction.2.8. Software Transactional Memory
stmset, which behave mostly like dict and set but allow concurrent access to different keys. (What is missing from them so far is lazy iteration: for example, stmdict.iterkeys() is implemented as iter(stmdict.keys()); and, unlike PyPy’s dictionaries and sets, the STM versions are not ordered.) There are also experimental stmiddict and stmidset classes using the identity of the key.

• time.time() and time.clock() turn the transaction inevitable in order to guarantee that a call that appears to be later will really return a higher number. If getting slightly unordered results is fine, use transaction.time() or transaction.clock(). The latter operations guarantee to return increasing results only if you can “prove” that two calls occurred in a specific order (for example because they are both called by the same thread). In cases where no such proof is possible, you might get randomly interleaved values. (If you have two independent transactions, they normally behave as if one of them was fully executed before the other; but using transaction.time() you might see the “hidden truth” that they are actually interleaved.)

• transaction.threadlocalproperty can be used at class-level:

```python
class Foo(object):  # must be a new-style class!
    x = transaction.threadlocalproperty()
    y = transaction.threadlocalproperty(dict)
```

This declares that instances of Foo have two attributes x and y that are thread-local: reading or writing them from concurrently-running transactions will return independent results. (Any other attributes of Foo instances will be globally visible from all threads, as usual.) This is useful together with TransactionQueue for these two cases:

- For attributes of long-lived objects that change during one transaction, but should always be reset to some initial value around transaction (for example, initialized to 0 at the start of a transaction; or, if used for a list of pending things to do within this transaction, it will always be empty at the end of one transaction).

- For general caches across transactions. With TransactionQueue you get a pool of a fixed number N of threads, each running the transactions serially. A thread-local property will have the value last stored in it by the same thread, which may come from a random previous transaction. Basically, you get N copies of the property’s value, and each transaction accesses a random copy. It works fine for caches.

In more details, the optional argument to threadlocalproperty() is the default value factory: in case no value was assigned in the current thread yet, the factory is called and its result becomes the value in that thread (like collections.defaultdict). If no default value factory is specified, uninitialized reads raise AttributeError.

• In addition to all of the above, there are cases where write-write conflicts are caused by writing the same value to an attribute again and again. See for example ea2e5f19614ab: this fixes two such issues where we write an object field without first checking if we already did it. The dont_change_any_more field is a flag set to True in that part of the code, but usually this rtyper_makekey() method will be called many times for the same object; the code used to repeatedly set the flag to True, but now it first checks and only does the write if it is False. Similarly, in the second half of the checkin, the method setup_block_entry() used to both assign the concretetype fields and return a list, but its two callers were different: one would really need the concretetype fields initialized, whereas the other would only need to get its result list — the concretetype field in that case might already be set or not, but that would not matter.

Note that Python is a complicated language; there are a number of less common cases that may cause conflict (of any kind) where we might not expect it at priori. In many of these cases it could be fixed; please report any case that you don’t understand.

**Atomic sections**

The TransactionQueue class described above is based on atomic sections, which are blocks of code which you want to execute without “releasing the GIL”. In STM terms, this means blocks of code that are executed while...
guaranteeing that the transaction is not interrupted in the middle. *This is experimental and may be removed in the future if Software lock elision is ever implemented.*

Here is a direct usage example:

```python
with transaction.atomic:
    assert len(lst1) == 10
    x = lst1.pop(0)
    lst1.append(x)
```

In this example, we are sure that the item popped off one end of the list is appended again at the other end atomically. It means that another thread can run `len(lst1)` or `x in lst1` without any particular synchronization, and always see the same results, respectively 10 and True. It will never see the intermediate state where `lst1` only contains 9 elements. Atomic sections are similar to re-entrant locks (they can be nested), but additionally they protect against the concurrent execution of *any* code instead of just code that happens to be protected by the same lock in other threads.

Note that the notion of atomic sections is very strong. If you write code like this:

```python
with __pypy__.thread.atomic:
    time.sleep(10)
```

then, if you think about it as if we had a GIL, you are executing a 10-seconds-long atomic transaction without releasing the GIL at all. This prevents all other threads from progressing at all. While it is not strictly true in pypy-stm, the exact rules for when other threads can progress or not are rather complicated; you have to consider it likely that such a piece of code will eventually block all other threads anyway.

Note that if you want to experiment with a *atomic* block, you may have to manually add a transaction break just before the atomic block. This is because the boundaries of the block are not guaranteed to be the boundaries of the transaction: the latter is at least as big as the block, but may be bigger. Therefore, if you run a big atomic block, it is a good idea to break the transaction just before. This can be done by calling `transaction.hint_commit_soon()`. (This may be fixed at some point.)

There are also issues with the interaction of regular locks and atomic blocks. This can be seen if you write to files (which have locks), including with a `print` to standard output. If one thread tries to acquire a lock while running in an atomic block, and another thread has got the same lock at that point, then the former may fail with a `thread.error`. (Don’t rely on it; it may also deadlock.) The reason is that “waiting” for some condition to become true – while running in an atomic block – does not really make sense. For now you can work around it by making sure that, say, all your prints are either in an atomic block or none of them are. (This kind of issue is theoretically hard to solve and may be the reason for atomic block support to eventually be removed.)

## Locks

### Not Implemented Yet

The thread module’s locks have their basic semantic unchanged. However, using them (e.g. in `with my_lock: blocks`) starts an alternative running mode, called Software lock elision. This means that PyPy will try to make sure that the transaction extends until the point where the lock is released, and if it succeeds, then the acquiring and releasing of the lock will be “elided”. This means that in this case, the whole transaction will technically not cause any write into the lock object — it was unacquired before, and is still unacquired after the transaction.

This is specially useful if two threads run `with my_lock: blocks` with the same lock. If they each run a transaction that is long enough to contain the whole block, then all writes into the lock will be elided and the two transactions will not conflict with each other. As usual, they will be serialized in some order: one of the two will appear to run before the other. Simply, each of them executes an “acquire” followed by a “release” in the same transaction. As explained above, the lock state goes from “unacquired” to “unacquired” and can thus be left unchanged.

This approach can gracefully fail: unlike atomic sections, there is no guarantee that the transaction runs until the end of the block. If you perform any input/output while you hold the lock, the transaction will end as usual just before the
input/output operation. If this occurs, then the lock elision mode is cancelled and the lock’s “acquired” state is really written.

Even if the lock is really acquired already, a transaction doesn’t have to wait for it to become free again. It can enter the elision-mode anyway and tentatively execute the content of the block. It is only at the end, when trying to commit, that the thread will pause. As soon as the real value stored in the lock is switched back to “unacquired”, it can then proceed and attempt to commit its already-executed transaction (which can fail and abort and restart from the scratch, as usual).

Note that this is all not implemented yet, but we expect it to work even if you acquire and release several locks. The elision-mode transaction will extend until the first lock you acquired is released, or until the code performs an input/output or a wait operation (for example, waiting for another lock that is currently not free). In the common case of acquiring several locks in nested order, they will all be elided by the same transaction.

**Miscellaneous functions**

- First, note that the `transaction` module is found in the file `lib_pypy/transaction.py`. This file can be copied around to execute the same programs on CPython or on non-STM PyPy, with fall-back behavior. (One case where the behavior differs is `atomic`, which in this fall-back case just a regular lock; so with `atomic` only prevent other threads from entering other with `atomic` sections, but won’t prevent other threads from running non-atomic code.)

- `transaction.getsegmentlimit()`: return the number of “segments” in this pypy-stm. This is the limit above which more threads will not be able to execute on more cores. (Right now it is limited to 4 due to inter-segment overhead, but should be increased in the future. It should also be settable, and the default value should depend on the number of actual CPUs.) If STM is not available, this returns 1.

- `__pypy__.thread.signals_enabled`: a context manager that runs its block of code with signals enabled. By default, signals are only enabled in the main thread; a non-main thread will not receive signals (this is like CPython). Enabling signals in non-main threads is useful for libraries where threads are hidden and the end user is not expecting his code to run elsewhere than in the main thread.

- `pypystm.exclusive_atomic`: a context manager similar to `transaction.atomic` but which complains if it is nested.

- `transaction.is_atomic()`: return True if called from an atomic context.

- `pypystm.count()`: return a different positive integer every time it is called. This works without generating conflicts. The returned integers are only roughly in increasing order; this should not be relied upon.

**More details about conflicts**

Based on Software Transactional Memory, the `pypy-stm` solution is prone to “conflicts”. To repeat the basic idea, threads execute their code speculatively, and at known points (e.g. between bytecodes) they coordinate with each other to agree on which order their respective actions should be “committed”, i.e. become globally visible. Each duration of time between two commit-points is called a transaction.

A conflict occurs when there is no consistent ordering. The classical example is if two threads both tried to change the value of the same global variable. In that case, only one of them can be allowed to proceed, and the other one must be either paused or aborted (restarting the transaction). If this occurs too often, parallelization fails.

How much actual parallelization a multithreaded program can see is a bit subtle. Basically, a program not using `transaction.atomic` or eliding locks, or doing so for very short amounts of time, will parallelize almost freely (as long as it’s not some artificial example where, say, all threads try to increase the same global counter and do nothing else).

However, if the program requires longer transactions, it comes with less obvious rules. The exact details may vary from version to version, too, until they are a bit more stabilized. Here is an overview.
Parallelization works as long as two principles are respected. The first one is that the transactions must not conflict with each other. The most obvious sources of conflicts are threads that all increment a global shared counter, or that all store the result of their computations into the same list — or, more subtly, that all pop() the work to do from the same list, because that is also a mutation of the list. (You can work around it with `transaction.stmdict`, but for that specific example, some STM-aware queue should eventually be designed.)

A conflict occurs as follows: when a transaction commits (i.e. finishes successfully) it may cause other transactions that are still in progress to abort and retry. This is a waste of CPU time, but even in the worst case scenario it is not worse than a GIL, because at least one transaction succeeds (so we get at worst N-1 CPUs doing useless jobs and 1 CPU doing a job that commits successfully).

Conflicts do occur, of course, and it is pointless to try to avoid them all. For example they can be abundant during some warm-up phase. What is important is to keep them rare enough in total.

Another issue is that of avoiding long-running so-called “inevitable” transactions (“inevitable” is taken in the sense of “which cannot be avoided”, i.e. transactions which cannot abort any more). Transactions like that should only occur if you use `atomic`, generally because of I/O in atomic blocks. They work, but the transaction is turned inevitable before the I/O is performed. For all the remaining execution time of the atomic block, they will impede parallel work. The best is to organize the code so that such operations are done completely outside `atomic`.

(This is not unrelated to the fact that blocking I/O operations are discouraged with Twisted, and if you really need them, you should do them on their own separate thread.)

In case lock elision eventually replaces atomic sections, we wouldn’t get long-running inevitable transactions, but the same problem occurs in a different way: doing I/O cancels lock elision, and the lock turns into a real lock. This prevents other threads from committing if they also need this lock. (More about it when lock elision is implemented and tested.)

### 2.8.5 Implementation

XXX this section mostly empty for now

#### Technical reports

STMGC-C7 is described in detail in a technical report.

A separate position paper gives an overview of our position about STM in general.

#### Reference to implementation details

The core of the implementation is in a separate C library called `stmgc`, in the `c7` subdirectory (current version of pypy-stm) and in the `c8` subdirectory (bleeding edge version). Please see the `README.txt` for more information. In particular, the notion of segment is discussed there.

PyPy itself adds on top of it the automatic placement of read and write barriers and of “becomes-inevitable-now” barriers, the logic to start/stop transactions as an RPython transformation and as supporting C code, and the support in the JIT (mostly as a transformation step on the trace and generation of custom assembler in `assembler.py`).

### 2.8.6 See also

See also https://bitbucket.org/pypy/pypy/raw/default/pypy/doc/project-ideas.rst (section about STM).
CHAPTER 3

Development

3.1 Contributing Guidelines

Contents

• Contributing Guidelines
  – Getting involved
  – Your first contribution
    • Source Control
    • Fork & Clone
    • Edit
    • Pull Request
  – Architecture
    • Where to start?
  – Building
    • Coding Guide
  – Testing
    • Test driven development
    • py.test and the py lib
    • Running PyPy’s unit tests
  – Testing After Translation
  – Tooling & Utilities
PyPy is a very large project that has a reputation of being hard to dive into. Some of this fame is warranted, some of it is purely accidental. There are three important lessons that everyone willing to contribute should learn:

- **PyPy has layers.** There are many pieces of architecture that are very well separated from each other. More about this below, but often the manifestation of this is that things are at a different layer than you would expect them to be. For example if you are looking for the JIT implementation, you will not find it in the implementation of the Python programming language.

- **Because of the above, we are very serious about Test Driven Development.** It’s not only what we believe in, but also that PyPy’s architecture is working very well with TDD in mind and not so well without it. Often development means progressing in an unrelated corner, one unittest at a time; and then flipping a giant switch, bringing it all together. (It generally works out of the box. If it doesn’t, then we didn’t write enough unit tests.) It’s worth repeating - PyPy’s approach is great if you do TDD, and not so great otherwise.

- **PyPy uses an entirely different set of tools - most of them included in the PyPy repository.** There is no Makefile, nor autoconf. More below.

The first thing to remember is that PyPy project is very different than most projects out there. It’s also different from a classic compiler project, so academic courses about compilers often don’t apply or lead in the wrong direction. However, if you want to understand how designing & building a runtime works in the real world then this is a great project!

### 3.1.1 Getting involved

PyPy employs a relatively standard open-source development process. You are encouraged as a first step to join our pypy-dev mailing list and IRC channel, details of which can be found in our contact section. The folks there are very friendly, and can point you in the right direction.

We give out commit rights usually fairly liberally, so if you want to do something with PyPy, you can become a committer. We also run frequent coding sprints which are separately announced and often happen around Python conferences such as EuroPython or PyCon. Upcoming events are usually announced on the blog.

Further Reading: Contact

### 3.1.2 Your first contribution

The first and most important rule how not to contribute to PyPy is “just hacking a feature”. This won’t work, and you’ll find your PR will typically require a lot of re-work. There are a few reasons why not:

- build times are large
- PyPy has very thick layer separation
- context of the cPython runtime is often required

Instead, reach out on the dev mailing list or the IRC channel, and we’re more than happy to help! :) Some ideas for first contributions are:

- Documentation - this will give you an understanding of the pypy architecture
• Test failures - find a failing test in the nightly builds, and fix it
• Missing language features - these are listed in our issue tracker

Source Control

PyPy development is based a typical fork/pull request based workflow, centered around Mercurial (hg), hosted on Bitbucket. If you have not used this workflow before, a good introduction can be found here:

https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow

The cycle for a new PyPy contributor goes typically like this:

Fork & Clone

• Make an account on bitbucket.

• Go to https://bitbucket.org/pypy/pypy/ and click "fork" (left icons). You get a fork of the repository, e.g. in https://bitbucket.org/yourname/pypy/.

• Clone your new repo (i.e. the fork) to your local machine with the command `hg clone ssh://hg@bitbucket.org/yourname/pypy`. It is a very slow operation but only ever needs to be done once. See also http://pypy.org/download.html#building-from-source. If you already cloned https://bitbucket.org/pypy/pypy before, even if some time ago, then you can reuse the same clone by editing the file `.hg/hgrc` in your clone to contain the line `default = ssh://hg@bitbucket.org/yourname/pypy`, and then do `hg pull` & `hg up`. If you already have such a clone but don’t want to change it, you can clone that copy with `hg clone /path/to/other/copy`, and then edit `.hg/hgrc` as above and do `hg pull` & `hg up`.

• Now you have a complete copy of the PyPy repo. Make a branch with a command like `hg branch name_of_your_branch`.

Edit

• Edit things. Use `hg diff` to see what you changed. Use `hg add` to make Mercurial aware of new files you added, e.g. new test files. Use `hg status` to see if there are such files. Write and run tests! (See the rest of this page.)

• Commit regularly with `hg commit`. A one-line commit message is fine. We love to have tons of commits; make one as soon as you have some progress, even if it is only some new test that doesn’t pass yet, or fixing things even if not all tests pass. Step by step, you are building the history of your changes, which is the point of a version control system. (There are commands like `hg log` and `hg up` that you should read about later, to learn how to navigate this history.)

• The commits stay on your machine until you do `hg push` to “push” them back to the repo named in the file `.hg/hgrc`. Repos are basically just collections of commits (a commit is also called a changeset): there is one repo per url, plus one for each local copy on each local machine. The commands `hg push` and `hg pull` copy commits around, with the goal that all repos in question end up with the exact same set of commits. By opposition, `hg up` only updates the “working copy” by reading the local repository, i.e. it makes the files that you see correspond to the latest (or any other) commit locally present.

• You should push often; there is no real reason not to. Remember that even if they are pushed, with the setup above, the commits are (1) only in `bitbucket.org/yourname/pypy`, and (2) in the branch you named. Yes, they are publicly visible, but don’t worry about someone walking around the thousands of repos on bitbucket saying “hah, look at the bad coding style of that guy”. Try to get into the mindset that your work is not
secret and it’s fine that way. We might not accept it as is for PyPy, asking you instead to improve some things, but we are not going to judge you.

Pull Request

- The final step is to open a pull request, so that we know that you’d like to merge that branch back to the original pypy/pypy repo. This can also be done several times if you have interesting intermediate states, but if you get there, then we’re likely to proceed to the next stage, which is...

- Get a regular account for pushing directly to bitbucket.org/pypy/pypy (just ask and you’ll get it, basically). Once you have it you can rewrite your file .hg/hgrc to contain default = ssh://hg@bitbucket.org/pypy/pypy. Your changes will then be pushed directly to the official repo, but (if you follow these rules) they are still on a branch, and we can still review the branches you want to merge.

- If you get closer to the regular day-to-day development, you’ll notice that we generally push small changes as one or a few commits directly to the branch default. Also, we often collaborate even if we are on other branches, which do not really “belong” to anyone. At this point you’ll need hg merge and learn how to resolve conflicts that sometimes occur when two people try to push different commits in parallel on the same branch. But it is likely an issue for later :-)

3.1.3 Architecture

PyPy has layers. Just like ogres or onions. Those layers help us keep the respective parts separated enough to be worked on independently and make the complexity manageable. This is, again, just a sanity requirement for such a complex project. For example writing a new optimization for the JIT usually does not involve touching a Python interpreter at all or the JIT assembler backend or the garbage collector. Instead it requires writing small tests in rpython/jit/metainterp/optimizeopt/test/test_* and fixing files there. After that, you can just compile PyPy and things should just work.

Further Reading: architecture

Where to start?

PyPy is made from parts that are relatively independent of each other. You should start looking at the part that attracts you most (all paths are relative to the PyPy top level directory). You may look at our directory reference or start off at one of the following points:

- pypy/interpreter contains the bytecode interpreter: bytecode dispatcher in pypy/interpreter/pyopcode.py, frame and code objects in pypy/interpreter/eval.py and pypy/interpreter/pyframe.py, function objects and argument passing in pypy/interpreter/function.py and pypy/interpreter/argument.py, the object space interface definition in pypy/interpreter/baseobjspace.py, modules in pypy/interpreter/module.py and pypy/interpreter/mixedmodule.py. Core types supporting the bytecode interpreter are defined in pypy/interpreter/typedef.py.

- pypy/interpreter/pypyparser contains a recursive descent parser, and grammar files that allow it to parse the syntax of various Python versions. Once the grammar has been processed, the parser can be translated by the above machinery into efficient code.

- pypy/interpreter/astcompiler contains the compiler. This contains a modified version of the compiler package from CPython that fixes some bugs and is translatable.

- pypy/objspace/std contains the Standard object space. The main file is pypy/objspace/std/objspace.py. For each type, the file xxxobject.py contains the implementation for objects of type xxx, as a first approximation. (Some types have multiple implementations.)
3.1.4 Building

For building PyPy, we recommend installing a pre-built PyPy first (see Downloading and Installing PyPy). It is possible to build PyPy with CPython, but it will take a lot longer to run – depending on your architecture, between two and three times as long.

Further Reading: Build

Coding Guide

As well as the usual pep8 and formatting standards, there are a number of naming conventions and coding styles that are important to understand before browsing the source.

Further Reading: Coding Guide

3.1.5 Testing

Test driven development

Instead, we practice a lot of test driven development. This is partly because of very high quality requirements for compilers and partly because there is simply no other way to get around such complex project, that will keep you sane. There are probably people out there who are smart enough not to need it, we’re not one of those. You may consider familiarizing yourself with pytest, since this is a tool we use for tests. This leads to the next issue:

pytest and the py lib

The pytest testing tool drives all our testing needs.

We use the py library for filesystem path manipulations, terminal writing, logging and some other support functionality.

You don’t necessarily need to install these two libraries because we also ship them inlined in the PyPy source tree.

Running PyPy’s unit tests

PyPy development always was and is still thoroughly test-driven. We use the flexible pytest testing tool which you can install independently and use for other projects.

The PyPy source tree comes with an inlined version of pytest which you can invoke by typing:

```
python pytest.py -h
```

This is usually equivalent to using an installed version:

```
pytest -h
```

If you encounter problems with the installed version make sure you have the correct version installed which you can find out with the `--version` switch.

You will need the build requirements to run tests successfully, since many of them compile little pieces of PyPy and then run the tests inside that minimal interpreter. The cpyext tests also require pycparser, and many tests build cases with hypothesis.

Now on to running some tests. PyPy has many different test directories and you can use shell completion to point at directories or files:
See `py.test` usage and invocations for some more generic info on how you can run tests.

Beware trying to run “all” pypy tests by pointing to the root directory or even the top level subdirectory `pypy`. It takes hours and uses huge amounts of RAM and is not recommended.

To run CPython regression tests you can point to the `lib-python` directory:

```
py.test lib-python/2.7/test/test_datetime.py
```

This will usually take a long time because this will run the PyPy Python interpreter on top of CPython. On the plus side, it’s usually still faster than doing a full translation and running the regression test with the translated PyPy Python interpreter.

### 3.1.6 Testing After Translation

While the usual invocation of `pytest` translates a piece of RPython code and runs it, we have a test extension to run tests without translation, directly on the host python. This is very convenient for modules such as `cpyext` to compare and contrast test results between CPython and PyPy. Untranslated tests are invoked by using the `-A` or `--runappdirect` option to `py.test`:

```
python2 pytest.py -A pypy/module/cpyext/test
```

where `python2` can be either `python2` or `pypy2`. On the `py3` branch, the collection phase must be run with `python2` so untranslated tests are run with:

```
cpython2 pytest.py -A pypy/module/cpyext/test --python=path/to/pypy3
```

### 3.1.7 Tooling & Utilities

If you are interested in the inner workings of the PyPy Python interpreter, there are some features of the untranslated Python interpreter that allow you to introspect its internals.

#### Interpreter-level console

To start interpreting Python with PyPy, install a C compiler that is supported by distutils and use Python 2.7 or greater to run PyPy:

```
cd pypy
python bin/pyinteractive.py
```

After a few seconds (remember: this is running on top of CPython), you should be at the PyPy prompt, which is the same as the Python prompt, but with an extra “>”.

If you press <Ctrl-C> on the console you enter the interpreter-level console, a usual CPython console. You can then access internal objects of PyPy (e.g. the `object space`) and any variables you have created on the PyPy prompt with the prefix `w_`: 
The mechanism works in both directions. If you define a variable with the w_ prefix on the interpreter-level, you will see it on the app-level:

```python
>>> w_a
W_IntObject(123)
```

Note that the prompt of the interpreter-level console is only ‘>>>’ since it runs on CPython level. If you want to return to PyPy, press <Ctrl-D> (under Linux) or <Ctrl-Z>, <Enter> (under Windows).

Also note that not all modules are available by default in this mode (for example: _continuation needed by greenlet), you may need to use one of --withmod-... command line options.

You may be interested in reading more about the distinction between interpreter-level and app-level.

**pyinteractive.py options**

To list the PyPy interpreter command line options, type:

```bash
cd pypy
python bin/pyinteractive.py --help
```

pyinteractive.py supports most of the options that CPython supports too (in addition to a large amount of options that can be used to customize pyinteractive.py). As an example of using PyPy from the command line, you could type:

```bash
python pyinteractive.py --withmod-time -c "from test import pystone; pystone.main(10)"
```

Alternatively, as with regular Python, you can simply give a script name on the command line:

```bash
python pyinteractive.py --withmod-time ../../lib-python/2.7/test/pystone.py 10
```

The --withmod-xxx option enables the built-in module xxx. By default almost none of them are, because initializing them takes time. If you want anyway to enable all built-in modules, you can use --allworkingmodules.

See our configuration sections for details about what all the commandline options do.

**Tracing bytecode and operations on objects**

You can use a simple tracing mode to monitor the interpretation of bytecodes. To enable it, set __pytrace__ = 1 on the interactive PyPy console:

```python
>>> __pytrace__ = 1
Tracing enabled
>>> x = 5
   <module>: LOAD_CONST 0 (5)
```
3.1.8 Demos

The example-interpreter repository contains an example interpreter written using the RPython translation toolchain.

3.1.9 graphviz & pygame for flow graph viewing (highly recommended)

graphviz and pygame are both necessary if you want to look at generated flow graphs:

graphviz: http://www.graphviz.org/Download.php
pygame: http://www.pygame.org/download.shtml

3.2 PyPy’s Configuration Handling

Due to more and more available configuration options it became quite annoying to hand the necessary options to where they are actually used and even more annoying to add new options. To circumvent these problems configuration management was introduced. There all the necessary options are stored in a configuration object, which is available nearly everywhere in the RPython toolchain and in the standard interpreter so that adding new options becomes trivial. Options are organized into a tree. Configuration objects can be created in different ways, there is support for creating an optparse command line parser automatically.

3.2.1 Main Assumption

Configuration objects are produced at the entry points and handed down to where they are actually used. This keeps configuration local but available everywhere and consistent. The configuration values are created using the command line.

3.2.2 API Details

The handling of options is split into two parts: the description of which options are available, what their possible values and defaults are and how they are organized into a tree. A specific choice of options is bundled into a configuration object which has a reference to its option description (and therefore makes sure that the configuration values adhere to the option description). This splitting is remotely similar to the distinction between types and instances in the type systems of the rtyper: the types describe what sort of fields the instances have.

The Options are organized in a tree. Every option has a name, as does every option group. The parts of the full name of the option are separated by dots: e.g. config.translation.thread.
Description of Options

All the constructors take a name and a doc argument as first arguments to give the option or option group a name and to document it. Most constructors take a default argument that specifies the default value of the option. If this argument is not supplied the default value is assumed to be None. Most constructors also take a cmdline argument where you can specify what the command line option should look like (for example cmdline="-v –version"). If cmdline is not specified a default cmdline option is created that uses the name of the option together with its full path. If None is passed in as cmdline then no command line option is created at all.

Some options types can specify requirements to specify that a particular choice for one option works only if a certain choice for another option is used. A requirement is specified using a list of pairs. The first element of the pair gives the path of the option that is required to be set and the second element gives the required value.

OptionDescription

This class is used to group suboptions.

```python
__init__(self, name, doc, children) children is a list of option descriptions (including OptionDescription instances for nested namespaces).
```

ChoiceOption

Represents a choice out of several objects. The option can also have the value None.

```python
__init__(self, name, doc, values, default=None, requires=None, cmdline=DEFAULT) values is a list of values the option can possibly take, requires is a dictionary mapping values to lists of of two-element tuples.
```

BoolOption

Represents a choice between True and False.

```python
__init__(self, name, doc, default=None, requires=None, suggests=None, cmdline=DEFAULT, negation=True) default specifies the default value of the option. requires is a list of two-element tuples describing the requirements when the option is set to true, suggests is a list of the same structure but the options in there are only suggested, not absolutely necessary. The difference is small: if the current option is set to True, both the required and the suggested options are set. The required options cannot be changed later, though. negation specifies whether the negative commandline option should be generated.
```

IntOption

Represents a choice of an integer.

```python
__init__(self, name, doc, default=None, cmdline=DEFAULT)
```

FloatOption

Represents a choice of a floating point number.

```python
__init__(self, name, doc, default=None, cmdline=DEFAULT)
```
StrOption

Represents the choice of a string.

    __init__(self, name, doc, default=None, cmdline=DEFAULT)

Configuration Objects

Config objects hold the chosen values for the options (of the default, if no choice was made). A Config object is
described by an OptionDescription instance. The attributes of the Config objects are the names of the children
of the OptionDescription. Example:

```python
>>> from rpython.config.config import OptionDescription, Config, BoolOption
>>> descr = OptionDescription("options", ",", [
...     BoolOption("bool", ",", default=False)])
>>> config = Config(descr)
>>> config.bool
False
>>> config.bool = True
>>> config.bool
True
```

Description of the (useful) methods on Config:

    __init__(self, descr, **overrides): descr is an instance of OptionDescription
        that describes the configuration object. overrides can be used to set different default values (see
        method override).

    override(self, overrides): override default values. This marks the overridden values as
defaults, which makes it possible to change them (you can usually change values only once).
    overrides is a dictionary of path strings to values.

    set(self, **kwargs): “do what I mean”-interface to option setting. Searches all paths starting
        from that config for matches of the optional arguments and sets the found option if the match is not
        ambiguous.

Production of optparse Parsers

To produce an optparse parser use the function to_optparse. It will create an option parser using callbacks in such
a way that the config object used for creating the parser is updated automatically.

    to_optparse(config, useoptions=None, parser=None): Returns an optparse parser.
        config is the configuration object for which to create the parser. useoptions is a list of options
        for which to create command line options. It can contain full paths to options or also paths to an
        option description plus an additional ".*" to produce command line options for all sub-options of
        that description. If useoptions is None, then all sub-options are turned into cmdline options.
        parser can be an existing parser object, if None is passed in, then a new one is created.

3.2.3 The usage of config objects in PyPy

The two large parts of PyPy, the Python interpreter and the RPython toolchain, have two separate sets of options.
The translation toolchain options can be found on the config attribute of all TranslationContext instances
and are described in rpython/config/translationoption.py. The interpreter options are attached to the object space, also
under the name config and are described in pypy/config/pypyoption.py. Both set of options are documented in the Configuration Options for PyPy section.

3.3 Potential Project List

3.3.1 Getting involved

We are happy to discuss ideas around the PyPy ecosystem. If you are interested in playing with RPython or PyPy, or have a new idea not mentioned here please join us on irc, channel #pypy (freenode). If you are unsure, but still think that you can make a valuable contribution to PyPy, dont hesitate to contact us on #pypy or on our mailing list. Here are some ideas to get you thinking:

• **Optimize PyPy Memory Usage**: Sometimes PyPy consumes more memory than CPython. Two examples: 1) PyPy seems to allocate and keep alive more strings when importing a big Python modules. 2) The base interpreter size (cold VM started from a console) of PyPy is bigger than the one of CPython. The general procedure of this project is: Run both CPython and PyPy of the same Python version and compare the memory usage (using Massif or other tools). If PyPy consumes a lot more memory then find and resolve the issue.

• **VMPProf + memory profiler**: vmpprof is a statistical memory profiler. We want extend it with new features and resolve some current limitations.

• **VMPProf visualisations**: vmpprof shows a flame graph of the statistical profile and some more information about specific call sites. It would be very interesting to experiment with different information (such as memory, or even information generated by our jit compiler).

• **Explicit typing in RPython**: PyPy wants to have better ways to specify the signature and class attribute types in RPython. See more information about this topic below on this page.

• **Virtual Reality (VR) visualisations for vmpprof**: This is a very open topic with lots of freedom to explore data visualisation for profiles. No VR hardware would be needed for this project. Either universities provide such hardware or in any other case we potentially can lend the VR hardware setup.

3.3.2 Simple tasks for newcomers

• Tkinter module missing support for threads: https://bitbucket.org/pypy/pypy/issue/1929/tkinter-broken-for-threaded-python-on-both

• Optimize random: https://bitbucket.org/pypy/pypy/issue/1901/try-using-a-different-implementation-of

• Implement AF_XXX packet types of sockets: https://bitbucket.org/pypy/pypy/issue/1942/support-for-af_xxx-sockets

• Help with documentation. One task would be to document rpython configuration options currently listed only on this site also on the RPython documentation site.

3.3.3 Mid-to-large tasks

Below is a list of projects that are interesting for potential contributors who are seriously interested in the PyPy project. They mostly share common patterns - they’re mid-to-large in size, they’re usually well defined as a standalone projects and they’re not being actively worked on. For small projects that you might want to work on look above or either look at the issue tracker, pop up on #pypy on irc.freenode.net or write to the mailing list. This is simply for the reason that small possible projects tend to change very rapidly.
This list is mostly for having an overview on potential projects. This list is by definition not exhaustive and we’re pleased if people come up with their own improvement ideas. In any case, if you feel like working on some of those projects, or anything else in PyPy, pop up on IRC or write to us on the mailing list.

3.3.4 Explicit typing in RPython

RPython is mostly based around type inference, but there are many cases where specifying types explicitly is useful. We would like to be able to optionally specify the exact types of the arguments to any function. We already have solutions in that space, `@rpython.rlib.objectmodel.enforceargs` and `@rpython.rlib.signature`, but they are inconvenient and limited. For instance, they do not easily allow to express the type “dict with ints as keys and lists of instances of Foo as values”.

Additionally, we would like to be able to specify the types of instance attributes. Unlike the function case, this is likely to require some refactoring of the annotator.

3.3.5 Make bytearray type fast

PyPy’s bytearray type is very inefficient. It would be an interesting task to look into possible optimizations on this. (XXX current status unknown; ask on #pypy for updates on this.)

3.3.6 Implement copy-on-write list slicing

The idea is to have a special implementation of list objects which is used when doing `myslice = mylist[a:b]`: the new list is not constructed immediately, but only when (and if) `myslice` or `mylist` are mutated.

3.3.7 NumPy rebooted

Our cpyext C-API compatibility layer can now run upstream NumPy unmodified. Release PyPy2.7-v6.0 still fails about 10 of the ~6000 test in the NumPy test suite. We need to improve our ctypes structure -> memoryview conversions, and to refactor the way NumPy adds docstrings.

We also are looking for help in how to hijack NumPy dtype conversion and ufunc calls to allow the JIT to make them fast, using our internal _numpypy module.

3.3.8 Improving the jitviewer

Analyzing performance of applications is always tricky. We have various tools, for example a jitviewer that help us analyze performance.

The old tool was partly rewritten and combined with vmprof. The service is hosted at vmprof.com.

The following shows an old image of the jitviewer. The code generated by the PyPy JIT in a hierarchical way:

- at the bottom level, it shows the Python source code of the compiled loops
- for each source code line, it shows the corresponding Python bytecode
- for each opcode, it shows the corresponding JIT operations, which are the ones actually sent to the backend for compiling (such as `i15 = i10 < 2000` in the example)
The jitviewer is a web application based on django and angularjs: if you have great web developing skills and want to help PyPy, this is an ideal task to get started, because it does not require any deep knowledge of the internals. Head over to vmprof-python, vmprof-server and vmprof-integration to find open issues and documentation.

### 3.3.9 Optimized Unicode Representation

CPython 3.3 will use an optimized unicode representation (see PEP 0393) which switches between different ways to represent a unicode string, depending on whether the string fits into ASCII, has only two-byte characters or needs four-byte characters.

The actual details would be rather different in PyPy, but we would like to have the same optimization implemented. Or maybe not. We can also play around with the idea of using a single representation: as a byte string in utf-8. (This idea needs some extra logic for efficient indexing, like a cache.) Work has begun on the unicode-utf and unicode-utf8-py3 branches. More is needed, for instance there are SIMD optimizations that are not yet used.

### 3.3.10 Convert RPython to Python3

The world is moving on, we should too.

### 3.3.11 Improve performance

- Make uninlined Python-level calls faster
- Switch to a sea-of-nodes IR, or a Lua-Jit-like IR which iterates on on the sea-of-nodes approach
- Use real register-allocation
- Improve instruction selection / scheduling
- Create a hybrid tracing/method JIT

### 3.3.12 Improve warmup

- Interpreter speed-ups
- Optimize while tracing
- Cache information between runs
3.3.13 Translation Toolchain

(XXX this is unlikely to be feasible.)
- Incremental or distributed translation.
- Allow separate compilation of extension modules.

3.3.14 Various GCs

PyPy has pluggable garbage collection policy. This means that various garbage collectors can be written for specialized purposes, or even various experiments can be done for the general purpose. Examples:
- A garbage collector that compact memory better for mobile devices
- A concurrent garbage collector (a lot of work)
- A collector that keeps object flags in separate memory pages, to avoid un-sharing all pages between several fork(ed) processes

3.3.15 STM (Software Transactional Memory)

This is work in progress. Besides the main development path, whose goal is to make a (relatively fast) version of pypy which includes STM, there are independent topics that can already be experimented with on the existing, JIT-less pypy-stm version:
- What kind of conflicts do we get in real use cases? And, sometimes, which data structures would be more appropriate? For example, a dict implemented as a hash table will suffer “stm collisions” in all threads whenever one thread writes anything to it; but there could be other implementations. Maybe alternate strategies can be implemented at the level of the Python interpreter (see list/dict strategies, pypy/objspace/std/{list, dict}object.py).
- More generally, there is the idea that we would need some kind of “debugger”-like tool to “debug” things that are not bugs, but stm conflicts. How would this tool look like to the end Python programmers? Like a profiler? Or like a debugger with breakpoints on aborted transactions? It would probably be all app-level, with a few hooks e.g. for transaction conflicts.
- Find good ways to have libraries using internally threads and atomics, but not exposing threads to the user. Right now there is a rough draft in lib_pypy/transaction.py, but much better is possible. For example we could probably have an iterator-like concept that allows each loop iteration to run in parallel.

3.3.16 Introduce new benchmarks

Our benchmark runner is showing its age. We should merge with the CPython site

Additionally, we’re usually happy to introduce new benchmarks. Please consult us before, but in general something that’s real-world python code and is not already represented is welcome. We need at least a standalone script that can run without parameters. Example ideas (benchmarks need to be got from them!):
- hg

3.3.17 Interfacing with C

While we could make cpyext faster, we would also like to explore other ideas. It seems cffi is only appropriate for small to medium-sized extensions, and it is hard to imagine NumPy abandoning the C-API. Here are a few ideas: *
Extend Cython to have a backend that can be understood by the JIT * Collaborate with C-extension authors to ensure full PyPy support (see below) * Put PyPy compatible packages on PyPI and in conda

3.3.18 Support more platforms

We have a plan for a Windows 64 port.

Make more python modules pypy-friendly

A lot of work has gone into PyPy’s implementation of CPython’s C-API, cpyext, over the last years to let it reach a practical level of compatibility, so that C extensions for CPython work on PyPy without major rewrites. However, there are still many edges and corner cases where it misbehaves.

For any popular extension that does not already advertise full PyPy compatibility, it would thus be useful to take a close look at it in order to make it fully compatible with PyPy. The general process is something like:

• Run the extension’s tests on PyPy and look at the test failures.
• Some of the failures may be solved by identifying cases where the extension relies on undocumented or internal details of CPython, and rewriting the relevant code to follow documented best practices. Open issues and send pull requests as appropriate given the extension’s development process.
• Other failures may highlight incompatibilities between cpyext and CPython. Please report them to us and try to fix them.
• Run benchmarks, either provided by the extension developers or created by you. Any case where PyPy is significantly slower than CPython is to be considered a bug and solved as above.

Alternatively, an approach we used to recommend was to rewrite C extensions using more pypy-friendly technologies, e.g. cffi. Here is a partial list of good work that needs to be finished:

wxPython https://bitbucket.org/amauryfa/wxpython-cffi
   Status: A project by a PyPy developer to adapt the Phoenix sip build system to cffi
   The project is a continuation of a 2013 GSOC https://bitbucket.org/waedt/wxpython_cffi
   TODO: Merge the latest version of the wrappers and finish the sip conversion

pygame https://github.com/CTPUG/pygame_cffi
   Status: see blog post <http://morepypy.blogspot.com/2014/03/pygamecffi-pygame-on-pypy.html>
   TODO: see the end of the blog post

pyopengl https://bitbucket.org/duangle/pyopengl-cffi
   Status: unknown

3.4 Project Documentation

architecture gives a complete view of PyPy’s basic design.

coding guide helps you to write code for PyPy (especially also describes coding in RPython a bit).

sprint reports lists reports written at most of our sprints, from 2003 to the present.

papers, talks and related projects lists presentations and related projects as well as our published papers.

PyPy video documentation is a page linking to the videos (e.g. of talks and introductions) that are available.
Technical reports is a page that contains links to the reports that we submitted to the European Union. development methodology describes our sprint-driven approach. LICENSE contains licensing details (basically a straight MIT-license). Glossary of PyPy words to help you align your inner self with the PyPy universe.

3.4.1 Coding Guide

Contents

• Coding Guide
  – Overview and motivation
    • CPython vs. PyPy
    • Application-level and interpreter-level execution and objects
    • Application level is often preferable
    • Our runtime interpreter is “RPython”
  – Wrapping rules
    • Wrapping
    • Naming conventions
    • Operations on w_xxx
    • Application-level exceptions
  – Modules in PyPy
    • Determining the location of a module implementation
    • Module directories / Import order
    • Modifying a CPython library module or regression test
    • Implementing a mixed interpreter/application level Module
    • application level definitions
    • interpreter level definitions
    • Testing modules in lib_pypy/
    • Testing modules in pypy/module
    • Testing modules in lib-python
  – Naming conventions and directory layout
    • Directory and File Naming
    • Naming of python objects
    • Committing & Branching to the repository
  – Using the development bug/feature tracker
  – Testing in PyPy
This document describes coding requirements and conventions for working with the PyPy code base. Please read it carefully and ask back any questions you might have. The document does not talk very much about coding style issues. We mostly follow PEP 8 though. If in doubt, follow the style that is already present in the code base.

### Overview and motivation

We are writing a Python interpreter in Python, using Python’s well known ability to step behind the algorithmic problems as a language. At first glance, one might think this achieves nothing but a better understanding how the interpreter works. This alone would make it worth doing, but we have much larger goals.

### CPython vs. PyPy

Compared to the CPython implementation, Python takes the role of the C Code. We rewrite the CPython interpreter in Python itself. We could also aim at writing a more flexible interpreter at C level but we want to use Python to give an alternative description of the interpreter.

The clear advantage is that such a description is shorter and simpler to read, and many implementation details vanish. The drawback of this approach is that this interpreter will be unacceptably slow as long as it is run on top of CPython.

To get to a useful interpreter again, we need to translate our high-level description of Python to a lower level one. One rather straight-forward way is to do a whole program analysis of the PyPy interpreter and create a C source, again. There are many other ways, but let’s stick with this somewhat canonical approach.

### Application-level and interpreter-level execution and objects

Since Python is used for implementing all of our code base, there is a crucial distinction to be aware of: that between interpreter-level objects and application-level objects. The latter are the ones that you deal with when you write normal python programs. Interpreter-level code, however, cannot invoke operations nor access attributes from application-level objects. You will immediately recognize any interpreter level code in PyPy, because half the variable and object names start with a `w_`, which indicates that they are wrapped application-level values.

Let’s show the difference with a simple example. To sum the contents of two variables `a` and `b`, one would write the simple application-level `a+b` – in contrast, the equivalent interpreter-level code is `space.add(w_a, w_b)`, where `space` is an instance of an object space, and `w_a` and `w_b` are typical names for the wrapped versions of the two variables.

It helps to remember how CPython deals with the same issue: interpreter level code, in CPython, is written in C and thus typical code for the addition is `PyNumber_Add(p_a, p_b)` where `p_a` and `p_b` are C variables of type `PyObject*`. This is conceptually similar to how we write our interpreter-level code in Python.
Moreover, in PyPy we have to make a sharp distinction between interpreter- and application-level exceptions: application exceptions are always contained inside an instance of `OperationError`. This makes it easy to distinguish failures (or bugs) in our interpreter-level code from failures appearing in a python application level program that we are interpreting.

**Application level is often preferable**

Application-level code is substantially higher-level, and therefore correspondingly easier to write and debug. For example, suppose we want to implement the `update` method of dict objects. Programming at application level, we can write an obvious, simple implementation, one that looks like an executable definition of `update`, for example:

```python
def update(self, other):
    for k in other.keys():
        self[k] = other[k]
```

If we had to code only at interpreter level, we would have to code something much lower-level and involved, say something like:

```python
def update(space, w_self, w_other):
    w_keys = space.call_method(w_other, 'keys')
    w_iter = space.iter(w_keys)
    while True:
        try:
            w_key = space.next(w_iter)
        except OperationError as e:
            if not e.match(space, space.w_StopIteration):
                raise  # re-raise other app-level exceptions
            break
        w_value = space.getitem(w_other, w_key)
        space.setitem(w_self, w_key, w_value)
```

This interpreter-level implementation looks much more similar to the C source code. It is still more readable than its C counterpart because it doesn’t contain memory management details and can use Python’s native exception mechanism.

In any case, it should be obvious that the application-level implementation is definitely more readable, more elegant and more maintainable than the interpreter-level one (and indeed, `dict.update` is really implemented at applevel in PyPy).

In fact, in almost all parts of PyPy, you find application level code in the middle of interpreter-level code. Apart from some bootstrapping problems (application level functions need a certain initialization level of the object space before they can be executed), application level code is usually preferable. We have an abstraction (called the ‘Gateway’) which allows the caller of a function to remain ignorant of whether a particular function is implemented at application or interpreter level.

**Our runtime interpreter is “RPython”**

In order to make a C code generator feasible all code on interpreter level has to restrict itself to a subset of the Python language, and we adhere to some rules which make translation to lower level languages feasible. Code on application level can still use the full expressivity of Python.

Unlike source-to-source translations (like e.g. Starkiller or more recently ShedSkin) we start translation from live python code objects which constitute our Python interpreter. When doing its work of interpreting bytecode our Python implementation must behave in a static way often referenced as “RPythonic”.

74 Chapter 3. Development
However, when the PyPy interpreter is started as a Python program, it can use all of the Python language until it reaches a certain point in time, from which on everything that is being executed must be static. That is, during initialization our program is free to use the full dynamism of Python, including dynamic code generation.

An example can be found in the current implementation which is quite elegant: For the definition of all the op-codes of the Python interpreter, the module `dis` is imported and used to initialize our bytecode interpreter. (See `__initclass__` in `pypy/interpreter/pyopcode.py`). This saves us from adding extra modules to PyPy. The import code is run at startup time, and we are allowed to use the CPython builtin import function.

After the startup code is finished, all resulting objects, functions, code blocks etc. must adhere to certain runtime restrictions which we describe further below. Here is some background for why this is so: during translation, a whole program analysis (“type inference”) is performed, which makes use of the restrictions defined in RPython. This enables the code generator to emit efficient machine level replacements for pure integer objects, for instance.

**Wrapping rules**

**Wrapping**

PyPy is made of Python source code at two levels: there is on the one hand *application-level code* that looks like normal Python code, and that implements some functionalities as one would expect from Python code (e.g. one can give a pure Python implementation of some built-in functions like `zip()`). There is also *interpreter-level code* for the functionalities that must more directly manipulate interpreter data and objects (e.g. the main loop of the interpreter, and the various object spaces).

Application-level code doesn’t see object spaces explicitly: it runs using an object space to support the objects it manipulates, but this is implicit. There is no need for particular conventions for application-level code. The sequel is only about interpreter-level code. (Ideally, no application-level variable should be called `space` or `w_xxx` to avoid confusion.)

The `w_` prefixes so lavishly used in the example above indicate, by PyPy coding convention, that we are dealing with *wrapped* (or *boxed*) objects, that is, interpreter-level objects which the object space constructs to implement corresponding application-level objects. Each object space supplies `wrap`, `unwrap`, `int_w`, `interpclass_w`, etc. operations that move between the two levels for objects of simple built-in types; each object space also implements other Python types with suitable interpreter-level classes with some amount of internal structure.

For example, an application-level Python `list` is implemented by the *standard object space* as an instance of `W_ListObject`, which has an instance attribute `wrappeditems` (an interpreter-level list which contains the application-level list’s items as wrapped objects).

The rules are described in more details below.

**Naming conventions**

- `space`: the object space is only visible at interpreter-level code, where it is by convention passed around by the name `space`.
- `w_xxx`: any object seen by application-level code is an object explicitly managed by the object space. From the interpreter-level point of view, this is called a *wrapped* object. The `w_` prefix is used for any type of application-level object.
- `xxx_w`: an interpreter-level container for wrapped objects, for example a list or a dict containing wrapped objects. Not to be confused with a wrapped object that would be a list or a dict: these are normal wrapped objects, so they use the `w_` prefix.
Operations on \texttt{w Xxx}

The core bytecode interpreter considers wrapped objects as black boxes. It is not allowed to inspect them directly. The allowed operations are all implemented on the object space: they are called \texttt{space.xxx()}, where \texttt{xxx} is a standard operation name (\texttt{add, getattr, call, eq...}). They are documented in the \texttt{object space document}.

A short warning: \textbf{don’t do} \texttt{w_x == w_y} or \texttt{w_x is w_y}! rationale for this rule is that there is no reason that two wrappers are related in any way even if they contain what looks like the same object at application-level. To check for equality, use \texttt{space.is_true(space.eq(w_x, w_y))} or even better \texttt{space.eq_w(w_x, w_y)} returning directly a interpreter-level bool. To check for identity, use \texttt{space.is_true(space.is_(w_x, w_y))} or better \texttt{space.is_w(w_x, w_y)}.

Application-level exceptions

Interpreter-level code can use exceptions freely. However, all application-level exceptions are represented as an \texttt{OperationError} at interpreter-level. In other words, all exceptions that are potentially visible at application-level are internally an \texttt{OperationError}. This is the case of all errors reported by the object space operations (\texttt{space.add()} etc.).

To raise an application-level exception:

```python
from pypy.interpreter.error import oefmt
raise oefmt(space.w_XxxError, "message")
raise oefmt(space.w_XxxError, "file \$s not found in \$s", filename, dir)
raise oefmt(space.w_XxxError, "file descriptor \$d not open", fd)
```

To catch a specific application-level exception:

```python
try:
    ...
except OperationError as e:
    if not e.match(space, space.w_XxxError):
        raise
    ...
```

This construct catches all application-level exceptions, so we have to match it against the particular \texttt{w_XxxError} we are interested in and re-raise other exceptions. The exception instance \texttt{e} holds two attributes that you can inspect: \texttt{e.w_type} and \texttt{e.w_value}. Do not use \texttt{e.w_type} to match an exception, as this will miss exceptions that are instances of subclasses.

Modules in PyPy

Modules visible from application programs are imported from interpreter or application level files. PyPy reuses almost all python modules of CPython’s standard library, currently from version 2.7.8. We sometimes need to modify modules and - more often - regression tests because they rely on implementation details of CPython.

If we don’t just modify an original CPython module but need to rewrite it from scratch we put it into \texttt{lib_pypy/} as a pure application level module.

When we need access to interpreter-level objects we put the module into \texttt{pypy/module}. Such modules use a \texttt{mixed module mechanism} which makes it convenient to use both interpreter- and application-level parts for the implementation. Note that there is no extra facility for pure-interpreter level modules, you just write a mixed module and leave the application-level part empty.
Determining the location of a module implementation

You can interactively find out where a module comes from, when running py.py. here are examples for the possible locations:

```python
>>> import sys
>>> sys.__file__
'/home/hpk/pypy-dist/pypy/module/sys'

>>> import cPickle
>>> cPickle.__file__
'/home/hpk/pypy-dist/lib_pypy/cPickle..py'

>>> import os
>>> os.__file__
'/home/hpk/pypy-dist/lib-python/2.7/os.py'
```

Module directories / Import order

Here is the order in which PyPy looks up Python modules:

- `pypy/module`
  - mixed interpreter/app-level builtin modules, such as the `sys` and `__builtin__` module.
- `contents of PYTHONPATH`
  - lookup application level modules in each of the : separated list of directories, specified in the PYTHONPATH environment variable.
- `lib_pypy/`
  - contains pure Python reimplementation of modules.
- `lib-python/2.7/`
  - The modified CPython library.

Modifying a CPython library module or regression test

Although PyPy is very compatible with CPython we sometimes need to change modules contained in our copy of the standard library, often due to the fact that PyPy works with all new-style classes by default and CPython has a number of places where it relies on some classes being old-style.

We just maintain those changes in place, to see what is changed we have a branch called `vendor/stdlib` wich contains the unmodified cpython stdlib

Implementing a mixed interpreter/application level Module

If a module needs to access PyPy’s interpreter level then it is implemented as a mixed module.

Mixed modules are directories in `pypy/module` with an `__init__.py` file containing specifications where each name in a module comes from. Only specified names will be exported to a Mixed Module’s applevel namespace.

Sometimes it is necessary to really write some functions in C (or whatever target language). See rffi details.
**application level definitions**

Application level specifications are found in the `appleveldefs` dictionary found in `__init__.py` files of directories in `pypy/module`. For example, in `pypy/module/__builtin__/__init__.py` you find the following entry specifying where `__builtin__.locals` comes from:

```python
...
'locals' : 'app_inspect.locals',
...
```

The `app_` prefix indicates that the submodule `app_inspect` is interpreted at application level and the wrapped function value for `locals` will be extracted accordingly.

**interpreter level definitions**

Interpreter level specifications are found in the `interpleveldefs` dictionary found in `__init__.py` files of directories in `pypy/module`. For example, in `pypy/module/__builtin__/__init__.py` the following entry specifies where `__builtin__.len` comes from:

```python
...
'len' : 'operation.len',
...
```

The `operation` submodule lives at interpreter level and `len` is expected to be exposable to application level. Here is the definition for `operation.len()`:

```python
def len(space, w_obj):
    "len(object) -> integer
    Return the number of items of a sequence or mapping."
    return space.len(w_obj)
```

Exposed interpreter level functions usually take a `space` argument and some wrapped values (see Wrapping rules). You can also use a convenient shortcut in `interpleveldefs` dictionaries: namely an expression in parentheses to specify an interpreter level expression directly (instead of pulling it indirectly from a file):

```python
...
'None' : '(space.w_None)',
'False' : '(space.w_False)',
...
```

The interpreter level expression has a `space` binding when it is executed.

Adding an entry under `pypy/module` (e.g. `mymodule`) entails automatic creation of a new config option (such as `--withmod-mymodule` and `--withoutmod-mymodule` (the latter being the default)) for `py.py` and `translate.py`.

**Testing modules in `lib_pypy`**

You can go to the `pypy/module/test_lib_pypy/` directory and invoke the testing tool (“py.test” or “python ./pypy/test_all.py”) to run tests against the `lib_pypy` hierarchy. Note, that tests in `pypy/module/test_lib_pypy/` are allowed and encouraged to let their tests run at interpreter level although `lib_pypy` modules eventually live at PyPy’s application level. This allows us to quickly test our python-coded reimplementations against CPython.
Testing modules in `pypy/module`

Simply change to `pypy/module` or to a subdirectory and *run the tests as usual*.

Testing modules in `lib-python`

In order to let CPython’s regression tests run against PyPy you can switch to the `lib-python/` directory and run the testing tool in order to start compliance tests. (XXX check windows compatibility for producing test reports).

Naming conventions and directory layout

Directory and File Naming

- directories/modules/namespaces are always **lowercase**
- never use plural names in directory and file names
- `__init__.py` is usually empty except for `pypy/objspace/*` and `pypy/module/*/__init__.py`.
- don’t use more than 4 directory nesting levels
- keep filenames concise and completion-friendly.

Naming of python objects

- class names are **CamelCase**
- functions/methods are lowercase and _ separated
- object space classes are spelled `XyzObjSpace`. e.g.
  - `StdObjSpace`
  - `FlowObjSpace`
- at interpreter level and in ObjSpace all boxed values have a leading `w_` to indicate “wrapped values”. This includes `w_self`. Don’t use `w_` in application level python only code.

Committing & Branching to the repository

- write good log messages because several people are reading the diffs.
- What was previously called trunk is called the default branch in mercurial. Branches in mercurial are always pushed together with the rest of the repository. To create a try1 branch (assuming that a branch named try1 doesn’t already exists) you should do:

```
hg branch try1
```

The branch will be recorded in the repository only after a commit. To switch back to the default branch:

```
hg update default
```

For further details use the help or refer to the [official wiki](https://pypypy.org):
Using the development bug/feature tracker

We use bitbucket for issues tracking and pull-requests.

Testing in PyPy

Our tests are based on the py.test tool which lets you write unitests without boilerplate. All tests of modules in a directory usually reside in a subdirectory test. There are basically two types of unit tests:

• **Interpreter Level tests.** They run at the same level as PyPy’s interpreter.

• **Application Level tests.** They run at application level which means that they look like straight python code but they are interpreted by PyPy.

**Interpreter level tests**

You can write test functions and methods like this:

```python
def test_something(space):
    # use space ...

class TestSomething(object):
    def test_some(self):
        # use 'self.space' here
```

Note that the prefix test for test functions and Test for test classes is mandatory. In both cases you can import Python modules at module global level and use plain ‘assert’ statements thanks to the usage of the py.test tool.

**Application Level tests**

For testing the conformance and well-behavedness of PyPy it is often sufficient to write “normal” application-level Python code that doesn’t need to be aware of any particular coding style or restrictions. If we have a choice we often use application level tests which usually look like this:

```python
def app_test_something():
    # application level test code

class AppTestSomething(object):
    def test_this(self):
        # application level test code
```

These application level test functions will run on top of PyPy, i.e. they have no access to interpreter details. You cannot use imported modules from global level because they are imported at interpreter-level while you test code runs at application level. If you need to use modules you have to import them within the test function.

Data can be passed into the AppTest using the setup_class method of the AppTest. All wrapped objects that are attached to the class there and start with w_ can be accessed via self (but without the w_) in the actual test method. An example:
class AppTestErrno(object):
    def setup_class(cls):
        cls.w_d = cls.space.wrap({'a': 1, 'b': 2})
    
    def test_dict(self):
        assert self.d['a'] == 1
        assert self.d['b'] == 2

Another possibility is to use cls.space.appexec, for example:

class AppTestSomething(object):
    def setup_class(cls):
        arg = 2
        cls.w_result = cls.space.appexec([cls.space.wrap(arg)], """(arg):
            return arg ** 6
        """)
    
    def test_power(self):
        assert self.result == 2 ** 6

which executes the code string function with the given arguments at app level. Note the use of `w_result` in `setup_class` but `self.result` in the test. Here is how to define an app level class in `setup_class` that can be used in subsequent tests:

class AppTestSet(object):
    def setup_class(cls):
        w_fakeint = cls.space.appexec([], """{
            class FakeInt(object):
                def __init__(self, value):
                    self.value = value
                def __hash__(self):
                    return hash(self.value)
                def __eq__(self, other):
                    if other == self.value:
                        return True
                    return False
            return FakeInt
        }"")
        cls.w_FakeInt = w_fakeint
    
    def test_fakeint(self):
        f1 = self.FakeInt(4)
        assert f1 == 4
        assert hash(f1) == hash(4)

Command line tool test_all

You can run almost all of PyPy’s tests by invoking:

```bash
python test_all.py file_or_directory
```

which is a synonym for the general `py.test` utility located in the `py/bin/` directory. For switches to modify test execution pass the `-h` option.
Coverage reports

In order to get coverage reports the pytest-cov plugin is included. It adds some extra requirements (coverage and cov-core) and can once they are installed coverage testing can be invoked via:

```
python test_all.py --cov file_or_directory_to_cover file_or_directory
```

Test conventions

- adding features requires adding appropriate tests. (It often even makes sense to first write the tests so that you are sure that they actually can fail.)
- All over the pypy source code there are test/ directories which contain unit tests. Such scripts can usually be executed directly or are collectively run by pypy/test_all.py

Changing documentation and website

documentation/website files in your local checkout

Most of the PyPy’s documentation is kept in pypy/doc. You can simply edit or add ‘.rst’ files which contain ReST-markuped files. Here is a ReST quickstart but you can also just look at the existing documentation and see how things work.

Note that the web site of http://pypy.org/ is maintained separately. For now it is in the repository https://bitbucket.org/pypy/pypy.org

Automatically test documentation/website changes

We automatically check referential integrity and ReST-conformance. In order to run the tests you need sphinx installed. Then go to the local checkout of the documentation directory and run the Makefile:

```
cd pypy/doc
make html
```

If you see no failures chances are high that your modifications at least don’t produce ReST-errors or wrong local references. Now you will have .html files in the documentation directory which you can point your browser to!

Additionally, if you also want to check for remote references inside the documentation issue:

```
make linkcheck
```

which will check that remote URLs are reachable.

3.4.2 Sprint reports from PyPy sprints 2003-2010

Here are links to sprint reports from various sprints in the PyPy project, an informal summary of features being worked on and the results of that work. A good summary of the progress over the years….enjoy!

- Hildesheim (Feb 2003)
- Gothenburg (May 2003)
- LovainLaNeuve (June 2003)
- Berlin (Sept 2003)
- Amsterdam (Dec 2003)
- Europython/Gothenburg (June 2004)
- Vilnius (Nov 2004)
- Leysin (Jan 2005)
- PyCon/Washington (March 2005)
- Europython/Gothenburg (June 2005)
- Hildesheim (July 2005)
- Heidelberg (Aug 2005)
- Paris (Oct 2005)
- Gothenburg (Dec 2005)
- Mallorca (Jan 2006)
- PyCon/Dallas (Feb 2006)
- LouvainLaNeuve (March 2006)
- Leysin (April 2006)
- Tokyo (April 2006)
- Düsseldorf (June 2006)
- Europython/Geneva (July 2006)
- Limerick (Aug 2006)
- Düsseldorf (October 2006)
- Leysin (January 2007)
- Hildesheim (Feb 2007) (also EU report writing sprint)
- Göteborg (November 2007)
- Leysin (January 2008)
- Berlin (May 2008)
- Vilnius after EuroPython (July 2008)
- Düsseldorf (August 2008)
- Wroclaw (February 2009)
- Leysin (April 2009)
- Göteborg (August 2009)
- Düsseldorf (November 2009)
- CERN (July 2010)
- Düsseldorf (October 2010)
3.4.3 Papers, talks and related projects

Papers

*Articles about PyPy published so far, most recent first:* (bibtex file)

- A Way Forward in Parallelising Dynamic Languages, R. Meier, A. Rigo
- Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages, C.F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, A. Rigo
- Towards a Jitting VM for Prolog Execution, C.F. Bolz, M. Leuschel, D. Schneider
- High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages, A. Cuni, Ph.D. thesis
- Tracing the Meta-Level: PyPy’s Tracing JIT Compiler, C.F. Bolz, A. Cuni, M. Fijalkowski, A. Rigo
- Faster than C#: Efficient Implementation of Dynamic Languages on .NET, A. Cuni, D. Ancona and A. Rigo
- Automatic JIT Compiler Generation with Runtime Partial Evaluation (Master Thesis), C.F. Bolz
- RPython: A Step towards Reconciling Dynamically and Statically Typed OO Languages, D. Ancona, M. Ancona, A. Cuni and N.D. Matsakis
- How to *not* write Virtual Machines for Dynamic Languages, C.F. Bolz and A. Rigo
- PyPy’s approach to virtual machine construction, A. Rigo and S. Pedroni

*Non-published articles (only submitted so far, or technical reports):*

- Automatic generation of JIT compilers for dynamic languages in .NET, D. Ancona, C.F. Bolz, A. Cuni and A. Rigo
- EU Reports: a list of all the reports we produced until 2007 for the European Union sponsored part of PyPy. Notably, it includes:
  - Core Object Optimization Results, PyPy Team
  - Compiling Dynamic Language Implementations, PyPy Team

*Other research using PyPy (as far as we know it):*

- Hardware Transactional Memory Support for Lightweight Dynamic Language Evolution, N. Riley and C. Zilles
- PyGirl: Generating Whole-System VMs from High-Level Prototypes using PyPy, C. Bruni and T. Verwaest
- Back to the Future in One Week – Implementing a Smalltalk VM in PyPy, C.F. Bolz, A. Kuhn, A. Lienhard, N. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo and T. Verwaest

*Previous work:*

- Representation-Based Just-in-Time Specialization and the Psyco Prototype for Python, A. Rigo

Talks and Presentations

*This part is no longer updated.* The complete list is here (in alphabetical order).
Talks in 2010

- PyCon 2010.

Talks in 2009

- RuPy 2009.
- EuroPython talks 2009.
- PyCon talks 2009.
- Wroclaw (Poland) presentation by Maciej Fijalkowski. Introduction, including about the current JIT.
- PyPy talk at OpenBossa 09 (blog post).

Talks in 2008

- Talk at PyCon Poland 08. In Polish.
- Back to the Future in One Week – Implementing a Smalltalk VM in PyPy by C.F. Bolz et al.; pdf of the presentation at S3 2008.
- PyPy at the Maemo summit.
- PyCon UK 2008 - JIT and PyCon UK 2008 - Status.
- PyCon Italy 2008.
- Talk by Maciej Fijalkowski at SFI 08, Cracow (Poland) Academic IT Festival.
- RuPy 2008.
- PyCon 2008.

Talks in 2007

- Our “road show” tour of the United States: presentations at IBM and at Google.
- Talks at Bern (Switzerland) 2007.
- PyCon UK 2007.
- A presentation in Dresden by Maciej Fijalkowski.
- Multiple talks at EuroPython 2007.
- A presentation at Bad Honnef 2007 by C.F. Bolz about the Prolog interpreter.
- A Dzug talk by Holger Krekel.
- Multiple talks at PyCon 2007.
• A talk at PyCon - Uno 2007.
• RuPy 2007.
• Warsaw 2007.

Talks in 2006

• Warsaw 2006.
• Tokyo 2006.
• PyPy’s VM Approach talk, given by Armin Rigo at the Dynamic Languages Symposium at OOPSLA’06 (Portland OR), and by Samuele Pedroni at Intel Hillsboro (OR) (October). The talk presents the paper PyPy’s approach to virtual machine construction accepted for the symposium.
• PyPy Status talk, given by Samuele Pedroni at the Vancouner Python Workshop 2006 (August).
• Sprint Driven Development, Agile Methodologies in a Distributed Open Source Project (PyPy) talk, by Bea During at XP 2006 (experience report).
• Kill -1: process refactoring in the PyPy project talk, by Bea During at the Agile track/Europython 2006.
• What can PyPy do for you, by Armin Rigo and Carl Friedrich Bolz given at EuroPython 2006. The talk describes practical usecases of PyPy.
• PyPy 3000, a purely implementation-centered lightning talk at EuroPython 2006, given by Armin Rigo and Holger Krekel.
• PyPy introduction at EuroPython 2006, given by Michael Hudson, also stating the status of the project.
• Very similar to the EuroPython intro talk (but somewhat older) is the PyPy intro talk, given by Michael Hudson at ACCU 2006 (April)
• PyPy development method talk, given by Bea During and Holger Krekel at Pycon2006

Talks in 2005

• PyPy - the new Python implementation on the block, given by Carl Friedrich Bolz and Holger Krekel at the 22nd Chaos Communication Conference in Berlin, Dec. 2005.
• Open Source, EU-Funding and Agile Methods, given by Holger Krekel and Bea During at the 22nd Chaos Communication Conference in Berlin, Dec. 2005
• Sprinting the PyPy way, an overview about our sprint methodology, given by Bea During during EuroPython 2005. (More PyPy talks were given, but are not present in detail.)
• PyCon 2005 animated slices, mostly reporting on the translator status.
• py lib slides from the py lib talk at PyCon 2005 (py is used as a support/testing library for PyPy).

Talks in 2004

• EU funding for FOSS talk on Chaos Communication Conference in Berlin, Dec 2004.
Talks in 2003

- oscon2003-paper an early paper presented at Oscon 2003 describing what the PyPy project is about and why you should care.
- Architecture introduction slides a mostly up-to-date introduction for the Amsterdam PyPy-Sprint Dec 2003.

Related projects

- TraceMonkey is using a tracing JIT, similar to the tracing JITs generated by our (in-progress) JIT generator.
- Dynamo showcased transparent dynamic optimization generating an optimized version of a binary program at runtime.
- Tailoring Dynamo to interpreter implementations and challenges - Gregory Sullivan et. al., Dynamic Native Optimization of Native Interpreters. IVME 03. 2003.
- Stackless is a recursion-free version of Python.
- Psyco is a just-in-time specialist for Python.
- JikesRVM a research dynamic optimizing Java VM written in Java.
- Squeak is a Smalltalk-80 implementation written in Smalltalk, being used in Croquet, an experimental distributed multi-user multi-programmer virtual world.
- LLVM the low level virtual machine project.
- CLR under the hood (powerpoint, works with open office) gives a good introduction to the underlying models of Microsoft's Common Language Runtime, the Intermediate Language, JIT and GC issues.
- spyweb translates Python programs to Scheme. (site unavailable)
- Jython is a Python implementation in Java.
- Tunes is not entirely unrelated. The web site changed a lot, but a snapshot of the old Tunes Wiki is available; browsing through it is a lot of fun.

3.4.4 More sprints

The PyPy project is a worldwide collaborative effort and its members are organizing sprints and presenting results at conferences all year round. This page is no longer maintained! See our blog for upcoming events.

EuroPython PyPy sprint 6-9 July 2006

Once again a PyPy sprint took place right after the EuroPython Conference from the 6th to the 9th of July.
Read more in the EuroPython 2006 sprint report.

PyPy at XP 2006 and Agile 2006

PyPy presented experience reports at the two main agile conferences this year, XP 2006 and Agile 2006. Both experience reports focus on aspects of the sprint-driven development method that is being used in PyPy.
Duesseldorf PyPy sprint 2-9 June 2006

The next PyPy sprint will be held in the Computer Science department of Heinrich-Heine Universitaet Duesseldorf from the 2nd to the 9th of June. Main focus of the sprint will be on the goals of the upcoming June 0.9 release.

Read more about the sprint

PyPy sprint at Akihabara (Tokyo, Japan)

April 23rd - 29th 2006. This sprint was in Akihabara, Tokyo, Japan, our hosts was FSII (Free Software Initiative of Japan) and we aimed for the sprint to promote Python and introduce people to PyPy. Good progress was also made on PyPy’s ootypesystem for the more high level backends. For more details, read the last sprint status page and enjoy the pictures.

PyPy at Python UK/ACCU Conference (United Kingdom)

April 19th - April 22nd 2006. Several talks about PyPy were held at this year’s Python UK/ACCU conference. Read more at the ACCU site.

PyPy at XPDay France 2006 in Paris March 23rd - March 24th 2006

Logilab presented PyPy at the first french XP Day that it was sponsoring and which was held in Paris. There was over a hundred attendants. Interesting talks included Python as an agile language and Tools for continuous integration.

Logic Sprint at Louvain-la-Neuve University (Louvain-la-Neuve, Belgium)

March 6th - March 10th 2006. PyPy developers focusing on adding logic programming to PyPy will met with the team that developed the Oz programming language and the Mozart interpreter.

Read the report and the original announcement.

PyCon Sprint 2006 (Dallas, Texas, USA)


A report is coming up.

Talks at PyCon 2006 (Dallas, Texas, USA)


PyPy at Solutions Linux in Paris January 31st - February 2nd 2006

PyPy developers from Logilab presented the intermediate results of the project during the Solutions Linux tradeshow in Paris. A lot of enthusiasts already knew about the project and were eager to learn about the details. Many people discovered PyPy on this occasion and said they were interested in the outcome and would keep an eye on its progress. Read the talk slides.
PyPy Sprint in Palma De Mallorca 23rd - 29th January 2006

The Mallorca sprint that took place in Palma de Mallorca is over. Topics included progressing with the JIT work started in Göteborg and Paris, GC and optimization work, stackless, and improving our way to write glue code for C libraries.

Read more in the announcement, there is a sprint report for the first three days and one for the rest of the sprint.

Preliminary EU reports released

After many hours of writing and typo-hunting we finally finished the reports for the EU. They contain most of the material found on our regular documentation page but also a lot of new material not covered there. Note that all these documents are not approved by the European Union and therefore only preliminary. (01/06/2006)

PyPy Sprint in Göteborg 7th - 11th December 2005

The Gothenburg sprint is over. It was a very productive sprint: work has been started on a JIT prototype, we added support for __del__ in PyPy, the socket module had some progress, PyPy got faster and work was started to expose the internals of our parser and bytecode compiler to the user. Michael and Carl have written a report about the first half and one about the second half of the sprint. (12/18/2005)

PyPy release 0.8.0

The third PyPy release is out, with an integrated and translatable compiler, speed progress, and now the possibility to translate our experimental “Thunk” object space (supporting lazy computed objects) with its features preserved. See the release 0.8 announcement for further details about the release and the getting started section for instructions about downloading it and trying it out. There is also a short FAQ. (11/03/2005)

PyPy Sprint in Paris 10th-16th October 2005

The Paris sprint is over. We are all at home again and more or less exhausted. The sprint attracted 18 participants and took place in Logilab offices in Paris. We were happy to have five new developers to the PyPy Community! The focus was on implementing continuation-passing style (stackless), making the translation process work for target languages with more powerful object systems and some tiny steps into the JIT direction. Michael and Carl have written a report about day one and one about day two and three. Together with Armin they wrote one about the rest of the sprint on the way back. (10/18/2005)

PyPy release 0.7.0

The first implementation of Python in Python is now also the second implementation of Python in C :-)

See the release announcement for further details about the release and the getting started section for instructions about downloading it and trying it out. We also have the beginning of a FAQ. (08/28/2005)

PyPy Sprint in Heidelberg 22nd-29th August 2005

The last PyPy sprint took place at the Heidelberg University in Germany from 22nd August to 29th August (both days included). Its main focus is translation of the whole PyPy interpreter to a low level language and reaching 2.4.1 Python compliance. The goal of the sprint is to release a first self-contained PyPy-0.7 version. Carl has written a report about day 1 - 3 and a heidelberg summary report detailing some of the works that led to the successful release of pypy-0.7.0! 

3.4. Project Documentation 89
PyPy Hildesheim2 finished: first self-contained PyPy run!

Up until 31st August we were in a PyPy sprint at Trillke-Gut. Carl has written a report about day 1, Holger about day 2 and day 3 and Carl again about day 4 and day 5. On day 6 Holger reports the breakthrough: PyPy runs on its own! Hurray! And Carl finally reports about the winding down of day 7 which saw us relaxing, discussing and generally having a good time.

EuroPython 2005 sprints finished

We had two sprints around EuroPython, one more internal core developer one and a public one. Both sprints were quite successful. Regarding the Pre-EuroPython sprint Michael Hudson has posted summaries of day 1, day 2 and day 3 on the pypy-dev mailing list. The larger public sprint has not been summarized yet but it went very well. We had 20 people initially attending to hear the tutorials and work a bit. Later with around 13-14 people we made the move to Python-2.4.1, integrated the parser, improved the LLVM backends and type inference in general. (07/13/2005)

3.4.5 PyPy video documentation

Copyrights and Licensing

The following videos are copyrighted by merlinux gmbh and available on YouTube.

If you need another license, don’t hesitate to contact us.

Trailer: PyPy at the PyCon 2006

This trailer shows the PyPy team at the PyCon 2006, a behind-the-scenes at sprints, talks and everywhere else.

Interview with Tim Peters

Interview with CPython core developer Tim Peters at PyCon 2006, Dallas, US. (2006-03-02)

Tim Peters, a longtime CPython core developer talks about how he got into Python, what he thinks about the PyPy project and why he thinks it would have never been possible in the US.

Interview with Bob Ippolito

What do you think about PyPy? Interview with American software developer Bob Ippolito at PyCon 2006, Dallas, US. (2006-03-01)

Bob Ippolito is an Open Source software developer from San Francisco and has been to two PyPy sprints. In this interview he is giving his opinion on the project.

Introductory talk on PyPy

This introductory talk is given by core developers Michael Hudson and Christian Tismer at PyCon 2006, Dallas, US. (2006-02-26)

Michael Hudson talks about the basic building blocks of Python, the currently available back-ends, and the status of PyPy in general. Christian Tismer takes over to explain how co-routines can be used to implement things like Stackless and Greenlets in PyPy.
Talk on Agile Open Source Methods in the PyPy project

Core developer Holger Krekel and project manager Beatrice During are giving a talk on the agile open source methods used in the PyPy project at PyCon 2006, Dallas, US. (2006-02-26)

Holger Krekel explains more about the goals and history of PyPy, and the structure and organization behind it. Bea During describes the intricacies of driving a distributed community in an agile way, and how to combine that with the formalities required for EU funding.

PyPy Architecture session

This architecture session is given by core developers Holger Krekel and Armin Rigo at PyCon 2006, Dallas, US. (2006-02-26)

Holger Krekel and Armin Rigo talk about the basic implementation, implementation level aspects and the RPython translation toolchain. This talk also gives an insight into how a developer works with these tools on a daily basis, and pays special attention to flow graphs.

Sprint tutorial

Sprint tutorial by core developer Michael Hudson at PyCon 2006, Dallas, US. (2006-02-27)

Michael Hudson gives an in-depth, very technical introduction to a PyPy sprint. The film provides a detailed and hands-on overview about the architecture of PyPy, especially the RPython translation toolchain.

Scripting .NET with IronPython by Jim Hugunin

Talk by Jim Hugunin (Microsoft) on the IronPython implementation on the .NET framework at the PyCon 2006, Dallas, US.

Jim Hugunin talks about regression tests, the code generation and the object layout, the new-style instance and gives a CLS interop demo.

Bram Cohen, founder and developer of BitTorrent

Bram Cohen is interviewed by Steve Holden at the PyCon 2006, Dallas, US.

Keynote speech by Guido van Rossum on the new Python 2.5 features

Guido van Rossum explains the new Python 2.5 features at the PyCon 2006, Dallas, US.

Trailer: PyPy sprint at the University of Palma de Mallorca

This trailer shows the PyPy team at the sprint in Mallorca, a behind-the-scenes of a typical PyPy coding sprint and talk as well as everything else.

Coding discussion of core developers Armin Rigo and Samuele Pedroni

Coding discussion between Armin Rigo and Samuele Pedroni during the PyPy sprint at the University of Palma de Mallorca, Spain. 27.1.2006
PyPy technical talk at the University of Palma de Mallorca

Technical talk on the PyPy project at the University of Palma de Mallorca, Spain. 27.1.2006

Core developers Armin Rigo, Samuele Pedroni and Carl Friedrich Bolz are giving an overview of the PyPy architecture, the standard interpreter, the RPython translation toolchain and the just-in-time compiler.

Warning: Some of these reports are interesting for historical reasons only.

3.4.6 PyPy - Overview over the EU-reports

Below reports summarize and discuss research and development results of the PyPy project during the EU funding period (Dec 2004 - March 2007). They also are very good documentation if you’d like to know in more detail about motivation and implementation of the various parts and aspects of PyPy. Feel free to send questions or comments to pypy-dev, the development list.

Reports of 2007

The PyPy EU Final Activity Report summarizes the 28 month EU project period (Dec 2004-March 2007) on technical, scientific and community levels. You do not need prior knowledge about PyPy but some technical knowledge about computer language implementations is helpful. The report contains reflections and recommendations which might be interesting for other project aiming at funded Open Source research. (2007-05-11)

D09.1 Constraint Solving and Semantic Web is a report about PyPy’s logic programming and constraint solving features, as well as the work going on to tie semantic web technologies and PyPy together. (2007-05-11)

D14.4 PyPy-1.0 Milestone report (for language developers and researchers) summarizes research & technical results of the PyPy-1.0 release and discusses related development process and community aspects. (2007-05-01)

D08.2 JIT Compiler Architecture is a report about the Architecture and working of our JIT compiler generator. (2007-05-01)

D08.1 JIT Compiler Release reports on our successfully including a JIT compiler for Python and the novel framework we used to automatically generate it in PyPy 1.0. (2007-04-30)

D06.1 Core Object Optimization Results documents the optimizations we implemented in the interpreter and object space: dictionary implementations, method call optimizations, etc. The report is still not final so we are very interested in any feedback. (2007-04-04)

D14.5 Documentation of the development process documents PyPy’s sprint-driven development process and puts it into the context of agile methodologies. (2007-03-30)

D13.1 Integration and Configuration is a report about our build and configuration toolchain as well as the planned Debian packages. It also describes the work done to integrate the results of other workpackages into the rest of the project. (2007-03-30)

D02.2 Release Scheme lists PyPy’s six public releases and explains the release structure, tools, directories and policies for performing PyPy releases. (2007-03-30)

D01.2-4 Project Organization is a report about the management activities within the PyPy project and PyPy development process. (2007-03-28)

D11.1 PyPy for Embedded Devices is a report about the possibilities of using PyPy technology for programming embedded devices. (2007-03-26)

D02.3 Testing Tool is a report about the py.test testing tool which is part of the py-lib. (2007-03-23)
D10.1 Aspect-Oriented, Design-by-Contract Programming and RPython static checking is a report about the aop module providing an Aspect Oriented Programming mechanism for PyPy, and how this can be leveraged to implement a Design-by-Contract module. It also introduces RPylint static type checker for RPython code. (2007-03-22)

D12.1 High-Level-Backends and Feature Prototypes is a report about our high-level backends and our several validation prototypes: an information flow security prototype, a distribution prototype and a persistence proof-of-concept. (2007-03-22)

D14.2 Tutorials and Guide Through the PyPy Source Code is a report about the steps we have taken to make the project approachable for newcomers. (2007-03-22)

D02.1 Development Tools and Website is a report about the codespeak development environment and additional tool support for the PyPy development process. (2007-03-21)

D03.1 Extension Compiler is a report about PyPy’s extension compiler and RCTypes, as well as the effort to keep up with CPython’s changes. (2007-03-21)

D07.1 Massive Parallelism and Translation Aspects is a report about PyPy’s optimization efforts, garbage collectors and massive parallelism (stackless) features. This report refers to the paper PyPy’s approach to virtual machine construction. Extends the content previously available in the document “Memory management and threading models as translation aspects – solutions and challenges”. (2007-02-28)

Reports of 2006

D14.3 Report about Milestone/Phase 2 is the final report about the second phase of the EU project, summarizing and detailing technical, research, dissemination and community aspects. Feedback is very welcome!

Reports of 2005

D04.1 Partial Python Implementation contains details about the 0.6 release. All the content can be found in the regular documentation section.

D04.2 Complete Python Implementation contains details about the 0.7 release. All the content can be found in the regular documentation section.

D04.3 Parser and Bytecode Compiler describes our parser and bytecode compiler.

D04.4 PyPy as a Research Tool contains details about the 0.8 release. All the content can be found in the regular documentation section.

D05.1 Compiling Dynamic Language Implementations is a paper that describes the translation process, especially the flow object space and the annotator in detail.

D05.2 A Compiled Version of PyPy contains more details about the 0.7 release. All the content can be found in the regular documentation section.

D05.3 Implementation with Translation Aspects describes how our approach hides away a lot of low level details.

D05.4 Encapsulating Low Level Aspects describes how we weave different properties into our interpreter during the translation process.

D14.1 Report about Milestone/Phase 1 describes what happened in the PyPy project during the first year of EU funding (December 2004 - December 2005)

3.4.7 Old discussion notes needing categorization

The following are old discussion notes which may or may not reflect the current reality.
Help from domain experts would be welcome, since some of these documents probably ought to be moved to a more prominent location, some should be deleted, and some left here.

**Ordering finalizers in the MiniMark GC**

**RPython interface**

In RPython programs like PyPy, we need a fine-grained method of controlling the RPython- as well as the app-level __del__(). To make it possible, the RPython interface is now the following one (from May 2016):

- RPython objects can have __del__() methods. These are called immediately by the GC when the last reference to the object goes away, like in CPython. However, the long-term goal is that all __del__() methods should only contain simple enough code. If they do, we call them “destructors”. They can’t use operations that would resurrect the object, for example. Use the decorator @rgc.must_be_light_finalizer to ensure they are destructors.

- RPython-level __del__() that are not passing the destructor test are supported for backward compatibility, but deprecated. The rest of this document assumes that __del__() are all destructors.

- For any more advanced usage — in particular for any app-level object with a __del__ — we don’t use the RPython-level __del__() method. Instead we use rgc.FinalizerController.register_finalizer(). This allows us to attach a finalizer method to the object, giving more control over the ordering than just an RPython __del__().

We try to consistently call __del__() a destructor, to distinguish it from a finalizer. A finalizer runs earlier, and in topological order; care must be taken that the object might still be reachable at this point if we’re clever enough. A destructor on the other hand runs last; nothing can be done with the object any more, and the GC frees it immediately.

**Destructors**

A destructor is an RPython __del__() method that is called directly by the GC when it is about to free the memory. Intended for objects that just need to free an extra block of raw memory.

There are restrictions on the kind of code you can put in __del__(), including all other functions called by it. These restrictions are checked. In particular you cannot access fields containing GC objects. Right now you can’t call any external C function either.

Destructors are called precisely when the GC frees the memory of the object. As long as the object exists (even in some finalizer queue or anywhere), its destructor is not called.

**Register_finalizer**

The interface for full finalizers is made with PyPy in mind, but should be generally useful.

The idea is that you subclass the rgc.FinalizerQueue class:

- You must give a class-level attribute base_class, which is the base class of all instances with a finalizer. (If you need finalizers on several unrelated classes, you need several unrelated FinalizerQueue subclasses.)

- You override the finalizer_trigger() method; see below.

Then you create one global (or space-specific) instance of this subclass; call it fin. At runtime, you call fin.register_finalizer(obj) for every instance obj that needs a finalizer. Each obj must be an instance of fin.base_class, but not every such instance needs to have a finalizer registered; typically we try to register a finalizer on as few objects as possible (e.g. only if it is an object which has an app-level __del__() method).
After a major collection, the GC finds all objects `obj` on which a finalizer was registered and which are unreachable, and mark them as reachable again, as well as all objects they depend on. It then picks a topological ordering (breaking cycles randomly, if any) and enqueues the objects and their registered finalizer functions in that order, in a queue specific to the prebuilt `fin` instance. Finally, when the major collection is done, it calls `fin.finalizer_trigger()`.

This method `finalizer_trigger()` can either do some work directly, or delay it to be done later (e.g. between two bytecodes). If it does work directly, note that it cannot (directly or indirectly) cause the GIL to be released.

To find the queued items, call `fin.next_dead()` repeatedly. It returns the next queued item, or `None` when the queue is empty.

In theory, it would kind of work if you cumulate several different `FinalizerQueue` instances for objects of the same class, and (always in theory) the same `obj` could be registered several times in the same queue, or in several queues. This is not tested though. For now the untranslated emulation does not support registering the same object several times.

Note that the Boehm garbage collector, used in `rpython -O0`, completely ignores `register_finalizer()`.

**Ordering of finalizers**

After a collection, the MiniMark GC should call the finalizers on some of the objects that have one and that have become unreachable. Basically, if there is a reference chain from an object `a` to an object `b` then it should not call the finalizer for `b` immediately, but just keep `b` alive and try again to call its finalizer after the next collection.

(Note that this creates rare but annoying issues as soon as the program creates chains of objects with finalizers more quickly than the rate at which major collections go (which is very slow). In August 2013 we tried instead to call all finalizers of all objects found unreachable at a major collection. That branch, `gc-del`, was never merged. It is still unclear what the real consequences would be on programs in the wild.)

The basic idea fails in the presence of cycles. It’s not a good idea to keep the objects alive forever or to never call any of the finalizers. The model we came up with is that in this case, we could just call the finalizer of one of the objects in the cycle – but only, of course, if there are no other objects outside the cycle that has a finalizer and a reference to the cycle.

More precisely, given the graph of references between objects:

```python
for each strongly connected component C of the graph:
    if C has at least one object with a finalizer:
        if there is no object outside C which has a finalizer and indirectly references the objects in C:
            mark one of the objects of C that has a finalizer
            copy C and all objects it references to the new space
    for each marked object:
        detach the finalizer (so that it's not called more than once)
        call the finalizer
```

**Algorithm**

During `deal_with_objects_with_finalizers()`, each object `x` can be in 4 possible states:

- `state[x] == 0`: unreachable
- `state[x] == 1`: (temporary state, see below)
- `state[x] == 2`: reachable from any finalizer
- `state[x] == 3`: alive
Initially, objects are in state 0 or 3 depending on whether they have been copied or not by the regular sweep done just before. The invariant is that if there is a reference from \(x\) to \(y\), then \(\text{state}[y] \geq \text{state}[x]\).

The state 2 is used for objects that are reachable from a finalizer but that may be in the same strongly connected component than the finalizer. The state of these objects goes to 3 when we prove that they can be reached from a finalizer which is definitely not in the same strongly connected component. Finalizers on objects with state 3 must not be called.

Let \(\text{closure}(x)\) be the list of objects reachable from \(x\), including \(x\) itself. Pseudo-code (high-level) to get the list of marked objects:

```python
marked = []
for x in objects_with_finalizers:
    if state[x] != 0:
        continue
    marked.append(x)
    for y in closure(x):
        if state[y] == 0:
            state[y] = 2
        elif state[y] == 2:
            state[y] = 3
for x in marked:
    assert state[x] >= 2
    if state[x] != 2:
        marked.remove(x)
```

This does the right thing independently on the order in which the \(\text{objects\_with\_finalizers}\) are enumerated. First assume that \([x_1, \ldots, x_n]\) are all in the same unreachable strongly connected component; no object with finalizer references this strongly connected component from outside. Then:

- when \(x_1\) is processed, \(\text{state}[x_1] = \ldots = \text{state}[x_n] = 0\) independently of whatever else we did before. So \(x_1\) gets marked and we set \(\text{state}[x_1] = \ldots = \text{state}[x_n] = 2\).
- when \(x_2, \ldots, x_n\) are processed, their state is \(!= 0\) so we do nothing.
- in the final loop, only \(x_1\) is marked and \(\text{state}[x_1] = 2\) so it stays marked.

Now, let’s assume that \(x_1\) and \(x_2\) are not in the same strongly connected component and there is a reference path from \(x_1\) to \(x_2\). Then:

- if \(x_1\) is enumerated before \(x_2\), then \(x_2\) is in \(\text{closure}(x_1)\) and so its state gets at least \(\geq 2\) when we process \(x_1\). When we process \(x_2\) later we just skip it (“continue” line) and so it doesn’t get marked.
- if \(x_2\) is enumerated before \(x_1\), then when we process \(x_2\) we mark it and set its state to \(\geq 2\) (before \(x_2\) is in \(\text{closure}(x_2)\)), and then when we process \(x_1\) we set \(\text{state}[x_2] = 2\). So in the final loop \(x_2\) gets removed from the “marked” list.

I think that it proves that the algorithm is doing what we want.

The next step is to remove the use of \(\text{closure()}\) in the algorithm in such a way that the new algorithm has a reasonable performance – linear in the number of objects whose state it manipulates:

```python
marked = []
for x in objects_with_finalizers:
    if state[x] != 0:
        continue
    marked.append(x)
    recursing on the objects \(y\) starting from \(x\):
    if state[y] == 0:
        state[y] = 1
```

(continues on next page)
follow y's children recursively

```python
elif state[y] == 2:
    state[y] = 3
    follow y's children recursively
else:
    don't need to recurse inside y

recursing on the objects y starting from x:
if state[y] == 1:
    state[y] = 2
    follow y's children recursively
else:
    don't need to recurse inside y
```

for x in marked:
    assert state[x] >= 2
    if state[x] != 2:
        marked.remove(x)

In this algorithm we follow the children of each object at most 3 times, when the state of the object changes from 0 to 1 to 2 to 3. In a visit that doesn’t change the state of an object, we don’t follow its children recursively.

In practice, in the MiniMark GCs, we can encode the 4 states with a combination of two bits in the header:

<table>
<thead>
<tr>
<th>state</th>
<th>GCFLAG_VISITED</th>
<th>GCFLAG_FINALIZATION_ORDERING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

So the loop above that does the transition from state 1 to state 2 is really just a recursive visit. We must also clear the FINALIZATION_ORDERING bit at the end (state 2 to state 3) to clean up before the next collection.

**Designing thread pickling or “the Essence of Stackless Python”**

Note from 2007-07-22: This document is slightly out of date and should be turned into a description of pickling. Some research is necessary to get rid of explicit resume points, etc…

Thread pickling is a unique feature in Stackless Python and should be implemented for PyPy pretty soon.

**What is meant by pickling?**

I’d like to define thread pickling as a restartable subset of a running program. The re- runnable part should be based upon Python frame chains, represented by coroutines, tasklets or any other application level switchable subcontext. It is surely possible to support pickling of arbitrary interlevel state, but this seems to be not mandatory as long as we consider Stackless as the reference implementation. Extensions of this might be considered when the basic task is fulfilled.

Pickling should create a re-startable coroutine-alike thing that can run on a different machine, same Python version, but not necessarily the same PyPy translation. This belongs to the harder parts.
What is not meant by pickling?

Saving the whole memory state and writing a loader that reconstructs the whole binary with its state in memory is not what I consider a real solution. In some sense, this can be a fall-back if we fail in every other case, but I consider it really nasty for the C backend.

If we had a dynamic backend that supports direct creation of the program and its state (example: a Forth backend), I would see it as a valid solution, since it is relocatable. It is of course a possible fall-back to write such a backend of we fail otherwise.

There are some simple steps and some more difficult ones. Let’s start with the simple.

Basic necessities

Pickling of a running thread involves a bit more than normal object pickling, because there exist many objects which don’t have a pickling interface, and people would not care about pickling them at all. But with thread pickling, these objects simply exist as local variables and are needed to restore the current runtime environment, and the user should not have to know what goes into the pickle.

Examples are

- generators
- frames
- cells
- iterators
- tracebacks

to name just a few. Fortunately most of these objects already have got a pickling implementation in Stackless Python, namely the prickelpit.c file.

It should be simple and straightforward to redo these implementations. Nevertheless there is a complication. The most natural way to support pickling is providing a `__getstate__/__setstate__` method pair. This is ok for extension types like coroutines/tasklets which we can control, but it should be avoided for existing types.

Consider for instance frames. We would have to add a `__getstate__` and a `__setstate__` method, which is an interface change. Furthermore, we would need to support creation of frames by calling the frame type, which is not really intended.

For other types with are already callable, things get more complicated because we need to make sure that creating new instances does not interfere with existing ways to call the type.

Directly adding a pickling interface to existing types is quite likely to produce overlaps in the calling interface. This happened for instance, when the module type became callable, and the signature was different from what Stackless added before.

For Stackless, I used the copyreg module, instead, and created special surrogate objects as placeholders, which replace the type of the object after unpickling with the right type pointer. For details, see the prickelpit.c file in the Stackless distribution.

As a conclusion, pickling of tasklets is an addition to Stackless, but not meant to be an extension to Python. The need to support pickling of certain objects should not change the interface. It is better to decouple this and to use surrogate types for pickling which cannot collide with future additions to Python.
The real problem

There are currently some crucial differences between Stackless Python (SLP for now) and the PyPy Stackless support (PyPy for now) as far as it is grown. When CPython does a call to a Python function, there are several helper functions involved for adjusting parameters, unpacking methods and some more. SLP takes a hard time to remove all these C functions from the C stack before starting the Python interpreter for the function. This change of behavior is done manually for all the helper functions by figuring out, which variables are still needed after the call. It turns out that in most cases, it is possible to let all the helper functions finish their work and return form the function call before the interpreter is started at all.

This is the major difference which needs to be tackled for PyPy. Whenever we run a Python function, quite a number of functions incarnate on the C stack, and they get not finished before running the new frame. In case of a coroutine switch, we just save the whole chain of activation records - c function entrypoints with the saved block variables. This is ok for coroutine switching, but in the sense of SLP, it is rather incomplete and not stackless at all. The stack still exists, we can unwind and rebuild it, but it is a problem.

Why a problem?

In an ideal world, thread pickling would just be building chains of pickled frames and nothing else. For every different extra activation record like mentioned above, we have the problem of how to save this information. We need a representation which is not machine or compiler dependent. Right now, PyPy is quite unstable in terms of which blocks it will produce, what gets inlined, etc. The best solution possible is to try to get completely rid of these extra structures.

Unfortunately this is not even possible with SLP, because there are different flavors of state which make it hard to go without extra information.

SLP switching strategies

SLP has undergone several rewrites. The first implementation was aiming at complete collaboration. A new frame’s execution was deferred until all the preparational C function calls had left the C stack. There was no extra state to be saved.

Well, this is only partially true - there are a couple of situations where a recursive call could not be avoided, since the necessary support would require heavy rewriting of the implementation.

Examples are

- map is a stateful implementation of iterating over a sequence of operations. It can be made non-recursive if the map operation creates its own frame to keep state.
- __init__ looks trivial, but the semantics is that the return value of __init__ is supposed to be None, and CPy has a special check for this after the call. This might simply be ignored, but it is a simple example for a case that cannot be handled automatically.
- things like operator.__add__ can theoretically generate a wild pattern of recursive calls while CPy tries to figure out if it is a numeric add or a sequence add, and other callbacks may occur when methods like __coerce__ get involved. This will never be solved for SLP, but might get a solution by the strategy outlined below.

The second implementation took a radically different approach. Context switches were done by hijacking parts of the C stack, storing them away and replacing them by the stack fragment that the target needs. This is very powerful and allows to switch even in the context of foreign code. With a little risk, I was even able to add concurrency to foreign Fortran code.

The above concept is called Hard (switching), the collaborative Soft (switching). Note that an improved version of Hard is still the building block for greenlets, which makes them not really green - I’d name it yellow.
The latest SLP rewrites combine both ideas, trying to use Soft whenever possible, but using Hard when nested interpreters are in the way.

Notabene, it was never tried to pickle tasklets when Hard was involved. In SLP, pickling works with Soft. To gather more pickleable situations, you need to invent new frame types or write replacement Python code and switch it using Soft.

**Analogies between SLP and PyPy**

Right now, PyPy saves C state of functions in tiny activation records: the alive variables of a block, together with the entry point of the function that was left. This is an improvement over storing raw stack slices, but the pattern is similar: The C stack state gets restored when we switch.

In this sense, it was the astonishing resume when Richard and I discussed this last week: PyPy essentially does a variant of Hard switching! At least it does a compromise that does not really help with pickling.

On the other hand, this approach is half the way. It turns out to be an improvement over SLP not to have to avoid recursions in the first place. Instead, it seems to be even more elegant and efficient to get rid of unnecessary state right in the context of a switch and no earlier!

**Ways to handle the problem in a minimalistic way**

Comparing the different approaches of SLP and PyPy, it appears to be not necessary to change the interpreter in the first place. PyPy does not need to change its calling behavior in order to be cooperative. The key point is to find out which activation records need to be stored at all. This should be possible to identify as a part of the stackless transform.

Consider the simple most common case of calling a normal Python function. There are several calls to functions involved, which do preparational steps. Without trying to be exact (this is part of the work to be done), involved steps are

- decode the arguments of the function
- prepare a new frame
- store the arguments in the frame
- execute the frame
- return the result

Now assume that we do not execute the frame, but do a context switch instead, then right now a sequence of activation records is stored on the heap. If we want to re-activate this chain of activation records, what do we really need to restore before we can do the function call?

- the argument decoding is done, already, and the fact that we could have done the function call shows, that no exception occurred. We can ignore the rest of this activation record and do the housekeeping.
- the frame is prepared, and arguments are stored in it. The operation succeeded, and we have the frame. We can ignore exception handling and just do housekeeping by getting rid of references.
- for executing the frame, we need a special function that executes frames. It is possible that we need different flavors due to contexts. SLP does this by using different registered functions which operate on a frame, depending on the frame’s state (first entry, reentry after call, returning, yielding etc)
- after executing the frame, exceptions need to be handled in the usual way, and we should return to the issuer of the call.
Some deeper analysis is needed to get these things correct. But it should have become quite clear, that after all the preparational steps have been done, there is no other state necessary than what we have in the Python frames: bound arguments, instruction pointer, that’s it.

My proposal is now to do such an analysis by hand, identify the different cases to be handled, and then trying to find an algorithm that automatically identifies the blocks in the whole program, where the restoring of the C stack can be avoided, and we can jump back to the previous caller, directly.

A rough sketch of the necessary analysis:

for every block in an RPython function that can reach unwind: Analyze control flow. It should be immediately leading to the return block with only one output variable. All other alive variables should have ended their liveness in this block.

I think this will not work in the first place. For the bound frame arguments for instance, I think we need some notation that these are held by the frame, and we can drop their liveness before doing the call, hence we don’t need to save these variables in the activation record, and hence the whole activation record can be removed.

As a conclusion of this incomplete first analysis, it seems to be necessary to identify useless activation records in order to support pickling. The remaining, irreducible activation records should then be those which hold a reference to a Python frame. Such a chain is pickleable if its root points back to the context switching code of the interp-level implementation of coroutines.

As an observation, this transform not only enables pickling, but also is an optimization, if we can avoid saving many activation records.

Another possible observation which I hope to be able to prove is this: The remaining irreducible activation records which don’t just hold a Python frame are those which should be considered special. They should be turned into something like special frames, and they would be the key to make PyPy completely stackless, a goal which is practically impossible for SLP! These activation records would need to become part of the official interface and need to get naming support for their necessary functions.

I wish to stop this paper here. I believe everything else needs to be tried in an implementation, and this is so far all I can do just with imagination.

best - chris

**Just an addition after some more thinking**

Actually it struck me after checking this in, that the problem of determining which blocks need to save state and which not it not really a Stackless problem. It is a system-immanent problem of a missing optimization that we still did not try to solve.

Speaking in terms of GC transform, and especially the refcounting, it is probably easy to understand what I mean. Our current refcounting implementation is naive, in the sense that we do not try to do the optimizations which every extension writer does by hand: We do not try to save references.

This is also why I’m always arguing that refcounting can be and effectively is efficient, because CPython does it very well.

Our refcounting is not aware of variable lifeness, it does not track references which are known to be held by other objects. Optimizing that would do two things: The refcounting would become very efficient, since we would save some 80% of it. The second part, which is relevant to the picking problem is this: By doing a proper analysis, we already would have lost references to all the variables which we don’t need to save any longer, because we know that they are held in, for instance, frames.

I hope you understand that: If we improve the life-time analysis of variables, the sketched problem of above about which blocks need to save state and which don’t, should become trivial and should just vanish. Doing this correctly will solve the pickuping problem quasi automatically, leading to a more efficient implementation at the same time.
I hope I told the truth and will try to prove it.
ciao - chris

**Possible improvements of the rpython language**

**Improve the interpreter API**

- Rationalize the modules, and the names, of the different functions needed to implement a pypy module. A typical rpython file is likely to contain many `import` statements:

```python
from pypy.interpreter.baseobjspace import W_Root
from pypy.interpreter.gateway import ObjSpace, W_Root
from pypy.interpreter.argument import Arguments
from pypy.interpreter.typedef import Typedef, GetSetProperty
from pypy.interpreter.typedef import interp_attrproperty, interp_attrproperty_w
from pypy.interpreter.gateway import interp2app
from pypy.interpreter.error import OperationError
from rpython.rtyper.lltypesystem import rffi, lltype
```

- A more direct declarative way to write Typedef:

```python
class W_Socket(W_Root):
    _typedef_name_ = 'socket'
    _typedef_base_ = W_EventualBaseClass

    @interp2app_method("connect", ['self', ObjSpace, W_Root])
    def connect_w(self, space, w_addr):
        ...
```

- Support for metaclasses written in rpython. For a sample, see the skipped test `pypy.objspace.std.test.TestTypeObject.test_metaclass_typedef`

**RPython language**

- Arithmetic with unsigned integer, and between integer of different signedness, when this is not ambiguous. At least, comparison and assignment with constants should be allowed.

- Allocate variables on the stack, and pass their address (“by reference”) to `llexternal` functions. For a typical usage, see `rpython.rlib.rsocket.RSocket.getsockopt_int`.

**Extensible type system for `llexternal`**

`llexternal` allows the description of a C function, and conveys the same information about the arguments as a C header. But this is often not enough. For example, a parameter of type `int*` is converted to `rffi.CArrayPtr(rffi.INT)`, but this information is not enough to use the function. The parameter could be an array of int, a reference to a single value, for input or output...

A “type system” could hold this additional information, and automatically generate some conversion code to ease the usage of the function from rpython. For example:

```c
# double frexp(double x, int *exp);
fexp = llexternal("frexp", [rffi.DOUBLE, OutPtr(rffi.int)], rffi.DOUBLE)
```
OutPtr indicates that the parameter is output-only, which need not to be initialized, and which value is returned to the caller. In rpython the call becomes:

\[
\text{fraction, exponent} = \text{frexp(value)}
\]

Also, we could imagine that one item in the llexternal argument list corresponds to two parameters in C. Here, OutCharBufferN indicates that the caller will pass a rpython string; the framework will pass buffer and length to the function:

```c
ssize_t write(int fd, const void *buf, size_t count);
write = llexternal("write", [rffi.INT, CharBufferAndSize], rffi.SSIZE_T)
```

The rpython code that calls this function is very simple:

```python
written = write(fd, data)
```

compared with the present:

```python
count = len(data)
buf = rffi.get_nonmovingbuffer(data)
try:
    written = rffi.cast(lltype.Signed, os_write(
        rffi.cast(rffi.INT, fd),
        buf, rffi.cast(rffi.SIZE_T, count)))
finally:
    rffi.free_nonmovingbuffer(data, buf)
```

Typemaps are very useful for large APIs where the same conversions are needed in many places. XXX example

**PyPy’s ctypes implementation**

**Summary**

Terminology:

- application level code - code written in full Python
- interpreter level code - code written in RPython, compiled to something else, say C, part of the interpreter.

PyPy’s ctypes implementation in its current state proves the feasibility of implementing a module with the same interface and behavior for PyPy as ctypes for CPython.

PyPy’s implementation internally uses libffi like CPython’s ctypes. In our implementation as much as possible of the code is written in full Python, not RPython. In CPython’s situation, the equivalent would be to write as little as possible code in C. We essentially favored rapid experimentation over worrying about speed for this first trial implementation. This allowed to provide a working implementation with a large part of ctypes features in 2 months real time.

We reused the `ctypes` package version 1.0.2 as-is from CPython. We implemented `_ctypes` which is a C module in CPython mostly in pure Python based on a lower-level layer extension module `_rawffi`.

**Low-level part: _rawffi**

This PyPy extension module (`pypy/module/_rawffi`) exposes a simple interface to create C objects (arrays and structures) and calling functions in dynamic libraries through libffi. Freeing objects in most cases and making sure that objects referring to each other are kept alive is responsibility of the higher levels.

This module uses bindings to libffi which are defined in `rpython/rlib/libffi.py`.
We tried to keep this module as small as possible. It is conceivable that other implementations (e.g. Jython) could use our ctypes implementation by writing their version of _rawffi.

High-level parts

The reused ctypes package lives in lib_pypy/ctypes. _ctypes implementing the same interface as _ctypes in CPython is in lib_pypy/_ctypes.

Discussion and limitations

Reimplementing ctypes features was in general possible. PyPy supports pluggable garbage collectors, some of them are moving collectors, this means that the strategy of passing direct references inside Python objects to an external library is not feasible (unless the GCs support pinning, which is not the case right now). The consequence of this is that sometimes copying instead of sharing is required, this may result in some semantics differences. C objects created with _rawffi itself are allocated outside of the GC heap, such that they can be passed to external functions without worries.

Porting the implementation to interpreter-level should likely improve its speed. Furthermore the current layering and the current _rawffi interface require more object allocations and copying than strictly necessary; this too could be improved.

Here is a list of the limitations and missing features of the current implementation:

- ctypes.pythonapi is missing. In previous versions, it was present and redirected to the cpyext C API emulation layer, but our implementation did not do anything sensible about the GIL and the functions were named with an extra “Py”, for example PyPyInt_FromLong(). It was removed for being unhelpful.
- We copy Python strings instead of having pointers to raw buffers
- Features we did not get to implement:
  - custom alignment and bit-fields
  - resizing (resize() function)
  - non-native byte-order objects
  - callbacks accepting by-value structures
  - slight semantic differences that ctypes makes between its primitive types and user subclasses of its primitive types

Running application examples

pyglet is known to run. We also had some success with pygame-ctypes (which is no longer maintained) and with a snapshot of the experimental pysqlite-ctypes. We will only describe how to run the pyglet examples.

pyglet

We tried pyglet checking it out from its repository at revision 1984.

From pyglet, the following examples are known to work:

- opengl.py
- multiple_windows.py
The pypy-c translated to run the ctypes tests can be used to run the pyglet examples as well. They can be run like e.g.:

```
$ cd pyglet/
$ PYTHONPATH=. ../ctypes-stable/pypy/goal/pypy-c examples/opengl.py
```

they usually should be terminated with ctrl-c. Refer to the their doc strings for details about how they should behave.

The following examples don’t work for reasons independent from ctypes:

- `image_convert.py` needs PIL
- `image_display.py` needs PIL
- `astraea/astraea.py` needs PIL

We did not try the following examples:

- `media_player.py` needs avbin or at least a proper sound card setup for .wav files
- `video.py` needs avbin
- `soundscape` needs avbin

**A JIT-aware profiler**

Goal: have a profiler which is aware of the PyPy JIT and which shows which percentage of the time have been spent in which loops.

Long term goal: integrate the data collected by the profiler with the jitviewer.

The idea is record an event in the PYPYLOG everytime we enter and exit a loop or a bridge.

**Expected output**

```
[100] {jit-profile-enter loop1 # e.g. an entry bridge [101] jit-profile-enter} ... [200] {jit-profile-enter loop0 # JUMP from loop1 to loop0 [201] jit-profile-enter} ... [500] {jit-profile-exit loop0 # e.g. because of a failing guard [501] jit-profile-exit}
```

In this example, the exiting from loop1 is implicit because we are entering loop0. So, we spent 200-100=100 ticks in the entry bridge, and 500-200=300 ticks in the actual loop.

**What to do about “inner” bridges?**

“Inner bridges” are those bridges which jump back to the loop where they originate from. There are two possible ways of dealing with them:

1. we ignore them: we record when we enter the loop, but not when we jump to a compiled inner bridge. The exit event will be recorded only in case of a non-compiled guard failure or a JUMP to another loop
2. we record the enter/exit of each inner bridge
The disadvantage of solution (2) is that there are certain loops which takes bridges at every single iteration. So, in this case we would record a huge number of events, possibly adding a lot of overhead and thus making the profiled data useless.

**Detecting the enter to/exit from a loop**

Ways to enter:
- just after the tracing/compilation
- from the interpreter, if the loop has already been compiled
- from another loop, via a JUMP operation
- from a hot guard failure (which we ignore, in case we choose solution (1) above)
- XXX: am I missing anything?

Ways to exit:
- guard failure (entering blackhole)
- guard failure (jumping to a bridge) (ignored in case of solution (1))
- jump to another loop
- XXX: am I missing anything?

About call_assembler: I think that at the beginning, we should just ignore call_assembler: the time spent inside the call will be accounted to the loop calling it.

**Rawrefcount and the GC**

**GC Interface**

“PyObject” is a raw structure with at least two fields, ob_refcnt and ob_pypy_link. The ob_refcnt is the reference counter as used on CPython. If the PyObject structure is linked to a live PyPy object, its current address is stored in ob_pypy_link and ob_refcnt is bumped by either the constant REFCNT_FROM_PYPY, or the constant REFCNT_FROM_PYPY_LIGHT (== REFCNT_FROM_PYPY + SOME_HUGE_VALUE) (to mean “light finalizer”).

Most PyPy objects exist outside cpyext, and conversely in cpyext it is possible that a lot of PyObjects exist without being seen by the rest of PyPy. At the interface, however, we can “link” a PyPy object and a PyObject. There are two kinds of link:

```c
rawrefcount.create_link_pypy(p, ob)
```

Makes a link between an existing object gcref ‘p’ and a newly allocated PyObject structure ‘ob’. ob->ob_refcnt must be initialized to either REFCNT_FROM_PYPY, or REFCNT_FROM_PYPY_LIGHT. (The second case is an optimization: when the GC finds the PyPy object and PyObject no longer referenced, it can just free() the PyObject.)

```c
rawrefcount.create_link_pyobj(p, ob)
```

Makes a link from an existing PyObject structure ‘ob’ to a newly allocated W_CPyExtPlaceHolderObject ‘p’. You must also add REFCNT_FROM_PYPY to ob->ob_refcnt. For cases where the PyObject contains all the data, and the PyPy object is just a proxy. The W_CPyExtPlaceHolderObject should have only a field that contains the address of the PyObject, but that’s outside the scope of the GC.

```c
rawrefcount.from_obj(p)
```
If there is a link from object ‘p’ made with create_link_pypy(), returns the corresponding ‘ob’. Otherwise, returns NULL.

rawrefcount.to_obj(Class, ob)

Returns ob->ob_pypy_link, cast to an instance of ‘Class’.

Collection logic

Objects existing purely on the C side have ob->ob_pypy_link == 0; these are purely reference counted. On the other hand, if ob->ob_pypy_link != 0, then ob->ob_refcnt is at least REFCNT_FROM_PYPY and the object is part of a “link”.

The idea is that links whose ‘p’ is not reachable from other PyPy objects and whose ‘ob->ob_refcnt’ is REFCNT_FROM_PYPY or REFCNT_FROM_PYPY_LIGHT are the ones who die. But it is more messy because PyObjects still (usually) need to have a tp_dealloc called, and this cannot occur immediately (and can do random things like accessing other references this object points to, or resurrecting the object).

Let P = list of links created with rawrefcount.create_link_pypy() and O = list of links created with rawrefcount.create_link_pyobj(). The PyPy objects in the list O are all W_CPyExtPlaceHolderObject: all the data is in the PyObjects, and all outside references (if any) are in C, as PyObject * fields.

So, during the collection we do this about P links:

```python
for (p, ob) in P:
    if ob->ob_refcnt != REFCNT_FROM_PYPY
        mark ‘p’ as surviving, as well as all its dependencies
```

At the end of the collection, the P and O links are both handled like this:

```python
for (p, ob) in P + O:
    if p is not surviving:  # even if ‘ob’ might be surviving
        unlink p and ob
    if ob->ob_refcnt == REFCNT_FROM_PYPY_LIGHT:
        free(ob)
    elif ob->ob_refcnt > REFCNT_FROM_PYPY_LIGHT:
        ob->ob_refcnt -= REFCNT_FROM_PYPY_LIGHT
    else:
        ob->ob_refcnt -= REFCNT_FROM_PYPY
    if ob->ob_refcnt == 0:
        invoke _Py_Dealloc(ob) later, outside the GC
```

GC Implementation

We need two copies of both the P list and O list, for young or old objects. All four lists can be regular AddressLists of ‘ob’ objects.

We also need an AddressDict mapping ‘p’ to ‘ob’ for all links in the P list, and update it when PyPy objects move.

Further notes

XXX XXX the rest is the ideal world, but as a first step, we’ll look XXX for the minimal tweaks needed to adapt the existing cpyext XXX
For objects that are opaque in CPython, like `<dict>`, we always create a PyPy object, and then when needed we make an empty PyObject and attach it with `create_link_pypy()/REFCNT_FROM_PYPY_LIGHT`.

For `<int>` and `<float>` objects, the corresponding PyObjects contain a “long” or “double” field too. We link them with `create_link_pypy()` and we can use `REFCNT_FROM_PYPY_LIGHT` too: `tp_dealloc` doesn’t need to be called, and instead just calling `free()` is fine.

For `<str>` objects, we need both a PyPy and a PyObject side. These are made with `create_link_pypy()/REFCNT_FROM_PYPY`.

For custom PyXxxObjects allocated from the C extension module, we need `create_link_pyobj()`.

For `<str>` or `<unicode>` objects coming from PyPy, we use `create_link_pypy()/REFCNT_FROM_PYPY_LIGHT` with a PyObject preallocated with the size of the string. We copy the string lazily into that area if `PyString_AS_STRING()` is called.

For `<str>`, `<unicode>`, `<tuple>` or `<list>` objects in the C extension module, we first allocate it as only a PyObject, which supports mutation of the data from C, like CPython. When it is exported to PyPy we could make a W_CPyExtPlaceHolderObject with `create_link_pyobj()`.

For `<tuple>` objects coming from PyPy, if they are not specialized, then the PyPy side holds a regular reference to the items. Then we can allocate a PyTupleObject and store in it borrowed PyObject pointers to the items. Such a case is created with `create_link_pypy()/REFCNT_FROM_PYPY_LIGHT`. If it is specialized, then it doesn’t work because the items are created just-in-time on the PyPy side. In this case, the PyTupleObject needs to hold real references to the PyObject items, and we use `create_link_pypy()/REFCNT_FROM_PYPY`. In all cases, we have a C array of PyObjects that we can directly return from `PySequence_Fast_ITEMS`, `PyTuple_ITEMS`, `PyTuple_GetItem`, and so on.

For `<list>` objects coming from PyPy, we can use a cpyext list strategy. The list turns into a PyListObject, as if it had been allocated from C in the first place. The special strategy can hold (only) a direct reference to the PyListObject, and we can use either `create_link_pyobj()` or `create_link_pypy()` (to be decided). `PySequence_Fast_ITEMS` then works for lists too, and `PyList_GetItem` can return a borrowed reference, and so on.

### 3.4.8 Distributed and agile development in PyPy

PyPy isn’t just about producing code - it’s also about how we produce code. The challenges of coordinating work within a community and making sure it is fused together with the parts of the project that is EU funded are tricky indeed. Our aim is of course to make sure that the communities way of working is disturbed as little as possible and that contributing to PyPy still feels fun and interesting (:-) but also to try to show to the EU as well as other funded projects that open source ideas, tools and methods are really good ways of running development projects. So the way PyPy as a project is being run - distributed and agile - is something we think might be of use to other open source development projects and commercial projects.

Main methods for achieving this is:

- Sprint driven development
- Sync meetings

Main tools for achieving this is:

- `py.test` - automated testing
- `Mercurial` - version control
- Transparent communication and documentation (mailinglists, IRC, tutorials etc etc)

### Sprint driven development:

What is a sprint and why are we sprinting?
Originally the sprint methodology used in the Python community grew from practices within Zope3 development. The definition of a sprint is “two-day or three-day focused development session, in which developers pair off together in a room and focus on building a particular subsystem”.

Other typical sprint factors:

- no more than 10 people (although other projects as well as PyPy haven been noted to have more than that. This is the recommendation and it is probably based on the idea of having a critical mass of people who can interact/communicate and work without adding the need for more than just the absolute necessary coordination time. The sprints during 2005 and 2006 have been having ca 13-14 people per sprint, the highest number of participants during a PyPy sprint has been 24 developers)

- a coach (the coach is the “manager” of the sprint, he/she sets the goals, prepares, leads and coordinate the work and track progress and makes this visible for the team. Important to note here - PyPy have never had coaches in our sprints. Instead we hold short status meetings in the whole group, decisions are made in the same way. So far this have worked well and we still have been able to achieve tremendous results under stressed conditions, releases and such like. What we do have is a local organizer, often a developer living in the area and one more developer who prepares and organizes sprint. They do not “manage” the sprint when its started - their role is more of the logistic nature. This doesn’t mean that we wont have use for the coach technique or something similar in the future).

- only coding (this is a tough one. There have been projects who have used the sprinting method to just visionalize och gather input. PyPy have had a similar brainstorming start up sprint. So far though this is the official line although again, if you visit a PyPy sprint we are doing quite a lot of other small activities in subgroups as well - planning sprints, documentation, coordinating our EU deliverables and evaluation etc. But don’t worry - our main focus is programming :-

- using XP techniques (mainly pairprogramming and unit testing - PyPy is leaning heavily on these aspects). Pairing up core developers with people with different levels of knowledge of the codebase have had the results that people can quite quickly get started and join in the development. Many of our participants (new to the project and the codebase) have expressed how pairprogramming in combination with working on the automated tests have been a great way of getting started. This is of course also a dilemma because our core developers might have to pair up to solve some extra hairy problems which affects the structure and effect of the other pairs.

It is a method that fits distributed teams well because it gets the team focused around clear (and challenging) goals while working collaborative (pairprogramming, status meeting, discussions etc) as well as accelerated (short increments and tasks, “doing” and testing instead of long start ups of planning and requirement gathering). This means that most of the time a sprint is a great way of getting results, but also to get new people acquainted with the codebase. It is also a great method for dissemination and learning within the team because of the pairprogramming.

If sprinting is combined with actually moving around and having the sprint close to the different active developer groups in the community as well as during conferences like PyCon and EuroPython, the team will have an easier task of recruiting new talents to the team. It also vitalizes the community and increases the contact between the different Python implementation projects.

As always with methodologies you have to adapt them to fit your project (and not the other way around which is much too common). The PyPy team have been sprinting since early 2003 and have done 22 sprints so far, 19 in Europe, 2 in the USA and 1 in Asia. Certain practices have proven to be more successful within this team and those are the one we are summarizing here.

**How is it done?**

There are several aspects of a sprint. In the PyPy team we focus on: 1. Content (goal) 2. Venue 3. Information 4. Process

1. Content (goal) is discussed on mailinglists (pypy-dev) and on IRC ca one month before the event. Beforehand we have some rough plans called “between sprints” and the sprintplan is based on the status of those issues
but also with a focus on upcoming releases and deliverables. Usually its the core developers who does this but the transparency and participation have increased since we started with our weekly “pypy-sync meetings” on IRC. The sync meetings in combination with a rough in between planning makes it easier for other developer to follow the progress and thus participating in setting goals for the upcoming sprints.

The goal needs to be challenging or it won’t rally the full effort of the team, but it must not be unrealistic as that tends to be very frustrating and dissatisfying. It is also very important to take into account the participants when you set the goal for the sprint. If the sprint takes place connected to a conference (or similar open events) the goals for the actual coding progress should be set lower (or handled in another way) and focus should shift to dissemination and getting new/interested people to a certain understanding of the PyPy codebase. Setting the right goal and making sure this is a shared one is important because it helps the participants coming in with somewhat similar expectations ;-)

2. Venue - in the PyPy project we have a rough view on where we are sprinting a few months ahead. No detailed plans have been made that far in advance. Knowing the dates and the venue makes flight bookings easier ;-) The venue is much more important than one would think. We need to have a somewhat comfortable environment to work in (where up to 15 people can sit and work), this means tables and chairs, light and electricity outlets. Is it a venue needing access cards so that only one person is allowed to open? How long can you stay - 24 hours per day or does the landlord want the team evacuated by 23:00? These are important questions that can gravely affect the “feel and atmosphere” of the sprint as well as the desired results!

Also, somewhat close to low cost places to eat and accommodate participants. Facilities for making tea/coffee as well as some kind of refrigerator for storing food. A permanent Internet connection is a must - has the venue were the sprint is planned to be weird rules for access to their network etc etc?

Whiteboards are useful tools and good to have. Beamers (PyPy jargon for a projector) are very useful for the status meetings and should be available, at least 1. The project also owns one beamer - specifically for sprint purposes.

The person making sure that the requirements for a good sprint venue is being met should therefore have very good local connections or, preferably live there.

3. Information - discussions about content and goals (pre announcements) are usually carried out on pypy-dev (mailinglist/IRC). All other info is distributed via email on pypy-sprint mailinglist and as web pages on codespeak. When dates, venue and content is fully decided a sprint announcement is being made and sent out to pypy-dev and pypy-sprint as well as more general purpose mailing lists like comp.lang.python and updated on codespeak - this happens 2-4 weeks before the sprint. It’s important that the sprint announcements points to information about local transportation (to the country and to the city and to the venue), currency issues, food and restaurants etc. There are also webpages in which people announce when they will arrive and where they are accommodated.

The planning text for the sprint is updated up till the sprint and is then used during the status meetings and between to track work. After the sprint (or even better: in between so that the memory is fresh) a sprint report is written by one of the developers and updated to codespeak, this is a kind of summary of the entire sprint and it tells of the work done and the people involved.

One very important strategy when planning the venue is cost efficiency. Keeping accommodation and food/travel costs as low as possible makes sure that more people can afford to visit or join the sprint fully. The partially EU funded parts of the project do have a so called sprint budget which we use to try to help developers to participate in our sprints (travel expenses and accommodation) and because most of the funding is so called matched funding we pay for most of our expenses in our own organizations and companies anyway.

4. Process - a typical PyPy sprint is 7 days with a break day in the middle. Usually sprinters show up the day before the sprint starts. The first day has a start up meeting, with tutorials if there are participants new to the project or if some new tool or feature have been implemented. A short presentation of the participants and their background and expectations is also good to do. Unfortunately there is always time spent the first day, mostly in the morning when people arrive to get the internet and server infrastructure up and running. That is why we are, through documentation, trying to get participants to set up the tools and configurations needed before they
arrive to the sprint.

Approximate hours being held are 10-17, but people tend to stay longer to code during the evenings. A short status meeting starts up the day and work is “paired” out according to need and wishes. The PyPy sprints are developer and group driven, because we have no “coach” our status meetings are very much group discussion while notes are taken and our planning texts are updated. Also - the sprint is done (planned and executed) within the developer group together with someone acquainted with the local region (often a developer living there). So within the team there is no one formally responsible for the sprints.

Suggestions for off hours activities and social events for the break day is a good way of emphasizing how important it is to take breaks - some pointers in that direction from the local organizer is good.

At the end of the sprint we do a technical summary (did we achieve the goals/content), what should be a rough focus for the work until the next sprint and the sprint wheel starts rolling again ;-) An important aspect is also to evaluate the sprint with the participants. Mostly this is done via emailed questions after the sprint, it could also be done as a short group evaluation as well. The reason for evaluating is of course to get feedback and to make sure that we are not missing opportunities to make our sprints even more efficient and enjoyable.

The main challenge of our sprint process is the fact that people show up at different dates and leave at different dates. That affects the shared introduction (goals/content, tutorials, presentations etc) and also the closure - the technical summary etc. Here we are still struggling to find some middle ground - thus increases the importance of feedback.

Can I join in?

Of course. Just follow the work on pypy-dev and if you specifically are interested in information about our sprints - subscribe to pypy-sprint@codespeak.net and read the news on codespeak for announcements etc.

If you think we should sprint in your town - send us an email - we are very interested in using sprints as a way of making contact with active developers (Python/compiler design etc)!

### 3.4.9 Embedding PyPy (DEPRECATED)

PyPy has a very minimal and a very strange embedding interface, based on the usage of cffi and the philosophy that Python is a better language than C. It was developed in collaboration with Roberto De Ioris from the uwsgi project. The PyPy uwsgi plugin is a good example of using the embedding API.

**NOTE:** You need a PyPy compiled with the option `--shared`, i.e. with a `libpypy-c.so` or `pypy-c.dll` file. This is the default in recent versions of PyPy.

The resulting shared library exports very few functions, however they are enough to accomplish everything you need, provided you follow a few principles. The API is:

```c
void rpython_startup_code(void);
```

This is a function that you have to call (once) before calling anything else. It initializes the RPython/Pypy GC and does a bunch of necessary startup code. This function cannot fail.

```c
int pypy_setup_home(char* home, int verbose);
```

This function searches the PyPy standard library starting from the given “PyPy home directory”. The arguments are:
• home: path to an executable inside the pypy directory (can be a .so name, can be made up). Used to look up the standard library, and is also set as sys.executable. From PyPy 5.5, you can just say NULL here, as long as the libpypy-c.so/dylib/dll is itself inside this directory.

• verbose: if non-zero, it will print error messages to stderr

Function returns 0 on success or -1 on failure, can be called multiple times until the library is found.

```c
void pypy_init_threads(void);
```
Initialize threads. Only need to be called if there are any threads involved. Must be called after pypy_setup_home()

```c
int pypy_execute_source(char* source);
```
Execute the Python source code given in the source argument. In case of exceptions, it will print the Python traceback to stderr and return 1, otherwise return 0. You should really do your own error handling in the source. It’ll acquire the GIL.

Note: this is meant to be called only once or a few times at most. See the more complete example below. In PyPy <= 2.6.0, the globals dictionary is reused across multiple calls, giving potentially strange results (e.g. objects dying too early). In PyPy >= 2.6.1, you get a new globals dictionary for every call (but then, all globals dictionaries are all kept alive forever, in sys._pypy_execute_source).

```c
int pypy_execute_source_ptr(char* source, void* ptr);
```

Note: Not available in PyPy <= 2.2.1

Just like the above, except it registers a magic argument in the source scope as c_argument, where void* is encoded as Python int.

```c
void pypy_thread_attach(void);
```
In case your application uses threads that are initialized outside of PyPy, you need to call this function to tell the PyPy GC to track this thread. Note that this function is not thread-safe itself, so you need to guard it with a mutex.

### Minimal example

Note that this API is a lot more minimal than say CPython C API, so at first it’s obvious to think that you can’t do much. However, the trick is to do all the logic in Python and expose it via cffi callbacks. We write a little C program:

```c
#include "PyPy.h"
#include <stdio.h>
#include <stdlib.h>

static char source[] = "print 'hello from pypy'";

int main(void)
{
    int res;

    rpython_startup_code();
    /* Before PyPy 5.5, you may need to say e.g. "/opt/pypy/bin" instead of NULL. */
    res = pypy_setup_home(NULL, 1);
    if (res) {
        printf("Error setting pypy home!\n");
    }
}
```
return 1;
}
res = pypy_execute_source((char*)source);
if (res) {
    printf("Error calling pypy_execute_source!\n");
}
return res;

If we save it as x.c now, compile it and run it (on Linux) with:

```bash
$ gcc -g -o x x.c -lpypy-c -L/opt/pypy/bin -I/opt/pypy/include
$ LD_LIBRARY_PATH=/opt/pypy/bin ./x
hello from pypy
```

On OSX it is necessary to set the rpath of the binary if one wants to link to it, with a command like:

```bash
gcc -o x x.c -lpypy-c -L. -Wl,-rpath -Wl,@executable_path
./x
hello from pypy
```

More complete example

Note: Note that we do not make use of extern "Python", the new way to do callbacks in CFFI 1.4: this is because these examples use the ABI mode, not the API mode, and with the ABI mode you still have to use ffi.callback(). It is work in progress to integrate extern "Python" with the idea of embedding (and it is expected to ultimately lead to a better way to do embedding than the one described here, and that would work equally well on CPython and PyPy).

Typically we need something more to do than simply execute source. The following is a fully fledged example, please consult cffi documentation for details. It’s a bit longish, but it captures a gist what can be done with the PyPy embedding interface:

```python
import cffi

ffi = cffi.FFI()
ffi.cdef('''
struct API {
    double (*add_numbers)(double x, double y);
};
''')

# Better define callbacks at module scope, it's important to # keep this object alive.
@ffi.callback("double (double, double)"
def add_numbers(x, y):
    return x + y

def fill_api(ptr):
    global api

# file "interface.py"

```
You can compile and run it with:

```
$ gcc -g -o x x.c -lpypy-c -L/opt/pypy/bin -I/opt/pypy/include
$ LD_LIBRARY_PATH=/opt/pypy/bin ./x
```

As you can see, what we did is create a struct API that contains the custom API that we need in our particular case. This struct is filled by Python to contain a function pointer that is then called form the C side. It is also possible to do have other function pointers that are filled by the C side and called by the Python side, or even non-function-pointer fields: basically, the two sides communicate via this single C structure that defines your API.
Finding pypy_home

You can usually skip this section if you are running PyPy >= 5.5.

The function pypy_setup_home() takes as first parameter the path to a file from which it can deduce the location of the standard library. More precisely, it tries to remove final components until it finds lib-python and lib_pypy. There is currently no “clean” way (pkg-config comes to mind) to find this path. You can try the following (GNU-specific) hack (don’t forget to link against dl), which assumes that the libpypy-c.so is inside the standard library directory. (This must more-or-less be the case anyway, otherwise the pypy program itself would not run.)

```c
#if !_GNU_SOURCE
#define _GNU_SOURCE
#endif
#include <dlfcn.h>
#include <limits.h>
#include <stdlib.h>
// caller should free returned pointer to avoid memleaks
// returns NULL on error
char* guess_pypyhome(void) {
    // glibc-only (dladdr is why we #define _GNU_SOURCE)
    Dl_info info;
    void*_rpython_startup_code = dlsym(0,"rpython_startup_code");
    if (_rpython_startup_code == 0) {
        return 0;
    }
    if (dladdr(_rpython_startup_code, &info) != 0) {
        const char* lib_path = info.dli_fname;
        char* lib_realpath = realpath(lib_path, 0);
        return lib_realpath;
    }
    return 0;
}
```

Threading

In case you want to use pthreads, what you need to do is to call pypy_thread_follow from each of the threads that you created (but not from the main thread) and call pypy_init_threads from the main thread.

3.4.10 Transparent Proxies (DEPRECATED)

**Warning:** This is a feature that was tried experimentally long ago, and we found no really good use cases. The basic functionality is still there, but we don’t recommend using it. Some of the examples below might not work anymore (e.g. you can’t tproxy a list object any more). The rest can be done by hacking in standard Python. If anyone is interested in working on tproxy again, he is welcome, but we don’t regard this as an interesting extension.

PyPy’s Transparent Proxies allow routing of operations on objects to a callable. Application-level code can customize objects without interfering with the type system - `type(proxied_list) is list` holds true when `proxied_list` is a proxied built-in `list` - while giving you full control on all operations that are performed on `proxied_list`.

See [D12.1] for more context, motivation and usage of transparent proxies.
Example of the core mechanism

The following example proxies a list and will return 42 on any addition operations:

```python
$ py.py --objspace-std-withtproxy
>>> from __pypy__ import tproxy
>>> def f(operation, *args, **kwargs):
...     if operation == '__add__':
...         return 42
...     raise AttributeError
>>> i = tproxy(list, f)
>>> type(i)
list
>>> i + 3
42
```

Example of recording all operations on builtins

Suppose we want to have a list which stores all operations performed on it for later analysis. We can use the `lib_pypy/tputil.py` module to help with transparently proxying builtin instances:

```python
from tputil import make_proxy

history = []
def recorder(operation):
    history.append(operation)
    return operation.delegate()

>>> l = make_proxy(recorder, obj=[])  # make_proxy(recorder, obj=[]) creates a transparent list proxy that allows us to delegate operations to the recorder() function. Calling `type(l)` does not lead to any operation being executed at all.
>>> type(l)
list
>>> l.append(3)
>>> len(l)
1
>>> len(history)
2
```

Note that `append()` shows up as `__getattribute__()` and that `type(l)` does not show up at all - the type is the only aspect of the instance which the proxy controller cannot change.

Transparent Proxy PyPy builtins and support

If you are using the `--objspace-std-withtproxy` option the `__pypy__` module provides the following builtins:

- **tproxy**(type, controller)
  Returns a proxy object representing the given type and forwarding all operations on this type to the controller.
  On each operation, `controller(opname, *args, **kwargs)` will be called.

- **get_tproxy_controller**(obj)
  Returns the responsible controller for a given object. For non-proxyed objects `None` is returned.
tputil helper module

The lib_pypy/tputil.py module provides:

**make_proxy** *(controller, type, obj)*

Creates a transparent proxy controlled by the given controller callable. The proxy will appear as a completely regular instance of the given type, but all operations on it are sent to the specified controller - which receives a ProxyOperation instance on each operation. If type is not specified, it defaults to type(obj) if obj is specified.

ProxyOperation instances have the following attributes:

proxyobj
The transparent proxy object of this operation.

opname
The name of this operation.

args
Any positional arguments for this operation.

kwargs
Any keyword arguments for this operation.

obj
(Only if provided to make_proxy())
A concrete object.

delegate()
If a concrete object instance obj was specified in the call to make_proxy(), then proxyoperation.delegate() can be called to delegate the operation to the object instance.

**Further points of interest**

A lot of tasks could be performed using transparent proxies, including, but not limited to:

- Remote versions of objects, on which we can directly perform operations (think about transparent distribution)
- Access to persistent storage such as a database (imagine an SQL object mapper which looks like any other object).
- Access to external data structures, such as other languages, as normal objects (of course some operations could raise exceptions, but since operations are executed at the application level, that is not a major problem)

**Implementation Notes**

PyPy’s standard object space allows us to internally have multiple implementations of a type and change the implementation at run-time, while application-level code consistently sees the exact same type and object. Multiple performance optimizations using these features have already been implemented: alternative object implementations. Transparent Proxies use this architecture to provide control back to application-level code.

Transparent proxies are implemented on top of the standard object space, in pypy/objspace/std/proxyobject.py, pypy/objspace/std/proxyobject.py and pypy/objspace/std/transparency.py. To use them you will need to pass a --objspace-std-withproxy option to pypy or translate.py. This registers implementations named W_TransparentXxx - which usually correspond to an appropriate W_XxxObject - and includes some interpreter hacks for objects that are too close to the interpreter to be implemented in the standard object space. The types of objects that can be proxied this way are user created classes & functions, lists, dicts, exceptions, tracebacks and frames.

---

3.4. Project Documentation 117
3.5 Source Code Documentation

*object spaces* discusses the object space interface and several implementations.

*bytecode interpreter* explains the basic mechanisms of the bytecode interpreter and virtual machine.

*Standard Interpreter Optimizations* describes our various strategies for improving the performance of our interpreter, including alternative object implementations (for strings, dictionaries and lists) in the standard object space.

*dynamic-language translation* is a paper that describes the translation process, especially the flow object space and the annotator in detail. (This document is one of the EU reports.)

*parser* contains (outdated, unfinished) documentation about the parser.

*configuration documentation* describes the various configuration options that allow you to customize PyPy.

*command line reference*

*directory cross-reference*

3.5.1 The Object Space

Contents

- The Object Space
  - Introduction
  - Object Space Interface
    - Administrative Functions
    - Operations on Objects in the Object Space
    - Convenience Functions
    - Creation of Application Level objects
    - Conversions from Application Level to Interpreter Level
    - Data Members
  - The Standard Object Space
    - Introduction
    - Object types
  - Object Space proxies

Introduction

The object space creates all objects in PyPy, and knows how to perform operations on them. It may be helpful to think of an object space as being a library offering a fixed API: a set of operations, along with implementations that correspond to the known semantics of Python objects.

For example, `add()` is an operation, with implementations in the object space that perform numeric addition (when `add()` is operating on numbers), concatenation (when `add()` is operating on sequences), and so on.

We have some working object spaces which can be plugged into the bytecode interpreter:
• The Standard Object Space is a complete implementation of the various built-in types and objects of Python. The Standard Object Space, together with the bytecode interpreter, is the foundation of our Python implementation. Internally, it is a set of interpreter-level classes implementing the various application-level objects – integers, strings, lists, types, etc. To draw a comparison with CPython, the Standard Object Space provides the equivalent of the C structures PyIntObject, PyListObject, etc.

• various Object Space proxies wrap another object space (e.g. the standard one) and adds new capabilities, like lazily computed objects (computed only when an operation is performed on them), security-checking objects, distributed objects living on several machines, etc.

The various object spaces documented here can be found in pypy/objspace.

Note that most object-space operations take and return application-level objects, which are treated as opaque “black boxes” by the interpreter. Only a very few operations allow the bytecode interpreter to gain some knowledge about the value of an application-level object.

Object Space Interface

This is the public API that all Object Spaces implement:

Administrative Functions

ggetattr, setattr, delattr, getattr, setitem, delitem,
pos, neg, abs, invert, add, sub, mul, truediv, floordiv, div, mod,
divmod, pow, lshift, rshift, and_, or_, xor,
nonzero, hex, oct, int, float, long, ord,
lt, le, eq, ne, gt, ge, cmp, coerce, contains,
inplace_add, inplace_sub, inplace_mul, inplace_truediv,
inplace_floordiv, inplace_div, inplace_mod, inplace_pow,
inplace_lshift, inplace_rshift, inplace_and, inplace_or, inplace_xor,
get, set, delete, userdel

call (w_callable, w_args, w_kwds)
Calls a function with the given positional (w_args) and keyword (w_kwds) arguments.

index (w_obj)
Implements index lookup (as introduced in CPython 2.5) using w_obj. Will return a wrapped integer or long, or raise a TypeError if the object doesn’t have an __index__() special method.
**is_** (w_x, w_y)

Implements w_x is w_y.

**isinstance** (w_obj, w_type)

Implements issubtype() with type(w_obj) and w_type as arguments.

### Convenience Functions

The following functions are used so often that it was worthwhile to introduce them as shortcuts – however, they are not strictly necessary since they can be expressed using several other object space methods.

**eq_w** (w_obj1, w_obj2)

Returns True when w_obj1 and w_obj2 are equal. Shortcut for space.is_true(space.eq(w_obj1, w_obj2)).

**is_w** (w_obj1, w_obj2)

Shortcut for space.is_true(space.is_(w_obj1, w_obj2)).

**hash_w** (w_obj)

Shortcut for space.int_w(space.hash(w_obj)).

**len_w** (w_obj)

Shortcut for space.int_w(space.len(w_obj)).

**NOTE** that the above four functions return interpreter-level objects, not application-level ones!

**not_** (w_obj)

Shortcut for space.newbool(not space.is_true(w_obj)).

**finditem** (w_obj, w_key)

Equivalent to getattr(w_obj, w_key) but returns an interpreter-level None instead of raising a KeyError if the key is not found.

**call_function** (w_callable, *args_w, **kw_w)

Collects the arguments in a wrapped tuple and dict and invokes space.call(w_callable, ...).

**call_method** (w_object, 'method', ...)

Uses space.getattr() to get the method object, and then space.call_function() to invoke it.

**unpackiterable** (w_iterable[, expected_length=-1])

Iterates over w_x (using space.iter() and space.next()) and collects the resulting wrapped objects in a list. If expected_length is given and the length does not match, raises an exception.

Of course, in cases where iterating directly is better than collecting the elements in a list first, you should use space.iter() and space.next() directly.

**unpacktuple** (w_tuple[, expected_length=None])

Equivalent to unpackiterable(), but only for tuples.

**callable** (w_obj)

Implements the built-in callable().

### Creation of Application Level objects

**wrap** (x)

Deprecated! Eventually this method should disappear. Returns a wrapped object that is a reference to the interpreter-level object x. This can be used either on simple immutable objects (integers, strings, etc) to create a new wrapped object, or on instances of W_Root to obtain an application-level-visible reference to them. For
example, most classes of the bytecode interpreter subclass \texttt{W\_Root} and can be directly exposed to application-level code in this way - functions, frames, code objects, etc.

\textbf{newint} \((i)\)

Creates a wrapped object holding an integral value. \texttt{newint} creates an object of type \texttt{W\_IntObject}.

\textbf{newlong} \((l)\)

Creates a wrapped object holding an integral value. The main difference to \texttt{newint} is the type of the argument (which is \texttt{rpython.rlib.rbigint.rbigint}). On PyPy3 this method will return an \texttt{int} (PyPy2 it returns a \texttt{long}).

\textbf{newbytes} \((t)\)

The given argument is a rpython bytestring. Creates a wrapped object of type \texttt{bytes} (both on PyPy2 and PyPy3).

\textbf{newtext} \((t)\)

The given argument is a rpython bytestring. Creates a wrapped object of type \texttt{str}. On PyPy3 this will return a wrapped unicode object. The object will hold a utf-8-nosg decoded value of \texttt{t}. The “utf-8-nosg” codec used here is slightly different from the “utf-8” implemented in Python 2 or Python 3: it is defined as utf-8 without any special handling of surrogate characters. They are encoded using the same three-bytes sequence that encodes any char in the range from ‘\u0800’ to ‘\uffff’.

PyPy2 will return a bytestring object. No encoding/decoding steps will be applied.

\textbf{newbool} \((b)\)

Creates a wrapped \texttt{bool} object from an \texttt{interpreter-level} object.

\textbf{newtuple} \([w\_x, w\_y, w\_z, \ldots]\)

Creates a new wrapped tuple out of an interpreter-level list of wrapped objects.

\textbf{newlist} \([\ldots]\)

Creates a wrapped \texttt{list} from an interpreter-level list of wrapped objects.

\textbf{newdict} ()

Returns a new empty dictionary.

\textbf{newslice} \((w\_start, w\_end, w\_step)\)

Creates a new slice object.

\textbf{newunicode} \((ustr)\)

Creates a Unicode string from an rpython unicode string. This method may disappear soon and be replaced by :py:function::\texttt{newutf8}.

\textbf{newutf8} \((bytestr)\)

Creates a Unicode string from an rpython byte string, decoded as “utf-8-nosg”. On PyPy3 it is the same as :py:function::\texttt{newtext}.

Many more space operations can be found in \texttt{pypy/interpreter/baseobjspace.py} and \texttt{pypy/objspace/std/objspace.py}.

\section*{Conversions from Application Level to Interpreter Level}

\textbf{unwrap} \((w\_x)\)

Returns the interpreter-level equivalent of \texttt{w\_x} – use this \texttt{ONLY} for testing, because this method is not RPython and thus cannot be translated! In most circumstances you should use the functions described below instead.

\textbf{is\_true} \((w\_x)\)

Returns a interpreter-level boolean (\texttt{True} or \texttt{False}) that gives the truth value of the wrapped object \texttt{w\_x}.

This is a particularly important operation because it is necessary to implement, for example, if-statements in the language (or rather, to be pedantic, to implement the conditional-branching bytecodes into which if-statements are compiled).
int_w(w_x)
If w_x is an application-level integer or long which can be converted without overflow to an integer, return an interpreter-level integer. Otherwise raise TypeError or OverflowError.

bigint_w(w_x)
If w_x is an application-level integer or long, return an interpreter-level rbigint. Otherwise raise TypeError.

ObjSpace.bytes_w(w_x)
Takes an application level bytes (on PyPy2 this equals str) and returns a rpython byte string.

ObjSpace.text_w(w_x)
PyPy2 takes either a str and returns a rpython byte string, or it takes an unicode and uses the systems default encoding to return a rpython byte string.

On PyPy3 it takes a str and it will return an utf-8 encoded rpython string.

str_w(w_x)
Deprecated. use text_w or bytes_w instead If w_x is an application-level string, return an interpreter-level string. Otherwise raise TypeError.

unicode_w(w_x)
Takes an application level :py:class::unicode and return an interpreter-level unicode string. This method may disappear soon and be replaced by :py:function::text_w.

float_w(w_x)
If w_x is an application-level float, integer or long, return an interpreter-level float. Otherwise raise TypeError (or:py:exc::OverflowError in the case of very large longs).

getindex_w(w_obj[, w_exception=None ])
Call index(w_obj). If the resulting integer or long object can be converted to an interpreter-level int, return that. If not, return a clamped result if w_exception is None, otherwise raise the exception at the application level.

(If w_obj can’t be converted to an index, index() will raise an application-level TypeError.)

interp_w(RequiredClass, w_x[, can_be_None=False ])
If w_x is a wrapped instance of the given bytecode interpreter class, unwrap it and return it. If can_be_None is True, a wrapped None is also accepted and returns an interpreter-level None. Otherwise, raises an OperationError encapsulating a TypeError with a nice error message.

interpclass_w(w_x)
If w_x is a wrapped instance of an bytecode interpreter class – for example Function, Frame, Cell, etc. – return it unwrapped. Otherwise return None.

Data Members

space.builtin
The Module containing the builtins.

space.sys
The sys Module.

space.w_None
The ObjSpace’s instance of None.

space.w_True
The ObjSpace’s instance of True.
space.w_False
The ObjSpace’s instance of False.

space.w_Ellipsis
The ObjSpace’s instance of Ellipsis.

space.w_NotImplemented
The ObjSpace’s instance of NotImplemented.

space.w_int
space.w_float
space.w_long
space.w_tuple
space.w_str
space.w_unicode
space.w_type
space.w_instance
space.w_slice
Python’s most common basic type objects.

space.w_[XYZ]Error
Python’s built-in exception classes (KeyError, IndexError, etc).

ObjSpace.MethodTable
List of tuples containing (method_name, symbol, number_of_arguments,
list_of_special_names) for the regular part of the interface.

NOTE that tuples are interpreter-level.

ObjSpace.BuiltinModuleTable
List of names of built-in modules.

ObjSpace.ConstantTable
List of names of the constants that the object space should define.

ObjSpace.ExceptionTable
List of names of exception classes.

ObjSpace.IrregularOpTable
List of names of methods that have an irregular API (take and/or return non-wrapped objects).

The Standard Object Space

Introduction

The Standard Object Space (pypy/objspace/std/) is the direct equivalent of CPython’s object library (the Objects/
subdirectory in the distribution). It is an implementation of the common Python types in a lower-level language.

The Standard Object Space defines an abstract parent class, W_Object as well as subclasses like W_IntObject,
W_ListObject, and so on. A wrapped object (a “black box” for the bytecode interpreter’s main loop) is an instance
of one of these classes. When the main loop invokes an operation (such as addition), between two wrapped objects w1
and w2, the Standard Object Space does some internal dispatching (similar to Object/abstract.c in CPython)
and invokes a method of the proper W_XYZObject class that can perform the operation.

The operation itself is done with the primitives allowed by RPython, and the result is constructed as a wrapped object.
For example, compare the following implementation of integer addition with the function int_add() in Object/
intobject.c:
```python
def add__Int_Int(space, w_int1, w_int2):
    x = w_int1.intval
    y = w_int2.intval
    try:
        z = ovfcheck(x + y)
    except OverflowError:
        raise FailedToImplementArgs(space.w_OverflowError, space.wrap("integer addition"))
    return W_IntObject(space, z)
```

This may seem like a lot of work just for integer objects (why wrap them into \texttt{W\_IntObject} instances instead of using plain integers?), but the code is kept simple and readable by wrapping all objects (from simple integers to more complex types) in the same way.

(Interestingly, the obvious optimization above has actually been made in PyPy, but isn’t hard-coded at this level – see \textit{Standard Interpreter Optimizations}.)

### Object types

The larger part of the \texttt{pypy/objspace/std/} package defines and implements the library of Python’s standard built-in object types. Each type \texttt{xxx\_int, float, list, tuple, str, type, etc.} is typically implemented in the module \texttt{xxxobject.py}.

The \texttt{W\_AbstractXxxObject} class, when present, is the abstract base class, which mainly defines what appears on the Python-level type object. There are then actual implementations as subclasses, which are called \texttt{W\_XxxObject} or some variant for the cases where we have several different implementations. For example, \texttt{pypy/objspace/std/bytesobject.py} defines \texttt{W\_AbstractBytesObject}, which contains everything needed to build the \texttt{str} app-level type; and there are subclasses \texttt{W\_BytesObject} (the usual string) and \texttt{W\_Buffer} (a special implementation tweaked for repeated additions, in \texttt{pypy/objspace/std/bufferobject.py}). For mutable data types like lists and dictionaries, we have a single class \texttt{W\_ListObject} or \texttt{W\_DictMultiObject} which has an indirection to the real data and a strategy; the strategy can change as the content of the object changes.

From the user’s point of view, even when there are several \texttt{W\_AbstractXxxObject} subclasses, this is not visible: at the app-level, they are still all instances of exactly the same Python type. PyPy knows that (e.g.) the application-level type of its interpreter-level \texttt{W\_BytesObject} instances is \texttt{str} because there is a \texttt{typedef} class attribute in \texttt{W\_BytesObject} which points back to the string type specification from \texttt{pypy/objspace/std/bytesobject.py}; all other implementations of strings use the same \texttt{typedef} from \texttt{pypy/objspace/std/bytesobject.py}.

For other examples of multiple implementations of the same Python type, see \textit{Standard Interpreter Optimizations}.

### Object Space proxies

We have implemented several \texttt{proxy object spaces}, which wrap another object space (typically the standard one) and add some capabilities to all objects. To find out more, see \textit{Transparent Proxies (DEPRECATED)}.

### 3.5.2 Bytecode Interpreter

- \textit{Introduction and Overview}
Introduction and Overview

This document describes the implementation of PyPy’s Bytecode Interpreter and related Virtual Machine functionalities.

PyPy’s bytecode interpreter has a structure reminiscent of CPython’s Virtual Machine: It processes code objects parsed and compiled from Python source code. It is implemented in the pypy/interpreter/ directory. People familiar with the CPython implementation will easily recognize similar concepts there. The major differences are the overall usage of the object space indirection to perform operations on objects, and the organization of the built-in modules (described here).

Code objects are a nicely preprocessed, structured representation of source code, and their main content is bytecode. We use the same compact bytecode format as CPython 2.7, with minor differences in the bytecode set. Our bytecode compiler is implemented as a chain of flexible passes (tokenizer, lexer, parser, abstract syntax tree builder and bytecode generator). The latter passes are based on the compiler package from the standard library of CPython, with various improvements and bug fixes. The bytecode compiler (living under pypy/interpreter/astcompiler/) is now integrated and is translated with the rest of PyPy.

Code objects contain condensed information about their respective functions, class and module body source codes. Interpreting such code objects means instantiating and initializing a Frame class and then calling its frame.eval() method. This main entry point initialize appropriate namespaces and then interprets each bytecode instruction. Python’s standard library contains the lib-python/2.7/dis.py module which allows to inspection of the virtual machine’s bytecode instructions:

```python
>>> import dis
>>> def f(x):
...     return x + 1
>>> dis.dis(f)
2 0 LOAD_FAST 0 (x)
3 LOAD_CONST 1 (1)
6 BINARY_ADD
7 RETURN_VALUE
```

CPython and PyPy are stack-based virtual machines, i.e. they don’t have registers but instead push object to and pull objects from a stack. The bytecode interpreter is only responsible for implementing control flow and pushing and pulling black box objects to and from this value stack. The bytecode interpreter does not know how to perform operations on those black box (wrapped) objects for which it delegates to the object space. In order to implement a conditional branch in a program’s execution, however, it needs to gain minimal knowledge about a wrapped object. Thus, each object space has to offer a is_true(w_obj) operation which returns an interpreter-level boolean value.
For the understanding of the interpreter’s inner workings it is crucial to recognize the concepts of interpreter-level and application-level code. In short, interpreter-level is executed directly on the machine and invoking application-level functions leads to an bytecode interpretation indirection. However, special care must be taken regarding exceptions because application level exceptions are wrapped into OperationErrors which are thus distinguished from plain interpreter-level exceptions. See application level exceptions for some more information on OperationErrors.

The interpreter implementation offers mechanisms to allow a caller to be unaware of whether a particular function invocation leads to bytecode interpretation or is executed directly at interpreter-level. The two basic kinds of Gateway classes expose either an interpreter-level function to application-level execution (interp2app) or allow transparent invocation of application-level helpers (app2interp) at interpreter-level.

Another task of the bytecode interpreter is to care for exposing its basic code, frame, module and function objects to application-level code. Such runtime introspection and modification abilities are implemented via interpreter descriptors (also see Raymond Hettingers how-to guide for descriptors in Python, PyPy uses this model extensively).

A significant complexity lies in function argument parsing. Python as a language offers flexible ways of providing and receiving arguments for a particular function invocation. Not only does it take special care to get this right, it also presents difficulties for the annotation pass which performs a whole-program analysis on the bytecode interpreter, argument parsing and gatewaying code in order to infer the types of all values flowing across function calls.

It is for this reason that PyPy resorts to generate specialized frame classes and functions at initialization time in order to let the annotator only see rather static program flows with homogeneous name-value assignments on function invocations.

Bytecode Interpreter Implementation Classes

Frame classes

The concept of Frames is pervasive in executing programs and on virtual machines in particular. They are sometimes called execution frame because they hold crucial information regarding the execution of a Code object, which in turn is often directly related to a Python Function. Frame instances hold the following state:

- the local scope holding name-value bindings, usually implemented via a “fast scope” which is an array of wrapped objects
- a blockstack containing (nested) information regarding the control flow of a function (such as while and try constructs)
- a value stack where bytecode interpretation pulls object from and puts results on. (locals_stack_w is actually a single list containing both the local scope and the value stack.)
- a reference to the globals dictionary, containing module-level name-value bindings
- debugging information from which a current line-number and file location can be constructed for tracebacks

Moreover the Frame class itself has a number of methods which implement the actual bytecodes found in a code object. The methods of the PyFrame class are added in various files:

- the class PyFrame is defined in pypy/interpreter/pyframe.py.
- the file pypy/interpreter/pyopcode.py add support for all Python opcode.

Code Class

PyPy’s code objects contain the same information found in CPython’s code objects. They differ from Function objects in that they are only immutable representations of source code and don’t contain execution state or references to the execution environment found in Frames. Frames and Functions have references to a code object. Here is a list of Code attributes:
• **co_flags** flags if this code object has nested scopes/generators/etc.
• **co_stacksize** the maximum depth the stack can reach while executing the code
• **co_code** the actual bytecode string
• **co_argcount** number of arguments this code object expects
• **co_varnames** a tuple of all argument names pass to this code object
• **co_nlocals** number of local variables
• **co_names** a tuple of all names used in the code object
• **co_consts** a tuple of prebuilt constant objects ("literals") used in the code object
• **co_cellvars** a tuple of Cells containing values for access from nested scopes
• **co_freevars** a tuple of Cell names from “above” scopes
• **co_filename** source file this code object was compiled from
• **co_firstlineno** the first linenumber of the code object in its source file
• **co_name** name of the code object (often the function name)
• **co_lnotab** a helper table to compute the line-numbers corresponding to bytecodes

**Function and Method classes**

The PyPy **Function** class (in pypy/interpreter/function.py) represents a Python function. A **Function** carries the following main attributes:

• **func_doc** the docstring (or None)
• **func_name** the name of the function
• **func_code** the **Code** object representing the function source code
• **func_defaults** default values for the function (built at function definition time)
• **func_dict** dictionary for additional (user-defined) function attributes
• **func_globals** reference to the globals dictionary
• **func_closure** a tuple of Cell references

Functions classes also provide a **__get__** descriptor which creates a Method object holding a binding to an instance or a class. Finally, Functions and Methods both offer a **call_args()** method which executes the function given an **Arguments** class instance.

**Arguments Class**

The **Argument** class (in pypy/interpreter/argument.py) is responsible for parsing arguments passed to functions. Python has rather complex argument-passing concepts:

• positional arguments
• keyword arguments specified by name
• default values for positional arguments, defined at function definition time
• “star args” allowing a function to accept remaining positional arguments
• “star keyword args” allow a function to accept additional arbitrary name-value bindings
Moreover, a `Function` object can get bound to a class or instance in which case the first argument to the underlying function becomes the bound object. The `Arguments` provides means to allow all this argument parsing and also cares for error reporting.

**Module Class**

A `Module` instance represents execution state usually constructed from executing the module’s source file. In addition to such a module’s global `__dict__` dictionary it has the following application level attributes:

- `__doc__` the docstring of the module
- `__file__` the source filename from which this module was instantiated
- `__path__` state used for relative imports

Apart from the basic `Module` used for importing application-level files there is a more refined `MixedModule` class (see `pypy/interpreter/mixedmodule.py`) which allows to define name-value bindings both at application level and at interpreter level. See the `__builtin__` module’s `pypy/module/__builtin__/__init__.py` file for an example and the higher level chapter on Modules in the coding guide.

**Gateway classes**

A unique PyPy property is the ability to easily cross the barrier between interpreted and machine-level code (often referred to as the difference between interpreter-level and application-level). Be aware that the according code (in `pypy/interpreter/gateway.py`) for crossing the barrier in both directions is somewhat involved, mostly due to the fact that the type-inferring annotator needs to keep track of the types of objects flowing across those barriers.

**Making interpreter-level functions available at application-level**

In order to make an interpreter-level function available at application level, one invokes `pypy.interpreter.gateway.interp2app(func)`. Such a function usually takes a `space` argument and any number of positional arguments. Additionally, such functions can define an `unwrap_spec` telling the `interp2app` logic how application-level provided arguments should be unwrapped before the actual interpreter-level function is invoked. For example, `interpreter descriptors` such as the `Module.__new__` method for allocating and constructing a `Module` instance are defined with such code:

```python
Module.typedef = TypeDef("module",
    __new__ = interp2app(Module.descr_module__new__.im_func,
                          unwrap_spec=[ObjSpace, W_Root, Arguments]),
    __init__ = interp2app(Module.descr_module__init__),
    # module dictionaries are readonly attributes
    __dict__ = GetSetProperty(descr_get_dict, cls=Module),
    __doc__ = 'module(name[, doc])

    Create a module object...
    )
```

The actual `Module.descr_module__new__` interpreter-level method referenced from the `__new__` keyword argument above is defined like this:

```python
def descr_module__new__(space, w_subtype, __args__):
    module = space.allocate_instance(Module, w_subtype)
    Module.__init__(module, space, None)
    return space.wrap(module)
```
Summarizing, the `interp2app` mechanism takes care to route an application level access or call to an internal interpreter-level object appropriately to the descriptor, providing enough precision and hints to keep the type-inferring annotator happy.

**Calling into application level code from interpreter-level**

Application level code is *often preferable*. Therefore, we often like to invoke application level code from interpreter-level. This is done via the Gateway’s `app2interp` mechanism which we usually invoke at definition time in a module. It generates a hook which looks like an interpreter-level function accepting a space and an arbitrary number of arguments. When calling a function at interpreter-level the caller side does usually not need to be aware if its invoked function is run through the PyPy interpreter or if it will directly execute on the machine (after translation).

Here is an example showing how we implement the Metaclass finding algorithm of the Python language in PyPy:

```python
app = gateway.applevel(r'''
    def find_metaclass(bases, namespace, globals, builtin):
        if '__metaclass__' in namespace:
            return namespace['__metaclass__']
        elif len(bases) > 0:
            base = bases[0]
            if hasattr(base, '__class__'):
                return base.__class__
            else:
                return type(base)
        elif '__metaclass__' in globals:
            return globals['__metaclass__']
        else:
            try:
                return builtin.__metaclass__
            except AttributeError:
                return type''', filename=__file__)
    find_metaclass = app.interphook('find_metaclass')
    # ...
```

The `find_metaclass` interpreter-level hook is invoked with five arguments from the `BUILD_CLASS` opcode implementation in `pypy/interpreter/pyopcode.py`:

```python
def BUILD_CLASS(f):
    w_methodsdict = f.valuestack.pop()
    w_bases = f.valuestack.pop()
    w_name = f.valuestack.pop()
    w_metaclass = find_metaclass(f.space, w_bases,
                                  w_methodsdict, f.w_globals,
                                  f.space.wrap(f.builtin))
    w_newclass = f.space.call_function(w_metaclass, w_name, w_bases, w_methodsdict)
    f.valuestack.push(w_newclass)
```

Note that at a later point we can rewrite the `find_metaclass` implementation at interpreter-level and we would not have to modify the calling side at all.

**Introspection and Descriptors**

Python traditionally has a very far-reaching introspection model for bytecode interpreter related objects. In PyPy and in CPython read and write accesses to such objects are routed to descriptors. Of course, in CPython those are
PyPy, Release 7.2.0

implemented in C while in PyPy they are implemented in interpreter-level Python code.

All instances of a `Function`, `Code`, `Frame` or `Module` classes are also `W_Root` instances which means they can be represented at application level. These days, a PyPy object space needs to work with a basic descriptor lookup when it encounters accesses to an interpreter-level object: an object space asks a wrapped object for its type via a `getclass` method and then calls the type’s `lookup(name)` function in order to receive a descriptor function. Most of PyPy’s internal object descriptors are defined at the end of `pypy/interpreter/typedef.py`. You can use these definitions as a reference for the exact attributes of interpreter classes visible at application level.

3.5.3 Standard Interpreter Optimizations

Contents

- Standard Interpreter Optimizations
  - Introduction
  - Object Optimizations
    - Integer Optimizations
      - Caching Small Integers
      - Integers as Tagged Pointers
    - Dictionary Optimizations
      - Dict Strategies
      - Identity Dicts
      - Map Dicts
    - List Optimizations
      - Range-Lists
    - User Class Optimizations
      - Method Caching
  - Interpreter Optimizations
    - Special Bytecodes
      - `LOOKUP_METHOD` & `CALL_METHOD`
  - Overall Effects

Introduction

One of the advantages – indeed, one of the motivating goals – of the PyPy standard interpreter (compared to CPython) is that of increased flexibility and configurability.

One example of this is that we can provide several implementations of the same object (e.g. lists) without exposing any difference to application-level code. This makes it easy to provide a specialized implementation of a type that is optimized for a certain situation without disturbing the implementation for the regular case.

This document describes several such optimizations. Most of them are not enabled by default. Also, for many of these optimizations it is not clear whether they are worth it in practice for a real-world application (they sure make some
Object Optimizations

Integer Optimizations

Caching Small Integers

Similar to CPython, it is possible to enable caching of small integer objects to not have to allocate all the time when doing simple arithmetic. Every time a new integer object is created it is checked whether the integer is small enough to be retrieved from the cache.

This option is disabled by default, you can enable this feature with the –objspace-std-withprebuiltint option.

Integers as Tagged Pointers

An even more aggressive way to save memory when using integers is “small int” integer implementation. It is another integer implementation used for integers that only needs 31 bits (or 63 bits on a 64 bit machine). These integers are represented as tagged pointers by setting their lowest bits to distinguish them from normal pointers. This completely avoids the boxing step, saving time and memory.

You can enable this feature with the –objspace-std-withsmalllong option.

Dictionary Optimizations

Dict Strategies

Dict strategies are an implementation approach for dictionaries (and lists) that make it possible to use a specialized representation of the dictionary’s data, while still being able to switch back to a general representation should that become necessary later.

Dict strategies are always enabled, by default there are special strategies for dicts with just string keys, just unicode keys and just integer keys. If one of those specialized strategies is used, then dict lookup can use much faster hashing and comparison for the dict keys. There is of course also a strategy for general keys.

Identity Dicts

We also have a strategy specialized for keys that are instances of classes which compares “by identity”, which is the default unless you override __hash__, __eq__ or __cmp__. This strategy will be used only with new-style classes.

Map Dicts

Map dictionaries are a special representation used together with dict strategies. This dict strategy is used only for instance dictionaries and tries to make instance dictionaries use less memory (in fact, usually memory behaviour should be mostly like that of using __slots__).
The idea is the following: Most instances of the same class have very similar attributes, and are even adding these keys to the dictionary in the same order while \_init\_() is being executed. That means that all the dictionaries of these instances look very similar: they have the same set of keys with different values per instance. What sharing dicts do is store these common keys into a common structure object and thus save the space in the individual instance dicts: the representation of the instance dict contains only a list of values.

**List Optimizations**

**Range-Lists**

Range-lists solve the same problem that the \_xrange\_ builtin solves poorly: the problem that \_range\_ allocates memory even if the resulting list is only ever used for iterating over it. Range lists are a different implementation for lists. They are created only as a result of a call to \_range\_. As long as the resulting list is used without being mutated, the list stores only the start, stop and step of the range. Only when somebody mutates the list the actual list is created. This gives the memory and speed behaviour of \_xrange\_ and the generality of use of \_range\_, and makes \_xrange\_ essentially useless.

This feature is enabled by default as part of the --objspace-std-withliststrategies option.

**User Class Optimizations**

**Method Caching**

A method cache is introduced where the result of a method lookup is stored (which involves potentially many lookups in the base classes of a class). Entries in the method cache are stored using a hash computed from the name being looked up, the call site (i.e. the bytecode object and the current program counter), and a special “version” of the type where the lookup happens (this version is incremented every time the type or one of its base classes is changed). On subsequent lookups the cached version can be used, as long as the instance did not shadow any of its classes attributes.

This feature is enabled by default.

**Interpreter Optimizations**

**Special Bytecodes**

**LOOKUP\_METHOD & CALL\_METHOD**

An unusual feature of Python's version of object oriented programming is the concept of a “bound method”. While the concept is clean and powerful, the allocation and initialization of the object is not without its performance cost. We have implemented a pair of bytecodes that alleviate this cost.

For a given method call \_obj\_.meth(x, y), the standard bytecode looks like this:

```
LOAD_GLOBAL   obj  # push 'obj' on the stack
LOAD_ATTR     meth # read the 'meth' attribute out of 'obj'
LOAD_GLOBAL   x    # push 'x' on the stack
LOAD_GLOBAL   y    # push 'y' on the stack
CALL_FUNCTION 2  # call the 'obj.meth' object with arguments x, y
```

We improved this by keeping method lookup separated from method call, unlike some other approaches, but using the value stack as a cache instead of building a temporary object. We extended the bytecode compiler to (optionally) generate the following code for \_obj\_.meth(x, y):

```
LOAD_GLOBAL obj
LOAD_ATTR meth
LOAD_GLOBAL x
LOAD_GLOBAL y
CALL_FUNCTION 2
```
LOAD_GLOBAL  obj
LOOKUP_METHOD  meth
LOAD_GLOBAL  x
LOAD_GLOBAL  y
CALL_METHOD  2

LOOKUP_METHOD contains exactly the same attribute lookup logic as LOAD_ATTR - thus fully preserving semantics - but pushes two values onto the stack instead of one. These two values are an “inlined” version of the bound method object: the *im_func* and *im_self*, i.e. respectively the underlying Python function object and a reference to *obj*. This is only possible when the attribute actually refers to a function object from the class; when this is not the case, LOOKUP_METHOD still pushes two values, but one (*im_func*) is simply the regular result that LOAD_ATTR would have returned, and the other (*im_self*) is an interpreter-level None placeholder.

After pushing the arguments, the layout of the stack in the above example is as follows (the stack grows upwards):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y (2nd arg)</td>
<td></td>
</tr>
<tr>
<td>x (1st arg)</td>
<td></td>
</tr>
<tr>
<td>obj (im_self)</td>
<td></td>
</tr>
<tr>
<td>function object (im_func)</td>
<td></td>
</tr>
</tbody>
</table>

The CALL_METHOD *N* bytecode emulates a bound method call by inspecting the *im_self* entry in the stack below the *N* arguments: if it is not None, then it is considered to be an additional first argument in the call to the *im_func* object from the stack.

**Overall Effects**

The impact these various optimizations have on performance unsurprisingly depends on the program being run. Using the default multi-dict implementation that simply special cases string-keyed dictionaries is a clear win on all benchmarks, improving results by anything from 15-40 per cent.

Another optimization, or rather set of optimizations, that has a uniformly good effect are the two ‘method optimizations’, i.e. the method cache and the LOOKUP_METHOD and CALL_METHOD opcodes. On a heavily object-oriented benchmark (richards) they combine to give a speed-up of nearly 50%, and even on the extremely un-object-oriented pystone benchmark, the improvement is over 20%.

When building pypy, all generally useful optimizations are turned on by default unless you explicitly lower the translation optimization level with the --opt option.

### 3.5.4 PyPy Parser

**Overview**

The PyPy parser includes a tokenizer and a recursive descent parser.

**Tokenizer**

At the moment, the tokenizer is implemented as a single function (*generate_tokens* in pypy/interpreter/pyparser/pytokenizer.py) that builds a list of tokens. The tokens are then fed to the parser.

**Parser**

The parser is a simple LL(1) parser that is similar to CPython’s.
Building the Python grammar

The python grammar is built at startup from the pristine CPython grammar file (see pypy/interpreter/pyparser/metaparser.py). The grammar builder first represents the grammar as rules corresponding to a set of Nondeterministic Finite Automatons (NFAs). It then converts them to a set of Deterministic Finite Automatons (DFAs). The difference between a NFA and a DFA is that a NFA may have several possible next states for any given input while a DFA may only have one. DFAs are therefore more limiting, but far more efficient to use in parsing. Finally, the assigns the grammar builder assigns each DFA state a number and packs them into a list for the parser to use. The final product is an instance of the Grammar class in pypy/interpreter/pyparser/parser.py.

Parser implementation

The workhorse of the parser is the add_token method of the Parser class. It tries to find a transition from the current state to another state based on the token it receives as a argument. If it can’t find a transition, it checks if the current state is accepting. If it’s not, a ParseError is raised. When parsing is done without error, the parser has built a tree of Node.

Parsing Python

The glue code between the tokenizer and the parser as well as extra Python specific code is in pypy/interpreter/pyparser/pyparse.py. The parse_source method takes a string of Python code and returns the parse tree. It also detects the coding cookie if there is one and decodes the source. Note that __future__ imports are handled before the parser is invoked by manually parsing the source in pypy/interpreter/pyparser/future.py.

Compiler

The next step in generating Python bytecode is converting the parse tree into an Abstract Syntax Tree (AST).

Building AST

Python’s AST is described in pypy/interpreter/astcompiler/tools/Python.asdl. From this definition, pypy/interpreter/astcompiler/tools/asdl_py.py generates pypy/interpreter/astcompiler/ast.py, which RPython classes for the compiler as well as bindings to application level code for the AST. Some custom extensions to the AST classes are in pypy/interpreter/astcompiler/asthelpers.py.

pypy/interpreter/astcompiler/astbuilder.py is responsible for converting parse trees into AST. It walks down the parse tree building nodes as it goes. The result is a toplevel mod node.

AST Optimization

pypy/interpreter/astcompiler/optimize.py contains the AST optimizer. It does constant folding of expressions, and other simple transformations like making a load of the name “None” into a constant.

Symbol analysis

Before writing bytecode, a symbol table is built in pypy/interpreter/astcompiler/symtable.py. It determines if every name in the source is local, implicitly global (no global declaration), explicitly global (there’s a global declaration of the name in the scope), a cell (the name in used in nested scopes), or free (it’s used in a nested function).
Bytecode generation

Bytecode is emitted in `pypy/interpreter/astcompiler/codegen.py`. Each bytecode is represented temporarily by the `Instruction` class in `pypy/interpreter/astcompiler/assemble.py`. After all bytecodes have been emitted, it’s time to build the code object. Jump offsets and bytecode information like the line number table and stack depth are computed. Finally, everything is passed to a brand new `PyCode` object.

3.5.5 Configuration Options for PyPy

This directory contains documentation for the many configuration options that can be used to affect PyPy’s behaviour. There are two main classes of option, object space options and translation options.

There are two main entry points that accept options: `py.py`, which implements Python on top of another Python interpreter and accepts all the object space options:

`./py.py <objspace options>`

and the `rpython/bin/rpython` translation entry point which takes arguments of this form:

`./rpython/bin/rpython <translation options> <target>`

For the common case of `<target>` being `targetpypystandalone.py`, you can then pass the object space options after `targetpypystandalone.py`, i.e. like this:

`./rpython/bin/rpython <translation options> targetpypystandalone.py <objspace options>`

There is an overview of all command line arguments that can be passed in either position.

Many of the more interesting object space options enable optimizations, which are described in Standard Interpreter Optimizations, or allow the creation of objects that can barely be imagined in CPython, which are documented in What PyPy can do for your objects.

The following diagram gives some hints about which PyPy features work together with which other PyPy features:
PyPy, Release 7.2.0

PyPy Compatibility Matrix

Contents

- PyPy Python interpreter options
  - Internal Options
- General translation options
  - Internal Options

PyPy Python interpreter options

The following options can be used after `rpython targetpypystandalone` or as options to `py.py`.

- `--allworkingmodules`: use as many working modules as possible
- `--ext`: Comma-separated list of third-party builtin modules
- `--hash`: The hash function to use for strings: fnv from CPython 2.7 or siphash24 from CPython >= 3.4
- `--objspace-disable_call_speedhacks`: make sure that all calls go through `space.call_args`
- `--objspace-disable_entrypoints`: Disable external entry points, notably the cpyext module and cffi’s embedding mode.
- `--objspace-honor__builtins__`: Honor the `__builtins__` key of a module dictionary
• –objspace-lonepycfiles: Import pyc files with no matching py file
• –objspace-std-intshortcut: special case addition and subtraction of two integers in BINARY_ADD/BINARY_SUBTRACT and their inplace counterparts
• –objspace-std-methodcachesizeexp: 2 ** methodcachesizeexp is the size of the of the method cache
• –objspace-std-newshortcut: cache and shortcut calling __new__ from builtin types
• –objspace-std-optimized_list_getitem: special case the ‘list[integer]’ expressions
• –objspace-std-withliststrategies: enable optimized ways to store lists of primitives
• –objspace-std-withmethodcachecounter: try to cache methods and provide a counter in __pypy__. for testing purposes only.
• –objspace-std-withprebuiltnint: prebuild commonly used int objects
• –objspace-std-withsmalllong: use a version of ‘long’ in a C long long
• –objspace-std-withspecialisedtuple: use specialised tuples
• –objspace-std-withproxy: support transparent proxies
• –prebuiltnintfrom: lowest integer which is prebuilt
• –prebuiltninto: highest integer which is prebuilt
• –soabi: Tag to differentiate extension modules built for different Python interpreters
• –translationmodules: use only those modules that are needed to run translate.py on pypy
• –withmod__builtin__: use module __builtin__
• –withmod__pypy__: use module __pypy__
• –withmod__ast: use module _ast
• –withmod__cffi_backend: use module _cffi_backend
• –withmod__codecs: use module _codecs
• –withmod__collections: use module _collections
• –withmod__continuation: use module _continuation
• –withmod__cppyy: use module _cppyy
• –withmod__csv: use module _csv
• –withmod__demo: use module _demo
• –withmod__hashlib: use module _hashlib
• –withmod__io: use module _io
• –withmod__jitlog: use module _jitlog
• –withmod__locale: use module _locale
• –withmod__lsprof: use module _lsprof
• –withmod__md5: use module _md5
• –withmod__minimal_curses: use module _minimal_curses
• –withmod__multibytecodec: use module _multibytecodec
• –withmod__multiprocessing: use module _multiprocessing
• –withmod__pypyjson: use module _pypyjson
• --withmod-_random: use module _random
• --withmod-_rawffi: use module _rawffi
• --withmod-_sha: use module _sha
• --withmod-_socket: use module _socket
• --withmod-_sre: use module _sre
• --withmod-_ssl: use module _ssl
• --withmod-_testing: use module _testing
• --withmod-_vmprof: use module _vmprof
• --withmod-_warnings: use module _warnings
• --withmod-_weakref: use module _weakref
• --withmod-_winreg: use module _winreg
• --withmod-array: use module array
• --withmod-binascii: use module binascii
• --withmod-bz2: use module bz2
• --withmod-cStringIO: use module cStringIO
• --withmod-cmath: use module cmath
• --withmod-cpyext: use module cpyext
• --withmod-crypt: use module crypt
• --withmod-errno: use module errno
• --withmod-exceptions: use module exceptions
• --withmod-faulthandler: use module faulthandler
• --withmod-fcntl: use module fcntl
• --withmod-gc: use module gc
• --withmod-imp: use module imp
• --withmod-iterools: use module itertools
• --withmod-marsh: use module marshal
• --withmod-math: use module math
• --withmod-micronumpy: use module micronumpy
• --withmod-mmap: use module mmap
• --withmod-operator: use module operator
• --withmod-parser: use module parser
• --withmod-posix: use module posix
• --withmod-pwd: use module pwd
• --withmod-pyexpat: use module pyexpat
• --withmod-pypyjit: use module pypyjit
• --withmod-select: use module select
• --withmod-signal: use module signal
• --withmod-struct: use module struct
• --withmod-symbol: use module symbol
• --withmod-sys: use module sys
• --withmod-termios: use module termios
• --withmod-thread: use module thread
• --withmod-time: use module time
• --withmod-token: use module token
• --withmod-unicodedata: use module unicodedata
• --withmod-zipimport: use module zipimport
• --withmod-zlib: use module zlib

Internal Options

• --withmod-_file: use module _file
• --withmod-_pickle_support: use module _pickle_support

General translation options

The following are options of bin/rpython. They must be given before the targetxxx on the command line.

• –opt -O: set the optimization level [0, 1, size, mem, 2, 3]
• -b -backend: Backend to use for code generation
• –cc: Specify compiler to use for compiling generated C
• –clever-malloc-removal: Drives inlining to remove mallocs in a clever way
• –clever-malloc-removal-threshold: Threshold when to inline functions in clever malloc removal
• –continuation: enable single-shot continuations
• –dont-write-c-files: Make the C backend write everything to /dev/null. Useful for benchmarking, so you don’t actually involve the disk
• –dump_static_data_info: Dump static data info
• –entrypoints: Comma separated list of keys choosing secondary entrypoints
• –fork-before: (UNIX) Create restartable checkpoint before step
• –gc: Garbage Collection Strategy
• –gcmovevypeptr: Remove the typeptr from every object
• –gcrootfinder: Strategy for finding GC Roots (framework GCs only)
• –if-block-merge: Merge if ... elif chains
• –inline-threshold: Threshold when to inline functions
• –jit-backend: choose the backend for the JIT
• **--keepgoing**: Continue annotating when errors are encountered, and report them all at the end of the annotation phase

• **--listcompr**: When true, look for and special-case the sequence of operations that results from a list comprehension and attempt to pre-allocate the list

• **--lldebug**: If true, makes an lldebug build

• **--lldebug0**: If true, makes an lldebug0 build

• **--log**: Include debug prints in the translation (PYPYLOG=...) 

• **--lto**: enable link time optimization

• **--make-jobs**: Specify -j argument to make for compilation (C backend only)

• **--no__thread**: don’t use __thread for implementing TLS

• **--output**: Output file name

• **--platform**: target platform

• **--profopt**: Enable profile guided optimization. Defaults to enabling this for PyPy. For other training workloads, please specify them in profoptargs

• **--profoptargs**: Absolute path to the profile guided optimization training script + the necessary arguments of the script

• **--revdb**: Give an executable that writes a log file for reverse debugging

• **--sandbox**: Produce a fully-sandboxed executable

• **--shared**: Build as a shared library

• **--thread**: enable use of threading primitives

• **--translation-backendopt-constfold**: Constant propagation

• **--translation-backendopt-inline**: Do basic inlining and malloc removal

• **--translation-backendopt-mallocs**: Remove mallocs

• **--translation-backendopt-none**: Do not run any backend optimizations

• **--translation-backendopt-print_statistics**: Print statistics while optimizing

• **--translation-backendopt-profile_based_inline**: Use call count profiling to drive inlining, specify arguments

• **--translation-backendopt-profile_based_inline_threshold**: Threshold when to inline functions for profile based inlining

• **--translation-backendopt-really_remove_asserts**: Really remove operations that look like ‘raise AssertionError’, without relying on the C compiler

• **--translation-backendopt-remove_asserts**: Remove operations that look like ‘raise AssertionError’, which lets the C optimizer remove the asserts

• **--translation-backendopt-stack_optimization**: Tranform graphs in SSI form into graphs tailored for stack based virtual machines (only for backends that support it)

• **--translation-backendopt-storesink**: Perform store sinking

• **--translation-icon**: Path to the (Windows) icon to use for the executable

• **--translation-jit**: generate a JIT

• **--translation-jit_opencoder_model**: the model limits the maximal length of traces. Use big if you want to go bigger than the default
• **--translation-jit_profiler**: integrate profiler support into the JIT
• **--translation-libname**: Windows: name and possibly location of the lib file to create
• **--translation-rweakref**: The backend supports RPython-level weakrefs
• **--translation-split_gc_address_space**: Ensure full separation of GC and non-GC pointers
• **--translation-taggedpointers**: When true, enable the use of tagged pointers. If false, use normal boxing
• **--translation-withsmallfuncsets**: Represent groups of less functions than this as indices into an array
• **--verbose**: Print extra information

**Internal Options**

• **--clever-malloc-removal-heuristic**: Dotted name of an heuristic function for inlining in clever malloc removal
• **--inline-heuristic**: Dotted name of an heuristic function for inlining
• **--translation-backendopt-profile_based_inline_heuristic**: Dotted name of an heuristic function for profile based inlining

**translation**

**translation.continuation**

• **name**: continuation
• **description**: enable single-shot continuations
• **command-line**: --continuation
• **command-line for negation**: --no-continuation
• **option type**: boolean option
• **default**: False
• **requirements**:
  - **translation.type_system** must be set to ‘lltype’

Enable the use of a stackless-like primitive called “stacklet”. In PyPy, this is exposed at app-level by the “continuation” module.

**translation.type_system**

• **name**: type_system
• **description**: Type system to use when RTyping
• **option type**: choice option
• **possible values**:
  - lltype
• **default**: lltype

Which type system to use when rtyping. This option should not be set explicitly.
translation.backend

- **name**: backend
- **description**: Backend to use for code generation
- **command-line**: `-b --backend`
- **option type**: choice option
- **possible values**:
  - c
- **default**: c
- **requirements**:
  - value 'c' requires:
    - translation.type_system to be set to 'lltype'

Which backend to use when translating, see translation documentation.

translation.shared

- **name**: shared
- **description**: Build as a shared library
- **command-line**: `--shared`
- **command-line for negation**: `--no-shared`
- **option type**: boolean option
- **default**: False

Build pypy as a shared library or a DLL, with a small executable to run it. This is necessary on Windows to expose the C API provided by the cpyext module.

translation.log

- **name**: log
- **description**: Include debug prints in the translation (PYPYLOG=...)
- **command-line**: `--log`
- **command-line for negation**: `--no-log`
- **option type**: boolean option
- **default**: True

Include debug prints in the translation.

These must be enabled by setting the PYPYLOG environment variable. The exact set of features supported by PYPYLOG is described in rpython/translator/c/src/debug_print.h.
translation.gc

- **name**: gc
- **description**: Garbage Collection Strategy
- **command-line**: –gc
- **option type**: choice option
- **possible values**:
  - boehm
  - ref
  - semispace
  - statistics
  - generation
  - hybrid
  - minimark
  - incminimark
  - none
- **default**: ref
- **requirements**:
  - value ‘boehm’ requires:
    * translation.continuation to be set to ‘False’
    * translation.gctransformer to be set to ‘boehm’
  - value ‘ref’ requires:
    * translation.rweakref to be set to ‘False’
    * translation.gctransformer to be set to ‘ref’
  - value ‘semispace’ requires:
    * translation.gctransformer to be set to ‘framework’
  - value ‘statistics’ requires:
    * translation.gctransformer to be set to ‘framework’
  - value ‘generation’ requires:
    * translation.gctransformer to be set to ‘framework’
  - value ‘hybrid’ requires:
    * translation.gctransformer to be set to ‘framework’
  - value ‘minimark’ requires:
    * translation.gctransformer to be set to ‘framework’
  - value ‘incminimark’ requires:
    * translation.gctransformer to be set to ‘framework’
  - value ‘none’ requires:
* translation.rweakref to be set to ‘False’
* translation.gctransformer to be set to ‘none’

Choose the Garbage Collector used by the translated program. The recommended default is “incminimark”.

- “ref”: reference counting. Takes very long to translate and the result is slow. Used only for tests. Don’t use it for real RPython programs.
- “none”: no GC. Leaks everything. Don’t use it for real RPython programs: the rate of leaking is immense.
- “semispacer”: a copying semi-space GC.
- “generation”: a generational GC using the semi-space GC for the older generation.
- “hybrid”: a hybrid collector of “generation” together with a mark-n-sweep old space
- “boehm”: use the Boehm conservative GC.
- “minimark”: a generational mark-n-sweep collector with good performance. Includes page marking for large arrays.
- “incminimark”: like minimark, but adds incremental major collections. Seems to come with no performance drawback over “minimark”, so it is the default. A few recent features of PyPy (like cpyext) are only working with this GC.

**translation.gctransformer**

- **name**: gctransformer
- **description**: GC transformer that is used - internal
- **option type**: choice option
- **possible values**:
  - boehm
  - ref
  - framework
  - none
- **default**: ref
- **requirements**:
  - value ‘boehm’ requires:
    * translation.gcrootfinder to be set to ‘n/a’
    * translation.gcremovetypeptr to be set to ‘False’
  - value ‘ref’ requires:
    * translation.gcrootfinder to be set to ‘n/a’
    * translation.gcremovetypeptr to be set to ‘False’
  - value ‘none’ requires:
    * translation.gcrootfinder to be set to ‘n/a’
    * translation.gcremovetypeptr to be set to ‘False’
translation.gcremovetypeptr

- **name**: gcremovetypeptr
- **description**: Remove the typeptr from every object
- **command-line**: –gcremovetypeptr
- **command-line for negation**: –no-gcremovetypeptr
- **option type**: boolean option
- **default**: True

If set, save one word in every object. Framework GC only.

translation.gcrootfinder

- **name**: gcrootfinder
- **description**: Strategy for finding GC Roots (framework GCs only)
- **command-line**: –gcrootfinder
- **option type**: choice option
- **possible values**:
  - n/a
  - shadowstack
  - asmgcc
- **default**: shadowstack
- **requirements**:
  - value ‘shadowstack’ requires:
    - translation.gctransformer to be set to ‘framework’
  - value ‘asmgcc’ requires:
    - translation.gctransformer to be set to ‘framework’
    - translation.backend to be set to ‘c’

Choose the method used to find the roots in the GC. This only applies to our framework GCs. You have a choice of two alternatives:

- **--gcrootfinder=shadowstack**: use a so-called “shadow stack”, which is an explicitly maintained custom stack of root pointers. This is the most portable solution.

- **--gcrootfinder=asmgcc**: use assembler hackery to find the roots directly from the normal stack. This is a bit faster, but platform specific. It works so far with GCC or MSVC, on i386 and x86-64. It is tested only on Linux so other platforms (as well as MSVC) may need various fixes before they can be used. Note asmgcc will be deprecated at some future date, and does not work with clang.
translation.thread

- **name:** thread
- **description:** enable use of threading primitives
- **command-line:** –thread
- **command-line for negation:** –no-thread
- **option type:** boolean option
- **default:** False

Enable threading. The only target where this has visible effect is PyPy (this also enables the thread module then).

translation.sandbox

- **name:** sandbox
- **description:** Produce a fully-sandboxed executable
- **command-line:** –sandbox
- **command-line for negation:** –no-sandbox
- **option type:** boolean option
- **default:** False
- **requirements:**
  - translation.thread must be set to ‘False’
- **suggestions:**
  - translation.gc should be set to ‘generation’
  - translation.gcrootfinder should be set to ‘shadowstack’

Generate a special fully-sandboxed executable.

The fully-sandboxed executable cannot be run directly, but only as a subprocess of an outer “controlling” process. The sandboxed process is “safe” in the sense that it doesn’t do any library or system call - instead, whenever it would like to perform such an operation, it marshals the operation name and the arguments to its stdout and it waits for the marshalled result on its stdin. This controller process must handle these operation requests, in any way it likes, allowing full virtualization.

For examples of controller processes, see pypy/translator/sandbox/interact.py and pypy/translator/sandbox/pypy_interact.py.

translation.rweakref

- **name:** rweakref
- **description:** The backend supports RPython-level weakrefs
- **command-line:** –translation-rweakref
- **command-line for negation:** –no-translation-rweakref
- **option type:** boolean option
- **default:** True
This indicates if the backend and GC policy support RPython-level weakrefs. Can be tested in an RPython program to select between two implementation strategies.

**translation.jit**

- **name:** jit
- **description:** generate a JIT
- **command-line:** –translation-jit
- **command-line for negation:** –no-translation-jit
- **option type:** boolean option
- **default:** False
- **suggestions:**
  - translation.gc should be set to ‘incminimark’
  - translation.gcrootfinder should be set to ‘shadowstack’
  - translation.list_comprehension_operations should be set to ‘True’

Enable the JIT generator, for targets that have JIT support. Experimental so far.

**translation.jit_backend**

- **name:** jit_backend
- **description:** choose the backend for the JIT
- **command-line:** –jit-backend
- **option type:** choice option
- **possible values:**
  - auto
  - x86
  - x86-without-sse2
  - arm
- **default:** auto

Choose the backend to use for the JIT. By default, this is the best backend for the current platform.

**translation.jit_profiler**

- **name:** jit_profiler
- **description:** integrate profiler support into the JIT
- **command-line:** –translation-jit_profiler
- **option type:** choice option
- **possible values:**
PyPy, Release 7.2.0

– off
– oprofile

• default: off
Integrate profiler support into the JIT

**translation.jit_opencoder_model**

• name: jit_opencoder_model
• description: the model limits the maximal length of traces. Use big if you want to go bigger than the default
• command-line: --translation-jit_opencoder_model
• option type: choice option
• possible values:
  – big
  – normal
• default: normal

**translation.check_str_without_nul**

• name: check_str_without_nul
• description: Forbid NUL chars in strings in some external function calls
• option type: boolean option
• default: False
If turned on, the annotator will keep track of which strings can potentially contain NUL characters, and complain if one such string is passed to some external functions — e.g. if it is used as a filename in os.open(). Defaults to False because it is usually more pain than benefit, but turned on by targetpypystandalone.

**translation.verbose**

• name: verbose
• description: Print extra information
• command-line: --verbose
• command-line for negation: --no-verbose
• option type: boolean option
• default: False
Print some more information during translation.
translation.cc

- **name**: cc
- **description**: Specify compiler to use for compiling generated C
- **command-line**: –cc
- **option type**: string option

Specify which C compiler to use.

translation.profopt

- **name**: profopt
- **description**: Enable profile guided optimization. Defaults to enabling this for PyPy. For other training workloads, please specify them in profoptargs
- **command-line**: –profopt
- **command-line for negation**: –no-profopt
- **option type**: boolean option
- **default**: False

Use GCCs profile-guided optimizations. This option specifies the arguments with which to call pypy-c (and in general the translated RPython program) to gather profile data. Example for pypy-c: “–c ‘from richards import main;main(); from test import pystone; pystone.main()’”

NOTE: be aware of what this does in JIT-enabled executables. What it does is instrument and later optimize the C code that happens to run in the example you specify, ignoring any execution of the JIT-generated assembler. That means that you have to choose the example wisely. If it is something that will just generate assembler and stay there, there is little value. If it is something that exercises heavily library routines that are anyway written in C, then it will optimize that. Most interesting would be something that causes a lot of JIT-compilation, like running a medium-sized test suite several times in a row, in order to optimize the warm-up in general.

translation.profoptargs

- **name**: profoptargs
- **description**: Absolute path to the profile guided optimization training script + the necessary arguments of the script
- **command-line**: –profoptargs
- **option type**: string option

translation.instrument

- **name**: instrument
- **description**: internal: turn instrumentation on
- **option type**: boolean option
- **default**: False

Internal option.
translation.countmallocs

- **name**: countmallocs
- **description**: Count mallocs and frees
- **option type**: boolean option
- **default**: False

Internal; used by some of the C backend tests to check that the number of allocations matches the number of frees.

translation.fork_before

- **name**: fork_before
- **description**: (UNIX) Create restartable checkpoint before step
- **command-line**: –fork-before
- **option type**: choice option
- **possible values**:
  - annotate
  - rtype
  - backendopt
  - database
  - source
  - pyjitpl

This is an option mostly useful when working on the PyPy toolchain. If you use it, translation will fork before the specified phase. If the translation crashes after that fork, you can fix the bug in the toolchain, and continue translation at the fork-point.

translation.dont_write_c_files

- **name**: dont_write_c_files
- **description**: Make the C backend write everyting to /dev/null. Useful for benchmarking, so you don’t actually involve the disk
- **command-line**: –dont-write-c-files
- **command-line for negation**: –no-dont-write-c-files
- **option type**: boolean option
- **default**: False

write the generated C files to /dev/null instead of to the disk. Useful if you want to use translation as a benchmark and don’t want to access the disk.
translation.instrumentctl

- **name**: instrumentctl
- **description**: internal
- **option type**: arbitrary option (mostly internal)

Internal option.

translation.output

- **name**: output
- **description**: Output file name
- **command-line**: --output
- **option type**: string option

Specify file name that the produced executable gets.

translation.secondaryentrypoints

- **name**: secondaryentrypoints
- **description**: Comma separated list of keys choosing secondary entrypoints
- **command-line**: --entrypoints
- **option type**: string option
- **default**: main

Enable secondary entrypoints support list. Needed for cpyext module.

translation.dump_static_data_info

- **name**: dump_static_data_info
- **description**: Dump static data info
- **command-line**: --dump_static_data_info
- **command-line for negation**: --no-dump_static_data_info
- **option type**: boolean option
- **default**: False
- **requirements**: translation.backend must be set to ‘c’

Dump information about static prebuilt constants, to the file TARGETNAME.staticdata.info in the /tmp/usession-... directory. This file can be later inspected using the script bin/reportstaticdata.py.
translation.no__thread

- **name**: no__thread
- **description**: don’t use __thread for implementing TLS
- **command-line**: –no__thread
- **option type**: boolean option
- **default**: False

Don’t use gcc __thread attribute for fast thread local storage implementation. Increases the chance that moving the resulting executable to another same processor Linux machine will work.

translation.make_jobs

- **name**: make_jobs
- **description**: Specify -j argument to make for compilation (C backend only)
- **command-line**: –make-jobs
- **option type**: integer option
- **default**: 3

Specify number of make jobs for make command.

translation.list_comprehension_operations

- **name**: list_comprehension_operations
- **description**: When true, look for and special-case the sequence of operations that results from a list comprehension and attempt to pre-allocate the list
- **command-line**: –listcompr
- **command-line for negation**: –no-listcompr
- **option type**: boolean option
- **default**: False

Experimental optimization for list comprehensions in RPython.

translation.withsmallfuncsets

- **name**: withsmallfuncsets
- **description**: Represent groups of less functions than this as indices into an array
- **command-line**: –translation-withsmallfuncsets
- **option type**: integer option
- **default**: 0

Represent function sets smaller than this option’s value as an integer instead of a function pointer. A call is then done via a switch on that integer, which allows inlining etc. Small numbers for this can speed up PyPy (try 5).
**translation.taggedpointers**

- **name:** taggedpointers
- **description:** When true, enable the use of tagged pointers. If false, use normal boxing
- **command-line:** –translation-taggedpointers
- **command-line for negation:** –no-translation-taggedpointers
- **option type:** boolean option
- **default:** False

Enable tagged pointers. This option is mostly useful for the Smalltalk and Prolog interpreters. For the Python interpreter the option –objspace-std-withsmalllong should be used.

**translation.keepgoing**

- **name:** keepgoing
- **description:** Continue annotating when errors are encountered, and report them all at the end of the annotation phase
- **command-line:** –keepgoing
- **command-line for negation:** –no-keepgoing
- **option type:** boolean option
- **default:** False

**translation.lldebug**

- **name:** lldebug
- **description:** If true, makes an lldebug build
- **command-line:** –lldebug
- **command-line for negation:** –no-lldebug
- **option type:** boolean option
- **default:** False

Run make lldebug when source is ready

**translation.lldebug0**

- **name:** lldebug0
- **description:** If true, makes an lldebug0 build
- **command-line:** –lldebug0
- **command-line for negation:** –no-lldebug0
- **option type:** boolean option
- **default:** False
Like lldebug, but in addition compile C files with -O0

**translation.lto**

- **name:** lto
- **description:** enable link time optimization
- **command-line:** –lto
- **command-line for negation:** –no-lto
- **option type:** boolean option
- **default:** False
- **requirements:**
  - translation.gcrootfinder must be set to ‘shadowstack’

**translation.icon**

- **name:** icon
- **description:** Path to the (Windows) icon to use for the executable
- **command-line:** –translation-icon
- **option type:** string option

**translation.libname**

- **name:** libname
- **description:** Windows: name and possibly location of the lib file to create
- **command-line:** –translation-libname
- **option type:** string option

**translation.backendopt**

**translation.backendopt.inline**

- **name:** inline
- **description:** Do basic inlining and malloc removal
- **command-line:** –translation-backendopt-inline
- **command-line for negation:** –no-translation-backendopt-inline
- **option type:** boolean option
- **default:** True
Inline flowgraphs based on an heuristic; the default one considers essentially the a weight for the flowgraph based on
the number of low-level operations in them (see \(--\text{inline-threshold}\)).

Some amount of inlining in order to have RPython builtin type helpers inlined is needed for malloc removal
(\(--\text{translation-backendopt-mallocs}\) to be effective.

This optimization is used by default.

**translation.backendopt.inline_threshold**

- **name:** inline_threshold
- **description:** Threshold when to inline functions
- **command-line:** \(--\text{inline-threshold}\)
- **option type:** float option
- **default:** 32.4

Weight threshold used to decide whether to inline flowgraphs. This is for basic inlining (\(--\text{translation-backendopt-inline}\)).

**translation.backendopt.inline_heuristic**

- **name:** inline_heuristic
- **description:** Dotted name of an heuristic function for inlining
- **command-line:** \(--\text{inline-heuristic}\)
- **option type:** string option
- **default:** rpython.translator.backendopt.inline.inlining_heuristic

Internal option. Switch to a different weight heuristic for inlining. This is for basic inlining (\(--\text{translation-backendopt-inline}\)).

**translation.backendopt.print_statistics**

- **name:** print_statistics
- **description:** Print statistics while optimizing
- **command-line:** \(--\text{translation-backendopt-print_statistics}\)
- **command-line for negation:** \(--\text{no-translation-backendopt-print_statistics}\)
- **option type:** boolean option
- **default:** False

Debugging option. Print statics about the forest of flowgraphs as they go through the various backend optimizations.

**translation.backendopt.merge_if_blocks**

- **name:** merge_if_blocks
- **description:** Merge if . . . elif chains
• **command-line**: –if-block-merge
• **command-line for negation**: –no-if-block-merge
• **option type**: boolean option
• **default**: True

This optimization converts parts of flow graphs that result from chains of ifs and elifs like this into merged blocks. By default flow graphing this kind of code:

```python
if x == 0:
    f()
elif x == 1:
    g()
elif x == 4:
    h()
else:
    j()
```

will result in a chain of blocks with two exits, somewhat like this:

![Diagram of Python flow graph](image1)

(reflecting how Python would interpret this code). Running this optimization will transform the block structure to contain a single “choice block” with four exits:

![Diagram of optimized flow graph](image2)

This can then be turned into a switch by the C backend, allowing the C compiler to produce more efficient code.
translation.backendopt.mallocs

- **name**: mallocs
- **description**: Remove mallocs
- **command-line**: –translation-backendopt-mallocs
- **command-line for negation**: –no-translation-backendopt-mallocs
- **option type**: boolean option
- **default**: True

This optimization enables “malloc removal”, which “explodes” allocations of structures which do not escape from the function they are allocated in into one or more additional local variables.

An example. Consider this rather unlikely seeming code:

```python
class C:
    pass
def f(y):
    c = C()
    c.x = y
    return c.x
```

Malloc removal will spot that the C object can never leave f and replace the above with code like this:

```python
def f(y):
    __c__.x = y
    return __c__.x
```

It is rare for code to be directly written in a way that allows this optimization to be useful, but inlining often results in opportunities for its use (and indeed, this is one of the main reasons PyPy does its own inlining rather than relying on the C compilers).

For much more information about this and other optimizations you can read section 4.1 of the technical report on “Massive Parallelism and Translation Aspects” which you can find on the Technical reports page.

translation.backendopt.constfold

- **name**: constfold
- **description**: Constant propagation
- **command-line**: –translation-backendopt-constfold
- **command-line for negation**: –no-translation-backendopt-constfold
- **option type**: boolean option
- **default**: True

Do constant folding of operations and constant propagation on flowgraphs.

translation.backendopt.profile_based_inline

- **name**: profile_based_inline
- **description**: Use call count profiling to drive inlining, specify arguments
• **command-line**: –translation-backendopt-profile_based_inline
  • **option type**: string option
  Inline flowgraphs only for call-sites for which there was a minimal number of calls during an instrumented run of the program. Callee flowgraphs are considered candidates based on a weight heuristic like for basic inlining. (see –translation-backendopt-inline, –translation-backendopt-profile_based_inline_threshold).
  The option takes as value a string which is the arguments to pass to the program for the instrumented run. This optimization is not used by default.

**translation.backendopt.profile_based_inline_threshold**

  • **name**: profile_based_inline_threshold
  • **description**: Threshold when to inline functions for profile based inlining
  • **command-line**: –translation-backendopt-profile_based_inline_threshold
  • **option type**: float option
  • **default**: 32.4
  Weight threshold used to decide whether to inline flowgraphs. This is for profile-based inlining (–translation-backendopt-profile_based_inline).

**translation.backendopt.profile_based_inline_heuristic**

  • **name**: profile_based_inline_heuristic
  • **description**: Dotted name of an heuristic function for profile based inlining
  • **command-line**: –translation-backendopt-profile_based_inline_heuristic
  • **option type**: string option
  • **default**: rpython.translator.backendopt.inline.inlining_heuristic
  Internal option. Switch to a different weight heuristic for inlining. This is for profile-based inlining (–translation-backendopt-profile_based_inline).

**translation.backendopt.clever_malloc_removal**

  • **name**: clever_malloc_removal
  • **description**: Drives inlining to remove mallocs in a clever way
  • **command-line**: –clever-malloc-removal
  • **command-line for negation**: –no-clever-malloc-removal
  • **option type**: boolean option
  • **default**: False
  Try to inline flowgraphs based on whether doing so would enable malloc removal (–translation-backendopt-mallocs.) by eliminating calls that result in escaping. This is an experimental optimization, also right now some eager inlining is necessary for helpers doing malloc itself to be inlined first for this to be effective. This option enable also an extra subsequent malloc removal phase.
Callee flowgraphs are considered candidates based on a weight heuristic like for basic inlining. (see \texttt{--translation-backendopt-inline}, \texttt{--clever-malloc-removal-threshold}).

\textbf{translation.backendopt.clever_malloc_removal_threshold}

\begin{itemize}
\item \textbf{name:} clever_malloc_removal_threshold
\item \textbf{description:} Threshold when to inline functions in clever malloc removal
\item \textbf{command-line:} \texttt{--clever-malloc-removal-threshold}
\item \textbf{option type:} float option
\item \textbf{default:} 32.4
\end{itemize}

Weight threshold used to decide whether to inline flowgraphs. This is for clever malloc removal (\texttt{--clever-malloc-removal}).

\textbf{translation.backendopt.clever_malloc_removal_heuristic}

\begin{itemize}
\item \textbf{name:} clever_malloc_removal_heuristic
\item \textbf{description:} Dotted name of an heuristic function for inlining in clever malloc removal
\item \textbf{command-line:} \texttt{--clever-malloc-removal-heuristic}
\item \textbf{option type:} string option
\item \textbf{default:} rpython.translator.backendopt.inline.inlining_heuristic
\end{itemize}

Internal option. Switch to a different weight heuristic for inlining. This is for clever malloc removal (\texttt{--clever-malloc-removal}).

\textbf{translation.backendopt.remove_asserts}

\begin{itemize}
\item \textbf{name:} remove_asserts
\item \textbf{description:} Remove operations that look like ‘raise AssertionError’, which lets the C optimizer remove the asserts
\item \textbf{command-line:} \texttt{--translation-backendopt-remove_asserts}
\item \textbf{command-line for negation:} \texttt{--no-translation-backendopt-remove_asserts}
\item \textbf{option type:} boolean option
\item \textbf{default:} False
\end{itemize}

Remove raising of assertions from the flowgraphs, which might give small speedups.

\textbf{translation.backendopt.really_remove_asserts}

\begin{itemize}
\item \textbf{name:} really_remove_asserts
\item \textbf{description:} Really remove operations that look like ‘raise AssertionError’, without relying on the C compiler
\item \textbf{command-line:} \texttt{--translation-backendopt-really_remove_asserts}
\item \textbf{command-line for negation:} \texttt{--no-translation-backendopt-really_remove_asserts}
\end{itemize}
• **option type**: boolean option
• **default**: False

**translation.backendopt.stack_optimization**

• **name**: stack_optimization
• **description**: Transform graphs in SSI form into graphs tailored for stack based virtual machines (only for back-ends that support it)
• **command-line**: –translation-backendopt-stack_optimization
• **command-line for negation**: –no-translation-backendopt-stack_optimization
• **option type**: boolean option
• **default**: True

Enable the optimized code generation for stack based machine, if the backend support it

**translation.backendopt.storesink**

• **name**: storesink
• **description**: Perform store sinking
• **command-line**: –translation-backendopt-storesink
• **command-line for negation**: –no-translation-backendopt-storesink
• **option type**: boolean option
• **default**: True

Store sinking optimization. On by default.

**translation.backendopt.replace_we_are_jitted**

• **name**: replace_we_are_jitted
• **description**: Replace we_are_jitted() calls by False
• **option type**: boolean option
• **default**: False

**translation.backendopt.none**

• **name**: none
• **description**: Do not run any backend optimizations
• **command-line**: –translation-backendopt-none
• **command-line for negation**: –no-translation-backendopt-none
• **option type**: boolean option
• **requirements**: None
– translation.backendopt.inline must be set to ‘False’
– translation.backendopt.inline_threshold must be set to ‘0’
– translation.backendopt.merge_if_blocks must be set to ‘False’
– translation.backendopt.mallocs must be set to ‘False’
– translation.backendopt.constfold must be set to ‘False’

Do not run any backend optimizations.

- **name**: backendopt
- **description**: Backend Optimization Options

This group contains options about various backend optimization passes. Most of them are described in the EU report about optimization

**translation.platform**

- **name**: platform
- **description**: target platform
- **command-line**: –platform
- **option type**: choice option
- **possible values**:
  - host
  - host
  - arm
- **default**: host

select the target platform, in case of cross-compilation

**translation.split_gc_address_space**

- **name**: split_gc_address_space
- **description**: Ensure full separation of GC and non-GC pointers
- **command-line**: –translation-split_gc_address_space
- **command-line for negation**: –no-translation-split_gc_address_space
- **option type**: boolean option
- **default**: False

**translation.reverse_debugger**

- **name**: reverse_debugger
- **description**: Give an executable that writes a log file for reverse debugging
- **command-line**: –revdb
- **command-line for negation**: –no-revdb
- **option type**: boolean option
- **default**: False
- **requirements**:
  - `translation.split_gc_address_space` must be set to ‘True’
  - `translation.jit` must be set to ‘False’
  - `translation.gc` must be set to ‘boehm’
  - `translation.continuation` must be set to ‘False’
- **name**: translation
- **description**: Translation Options

**objspace**

**objspace.usemodules**

**objspace.usemodules.faulthandler**

- **name**: faulthandler
- **description**: use module faulthandler
- **command-line**: –withmod-faulthandler
- **command-line for negation**: –withoutmod-faulthandler
- **option type**: boolean option
- **default**: False
- **requirements**:
  - `objspace.usemodules._vmprof` must be set to ‘True’

**objspace.usemodules.binascii**

- **name**: binascii
- **description**: use module binascii
- **command-line**: –withmod-binascii
- **command-line for negation**: –withoutmod-binascii
- **option type**: boolean option
- **default**: False

Use the RPython ‘binascii’ module.
**objspace.usemodules._jitlog**

- **name:** _jitlog
- **description:** use module _jitlog
- **command-line:** –withmod-_jitlog
- **command-line for negation:** –withoutmod-_jitlog
- **option type:** boolean option
- **default:** False

**objspace.usemodules._md5**

- **name:** _md5
- **description:** use module _md5
- **command-line:** –withmod-_md5
- **command-line for negation:** –withoutmod-_md5
- **option type:** boolean option
- **default:** False

Use the built-in ‘_md5’ module. This module is expected to be working and is included by default. There is also a pure Python version in lib_pypy which is used if the built-in is disabled, but it is several orders of magnitude slower.

**objspace.usemodules.itertools**

- **name:** itertools
- **description:** use module itertools
- **command-line:** –withmod-itertools
- **option type:** boolean option
- **default:** True

Use the interp-level ‘itertools’ module. If not included, a slower app-level version of itertools is used.

**objspace.usemodules._minimal_curses**

- **name:** _minimal_curses
- **description:** use module _minimal_curses
- **command-line:** –withmod-_minimal_curses
- **command-line for negation:** –withoutmod-_minimal_curses
- **option type:** boolean option
- **default:** False

Use the ‘_curses’ module. This module is just a stub. It only implements a few functions.
objspace.usemodules.errno

- **name**: errno
- **description**: use module errno
- **command-line**: –withmod-errno
- **command-line for negation**: –withoutmod-errno
- **option type**: boolean option
- **default**: True

Use the ‘errno’ module. This module is expected to be working and is included by default.

objspace.usemodules.sys

- **name**: sys
- **description**: use module sys
- **command-line**: –withmod-sys
- **option type**: boolean option
- **default**: True

Use the ‘sys’ module. This module is essential, included by default and should not be removed.

objspace.usemodules._collections

- **name**: _collections
- **description**: use module _collections
- **command-line**: –withmod-_collections
- **command-line for negation**: –withoutmod-_collections
- **option type**: boolean option
- **default**: False

Use the ‘_collections’ module. Used by the ‘collections’ standard lib module. This module is expected to be working and is included by default.

objspace.usemodules._warnings

- **name**: _warnings
- **description**: use module _warnings
- **command-line**: –withmod-_warnings
- **option type**: boolean option
- **default**: True

Use the ‘_warning’ module. This module is expected to be working and is included by default.
**objspace.usemodules.parser**

- **name:** parser
- **description:** use module parser
- **command-line:** –withmod-parser
- **command-line for negation:** –withoutmod-parser
- **option type:** boolean option
- **default:** True

Use the ‘parser’ module. This is PyPy implementation of the standard library ‘parser’ module (e.g. if this option is enabled and you say `import parser` you get this module). It is enabled by default.

**objspace.usemodules._multibytecodec**

- **name:** _multibytecodec
- **description:** use module _multibytecodec
- **command-line:** –withmod-_multibytecodec
- **command-line for negation:** –withoutmod-_multibytecodec
- **option type:** boolean option
- **default:** False


**objspace.usemodules._codecs**

- **name:** _codecs
- **description:** use module _codecs
- **command-line:** –withmod-_codecs
- **command-line for negation:** –withoutmod-_codecs
- **option type:** boolean option
- **default:** True

Use the ‘_codecs’ module. Used by the ‘codecs’ standard lib module. This module is expected to be working and is included by default.

**objspace.usemodules.token**

- **name:** token
- **description:** use module token
- **command-line:** –withmod-token
• command-line for negation: –withoutmod-token

• option type: boolean option

• default: True

Use the ‘token’ module. This module is expected to be working and is included by default.

objc.space.usemodules._io

• name: _io

• description: use module _io

• command-line: –withmod-_io

• command-line for negation: –withoutmod-_io

• option type: boolean option

• default: True

Use the ‘_io module. Used by the ‘io’ standard lib module. This module is expected to be working and is included by default.

objc.space.usemodules._socket

• name: _socket

• description: use module _socket

• command-line: –withmod-_socket

• command-line for negation: –withoutmod-_socket

• option type: boolean option

• default: False

Use the ‘_socket’ module.

This is our implementation of ‘_socket’, the Python builtin module exposing socket primitives, which is wrapped and used by the standard library ‘socket.py’ module. It is based on rffi.

objc.space.usemodules.pyexpat

• name: pyexpat

• description: use module pyexpat

• command-line: –withmod-pyexpat

• command-line for negation: –withoutmod-pyexpat

• option type: boolean option

• default: False

Use the pyexpat module, written in RPython.
**objspace.usemodules._multiprocessing**

- **name**: _multiprocessing
- **description**: use module _multiprocessing
- **command-line**: –withmod-_multiprocessing
- **command-line for negation**: –withoutmod-_multiprocessing
- **option type**: boolean option
- **default**: False
- **requirements**:
  - objspace.usemodules.time must be set to ‘True’
  - objspace.usemodules.thread must be set to ‘True’

Use the ‘_multiprocessing’ module. Used by the ‘multiprocessing’ standard lib module. This module is expected to be working and is included by default.

**objspace.usemodules.array**

- **name**: array
- **description**: use module array
- **command-line**: –withmod-array
- **command-line for negation**: –withoutmod-array
- **option type**: boolean option
- **default**: False

Use interpreter-level version of array module (on by default).

**objspace.usemodules.gc**

- **name**: gc
- **description**: use module gc
- **command-line**: –withmod-gc
- **command-line for negation**: –withoutmod-gc
- **option type**: boolean option
- **default**: True

Use the ‘gc’ module. This module is expected to be working and is included by default. Note that since the gc module is highly implementation specific, it contains only the collect function in PyPy, which forces a collection when compiled with the framework or with Boehm.
**objc_modules._weakref**

- **name**: _weakref
- **description**: use module _weakref
- **command-line**: --withmod-_weakref
- **command-line for negation**: --withoutmod-_weakref
- **option type**: boolean option
- **default**: True

Use the '_weakref' module, necessary for the standard lib 'weakref' module. PyPy’s weakref implementation is not completely stable yet. The first difference to CPython is that weak references only go away after the next garbage collection, not immediately. The other problem seems to be that under certain circumstances (that we have not determined) weak references keep the object alive.

**objc_modulesfcntl**

- **name**: fcntl
- **description**: use module fcntl
- **command-line**: --withmod-fcntl
- **command-line for negation**: --withoutmod-fcntl
- **option type**: boolean option
- **default**: False

Use the ‘fcntl’ module. This module is expected to be fully working.

**objc_modules.cStringIO**

- **name**: cStringIO
- **description**: use module cStringIO
- **command-line**: --withmod-cStringIO
- **command-line for negation**: --withoutmod-cStringIO
- **option type**: boolean option
- **default**: False

Use the built-in cStringIO module.

If not enabled, importing cStringIO gives you the app-level implementation from the standard library StringIO module.

**objc_modules.select**

- **name**: select
- **description**: use module select
- **command-line**: --withmod-select
- **command-line for negation**: --withoutmod-select
• **option type**: boolean option
  • **default**: False

Use the ‘select’ module. This module is expected to be fully working.

**objspace.usemodules._file**

  • **name**: _file
  • **description**: use module _file
  • **command-line**: –withmod-_file
  • **option type**: boolean option
  • **default**: True

Use the ‘_file’ module. It is an internal module that contains helper functionality for the builtin `file` type.

**objspace.usemodules.math**

  • **name**: math
  • **description**: use module math
  • **command-line**: –withmod-math
  • **command-line for negation**: –withoutmod-math
  • **option type**: boolean option
  • **default**: True

Use the ‘math’ module. This module is expected to be working and is included by default.

**objspace.usemodules._winreg**

  • **name**: _winreg
  • **description**: use module _winreg
  • **command-line**: –withmod-_winreg
  • **command-line for negation**: –withoutmod-_winreg
  • **option type**: boolean option
  • **default**: False

Use the built-in ‘_winreg’ module, provides access to the Windows registry. This module is expected to be working and is included by default on Windows.

**objspace.usemodules.pypyjit**

  • **name**: pypyjit
  • **description**: use module pypyjit
  • **command-line**: –withmod-pypyjit
• **command-line for negation**: –withoutmod-pypyjit
• **option type**: boolean option
• **default**: False

Use the ‘pypyjit’ module.

**objc.space.usemodules._cffi_backend**

• **name**: _cffi_backend
• **description**: use module _cffi_backend
• **command-line**: –withmod-_cffi_backend
• **command-line for negation**: –withoutmod-_cffi_backend
• **option type**: boolean option
• **default**: False

Core of CFFI (http://cffi.readthedocs.org)

**objc.space.usemodules posix**

• **name**: posix
• **description**: use module posix
• **command-line**: –withmod-posix
• **option type**: boolean option
• **default**: True

Use the essential ‘posix’ module. This module is essential, included by default and cannot be removed (even when specified explicitly, the option gets overridden later).

**objc.space.usemodules._lsprof**

• **name**: _lsprof
• **description**: use module _lsprof
• **command-line**: –withmod-_lsprof
• **command-line for negation**: –withoutmod-_lsprof
• **option type**: boolean option
• **default**: False

Use the ‘_lsprof’ module.

**objc.space.usemodules._rawffi**

• **name**: _rawffi
• **description**: use module _rawffi
• command-line: –withmod-_rawffi
• command-line for negation: –withoutmod-rawffi
• option type: boolean option
• default: False
• suggestions:
  – objspace.usemodules.struct should be set to ‘True’

A module providing very low-level interface to C-level libraries, for use when implementing ctypes, not intended for a direct use at all.

objspace.usemodules.zlib

• name: zlib
• description: use module zlib
• command-line: –withmod-zlib
• command-line for negation: –withoutmod-zlib
• option type: boolean option
• default: False

Use the ‘zlib’ module. This module is expected to be working and is included by default.

objspace.usemodules.bz2

• name: bz2
• description: use module bz2
• command-line: –withmod-bz2
• command-line for negation: –withoutmod-bz2
• option type: boolean option
• default: False

Use the ‘bz2’ module. This module is expected to be working and is included by default.

objspace.usemodules.symbol

• name: symbol
• description: use module symbol
• command-line: –withmod-symbol
• command-line for negation: –withoutmod-symbol
• option type: boolean option
• default: True

Use the ‘symbol’ module. This module is expected to be working and is included by default.
objspace.usemodules._pickle_support

- **name**: _pickle_support
- **description**: use module _pickle_support
- **command-line**: –withmod-_pickle_support
- **command-line for negation**: –withoutmod-_pickle_support
- **option type**: boolean option
- **default**: True

Use the `_pickle_support` module. Internal helpers for pickling runtime builtin types (frames, cells, etc) for stackless tasklet pickling support. .. stackless: ../stackless.html

objspace.usemodules.zipimport

- **name**: zipimport
- **description**: use module zipimport
- **command-line**: –withmod-zipimport
- **command-line for negation**: –withoutmod-zipimport
- **option type**: boolean option
- **default**: False

This module implements zipimport mechanism described in PEP 302. It’s supposed to work and translate, so it’s included by default.

objspace.usemodules._testing

- **name**: _testing
- **description**: use module _testing
- **command-line**: –withmod-_testing
- **command-line for negation**: –withoutmod-_testing
- **option type**: boolean option
- **default**: True

Use the `_testing` module. This module exists only for PyPy own testing purposes. This module is expected to be working and is included by default.

objspace.usemodules._csv

- **name**: _csv
- **description**: use module _csv
- **command-line**: –withmod-_csv
- **command-line for negation**: –withoutmod-_csv
• **option type**: boolean option
  • **default**: False

Implementation in RPython for the core of the ‘csv’ module

**objcse.usemodules.operator**

• **name**: operator
  • **description**: use module operator
  • **command-line**: --withmod-operator
  • **command-line for negation**: --withoutmod-operator
  • **option type**: boolean option
  • **default**: True

Use the ‘operator’ module. This module is expected to be working and is included by default.

**objcse.usemodules.mmap**

• **name**: mmap
  • **description**: use module mmap
  • **command-line**: --withmod-mmap
  • **command-line for negation**: --withoutmod-mmap
  • **option type**: boolean option
  • **default**: False

Use the ‘mmap’ module. This module is expected to be fully working.

**objcse.usemodules.pwd**

• **name**: pwd
  • **description**: use module pwd
  • **command-line**: --withmod-pwd
  • **command-line for negation**: --withoutmod-pwd
  • **option type**: boolean option
  • **default**: False

Use the ‘pwd’ module. This module is expected to be fully working.

**objcse.usemodules.signal**

• **name**: signal
  • **description**: use module signal
  • **command-line**: --withmod-signal
**PyPy, Release 7.2.0**

- **command-line for negation:** –withoutmod-signal
- **option type:** boolean option
- **default:** False

Use the ‘signal’ module. This module is expected to be fully working.

**objspace.usemodules.exceptions**

- **name:** exceptions
- **description:** use module exceptions
- **command-line:** –withmod-exceptions
- **option type:** boolean option
- **default:** True

Use the ‘exceptions’ module. This module is essential, included by default and should not be removed.

**objspace.usemodules.imp**

- **name:** imp
- **description:** use module imp
- **command-line:** –withmod-imp
- **command-line for negation:** –withoutmod-imp
- **option type:** boolean option
- **default:** True

Use the ‘imp’ module. This module is included by default.

**objspace.usemodules._hashlib**

- **name:** _hashlib
- **description:** use module _hashlib
- **command-line:** –withmod-_hashlib
- **command-line for negation:** –withoutmod-_hashlib
- **option type:** boolean option
- **default:** False

Use the ‘_hashlib’ module. Used by the ‘hashlib’ standard lib module, and indirectly by the various cryptographic libs. This module is expected to be working and is included by default.

**objspace.usemodules.thread**

- **name:** thread
- **description:** use module thread
Use the ‘thread’ module.

**objspace.usemodules.cpyext**

- **name**: cpyext
- **description**: use module cpyext
- **command-line**: –withmod-cpyext
- **command-line for negation**: –withoutmod-cpyext
- **option type**: boolean option
- **default**: False

**requirements:**
- objspace.usemodules.array must be set to ‘True’

**suggestions:**
- translation.secondaryentrypoints should be set to ‘cpyext,main’

Use cpyext module to load and run CPython extension modules

Use the ‘termios’ module. This module is expected to be fully working.

**objspace.usemodules._pypyjson**

- **name**: _pypyjson
- **description**: use module _pypyjson
- **command-line**: –withmod-_pypyjson
- **command-line for negation**: –withoutmod-_pypyjson
- **option type**: boolean option
- **default**: False
RPython speedups for the stdlib json module

**objspace.usemodules._vmprof**

- **name:** _vmprof
- **description:** use module _vmprof
- **command-line:** --withmod-_vmprof
- **command-line for negation:** --withoutmod-_vmprof
- **option type:** boolean option
- **default:** False

**objspace.usemodules.crypt**

- **name:** crypt
- **description:** use module crypt
- **command-line:** --withmod-crypt
- **command-line for negation:** --withoutmod-crypt
- **option type:** boolean option
- **default:** False

Use the ‘crypt’ module. This module is expected to be fully working.

**objspace.usemodules._sha**

- **name:** _sha
- **description:** use module _sha
- **command-line:** --withmod-_sha
- **command-line for negation:** --withoutmod-_sha
- **option type:** boolean option
- **default:** False

Use the built-in `_sha` module. This module is expected to be working and is included by default. There is also a pure Python version in lib_pypy which is used if the built-in is disabled, but it is several orders of magnitude slower.

**objspace.usemodules._random**

- **name:** _random
- **description:** use module _random
- **command-line:** --withmod-_random
- **command-line for negation:** --withoutmod-_random
- **option type:** boolean option
Use the ‘_random’ module. It is necessary to use the module “random” from the standard library. This module is expected to be working and is included by default.

**objspace.usemodules.__pypy__**

- **name**: __pypy__
- **description**: use module __pypy__
- **command-line**: –withmod-__pypy__
- **command-line for negation**: –withoutmod-__pypy__
- **option type**: boolean option
- **default**: True

Use the ‘__pypy__’ module. This module is expected to be working and is included by default. It contains special PyPy-specific functionality. For example most of the special functions described in the object space proxies document are in the module. See the __pypy__ module documentation for more details.

**objspace.usemodules._demo**

- **name**: _demo
- **description**: use module _demo
- **command-line**: –withmod-_demo
- **command-line for negation**: –withoutmod-_demo
- **option type**: boolean option
- **default**: False

Use the ‘_demo’ module.

This is the demo module for mixed modules. Not enabled by default.

**objspace.usemodules.unicodedata**

- **name**: unicodedata
- **description**: use module unicodedata
- **command-line**: –withmod-unicodedata
- **command-line for negation**: –withoutmod-unicodedata
- **option type**: boolean option
- **default**: False

Use the ‘unicodedata’ module. This module is expected to be fully working.
objspace.usemodules._locale

- name: _locale
- description: use module _locale
- command-line: –withmod-_locale
- command-line for negation: –withoutmod-_locale
- option type: boolean option
- default: False

Use the ‘_locale’ module. This module runs _locale written in RPython (instead of ctypes version). It’s not really finished yet; it’s enabled by default on Windows.

objspace.usemodules._continuation

- name: _continuation
- description: use module _continuation
- command-line: –withmod-_continuation
- command-line for negation: –withoutmod-_continuation
- option type: boolean option
- default: False

Use the ‘_continuation’ module.

Exposes the continulet app-level primitives. See also –continuation.

objspace.usemodules.time

- name: time
- description: use module time
- command-line: –withmod-time
- command-line for negation: –withoutmod-time
- option type: boolean option
- default: True

Use the ‘time’ module.

objspace.usemodules._sre

- name: _sre
- description: use module _sre
- command-line: –withmod-_sre
- command-line for negation: –withoutmod-_sre
- option type: boolean option
• **default:** True

Use the ‘_sre’ module. This module is expected to be working and is included by default.

**objsusemodules.micronumpy**

• **name:** micronumpy
• **description:** use module micronumpy
• **command-line:** –withmod-micronumpy
• **command-line for negation:** –withoutmod-micronumpy
• **option type:** boolean option
• **default:** False

Use the micronumpy module. This module provides a very basic numpy-like interface. Major use-case is to show how jit scales for other code.

**objsusemodules.struct**

• **name:** struct
• **description:** use module struct
• **command-line:** –withmod-struct
• **command-line for negation:** –withoutmod-struct
• **option type:** boolean option
• **default:** False

Use the built-in ‘struct’ module. This module is expected to be working and is included by default. There is also a pure Python version in lib_pypy which is used if the built-in is disabled, but it is several orders of magnitude slower.

**objsusemodules._cppyy**

• **name:** _cppyy
• **description:** use module _cppyy
• **command-line:** –withmod-_cppyy
• **command-line for negation:** –withoutmod-_cppyy
• **option type:** boolean option
• **default:** False
• **requirements:**
  – **objsusemodules.cpyext** must be set to ‘True’

The internal backend for cppyy
objspace.usemodules.marshal

- **name**: marshal
- **description**: use module marshal
- **command-line**: –withmod-marshal
- **command-line for negation**: –withoutmod-marshal
- **option type**: boolean option
- **default**: True

Use the ‘marshal’ module. This module is expected to be working and is included by default.

objspace.usemodules.__builtin__

- **name**: __builtin__
- **description**: use module __builtin__
- **command-line**: –withmod-__builtin__
- **option type**: boolean option
- **default**: True

Use the ‘__builtin__’ module. This module is essential, included by default and should not be removed.

objspace.usemodules._ast

- **name**: _ast
- **description**: use module _ast
- **command-line**: –withmod-_ast
- **command-line for negation**: –withoutmod-_ast
- **option type**: boolean option
- **default**: True

Use the ‘_ast’ module. This module is expected to be working and is included by default.

objspace.usemodules.cmath

- **name**: cmath
- **description**: use module cmath
- **command-line**: –withmod-cmath
- **command-line for negation**: –withoutmod-cmath
- **option type**: boolean option
- **default**: True

Use the ‘cmath’ module. This module is expected to be working and is included by default.
**objspace.usemodules._ssl**

- **name**: _ssl
- **description**: use module _ssl
- **command-line**: –withmod-_ssl
- **command-line for negation**: –withoutmod-_ssl
- **option type**: boolean option
- **default**: False

Use the `_ssl` module, which implements SSL socket operations.

- **name**: usemodules
- **description**: Which Modules should be used

**objspace.allworkingmodules**

- **name**: allworkingmodules
- **description**: use as many working modules as possible
- **command-line**: –allworkingmodules
- **command-line for negation**: –no-allworkingmodules
- **option type**: boolean option
- **default**: True

This option enables the usage of all modules that are known to be working well and that translate without problems.

Note that this option defaults to True (except when running `py.py` because it takes a long time to start). To force it to False, use `--no-allworkingmodules`.

**objspace.extmodules**

- **name**: extmodules
- **description**: Comma-separated list of third-party builtin modules
- **command-line**: –ext
- **option type**: string option

You can pass a comma-separated list of third-party builtin modules which should be translated along with the standard modules within `pypy.module`.

The module names need to be fully qualified (i.e. have a . in them), be on the `$PYTHONPATH` and not conflict with any existing ones, e.g. `mypkg.somemod`.

Once translated, the module will be accessible with a simple:

```python
import somemod
```
**objspace.translationmodules**

- **name**: translationmodules
- **description**: use only those modules that are needed to run translate.py on pypy
- **command-line**: –translationmodules
- **command-line for negation**: –no-translationmodules
- **option type**: boolean option
- **default**: False
- **suggestions**:
  - objspace.allworkingmodules should be set to ‘False’

This option enables all modules which are needed to translate PyPy using PyPy.

**objspace.lonepycfiles**

- **name**: lonepycfiles
- **description**: Import pyc files with no matching py file
- **command-line**: –objspace-lonepycfiles
- **command-line for negation**: –no-objspace-lonepycfiles
- **option type**: boolean option
- **default**: False

If turned on, PyPy accepts to import a module x if it finds a file x.pyc even if there is no file x.py.

This is the way that CPython behaves, but it is disabled by default for PyPy because it is a common cause of issues: most typically, the x.py file is removed (manually or by a version control system) but the x module remains accidentally importable because the x.pyc file stays around.

The usual reason for wanting this feature is to distribute non-open-source Python programs by distributing pyc files only, but this use case is not practical for PyPy at the moment because multiple versions of PyPy compiled with various optimizations might be unable to load each other’s pyc files.

**objspace.soabi**

- **name**: soabi
- **description**: Tag to differentiate extension modules built for different Python interpreters
- **command-line**: –soabi
- **option type**: string option

This option controls the tag included into extension module file names. The default is something like pypy-14, which means that import foo will look for a file named foo.pypy-14.so (or foo.pypy-14.pyd on Windows).

This is an implementation of PEP3149, with two differences:

- the filename without tag foo.so is not considered.
- the feature is also available on Windows.
When set to the empty string (with `--soabi=`), the interpreter will only look for a file named `foo.so`, and will crash if this file was compiled for another Python interpreter.

**objspace.honor__builtins__**

- **name:** honor__builtins__
- **description:** Honor the __builtins__ key of a module dictionary
- **command-line:** --objspace-honor__builtins__
- **command-line for negation:** --no-objspace-honor__builtins__
- **option type:** boolean option
- **default:** False

**objspace.disable_call_speedhacks**

- **name:** disable_call_speedhacks
- **description:** make sure that all calls go through space.call_args
- **command-line:** --objspace-disable_call_speedhacks
- **command-line for negation:** --no-objspace-disable_call_speedhacks
- **option type:** boolean option
- **default:** False

disable the speed hacks that the interpreter normally does. Usually you don’t want to set this to False, but some object spaces require it.

**objspace.disable_entrypoints**

- **name:** disable_entrypoints
- **description:** Disable external entry points, notably the cpyext module and cffi’s embedding mode.
- **command-line:** --objspace-disable_entrypoints
- **command-line for negation:** --no-objspace-disable_entrypoints
- **option type:** boolean option
- **default:** False
- **requirements:**
  - `objspace.usemodules.cpyext` must be set to ‘False’

**objspace.hash**

- **name:** hash
- **description:** The hash function to use for strings: fnv from CPython 2.7 or siphash24 from CPython >= 3.4
- **command-line:** --hash
• **option type**: choice option
• **possible values**:
  – fnv
  – siphash24
• **default**: fnv

**objc.space.std**

**objc.space.std.withtproxy**

• **name**: withtproxy
• **description**: support transparent proxies
• **command-line**: –objc.space-std-withtproxy
• **command-line for negation**: –no-objc.space-std-withtproxy
• **option type**: boolean option
• **default**: True

Enable transparent proxies.

**objc.space.std.withprebuiltint**

• **name**: withprebuiltint
• **description**: prebuild commonly used int objects
• **command-line**: –objc.space-std-withprebuiltint
• **command-line for negation**: –no-objc.space-std-withprebuiltint
• **option type**: boolean option
• **default**: False

This option enables the caching of small integer objects (similar to what CPython does). The range of which integers are cached can be influenced with the –prebuiltintfrom and –prebuiltintto options.

**objc.space.std.prebuiltintfrom**

• **name**: prebuiltintfrom
• **description**: lowest integer which is prebuilt
• **command-line**: –prebuiltintfrom
• **option type**: integer option
• **default**: -5

see –objc.space-std-withprebuiltint.
**objspace.std.prebuiltintto**

- **name**: prebuiltintto
- **description**: highest integer which is prebuilt
- **command-line**: –prebuiltintto
- **option type**: integer option
- **default**: 100

See –objspace-std-withprebuil tint.

**objspace.std.withsmal long**

- **name**: withsmalllong
- **description**: use a version of ‘long’ in a C long long
- **command-line**: –objspace-std-withsmall long
- **command-line for negation**: –no-objspace-std-withsmall long
- **option type**: boolean option
- **default**: False

Enable “small longs”, an additional implementation of the Python type “long”, implemented with a C long long. It is mostly useful on 32-bit; on 64-bit, a C long long is the same as a C long, so its usefulness is limited to Python objects of type “long” that would anyway fit in an “int”.

**objspace.std.withspecialisedtuple**

- **name**: withspecialisedtuple
- **description**: use specialised tuples
- **command-line**: –objspace-std-withspecialisedtuple
- **command-line for negation**: –no-objspace-std-withspecialisedtuple
- **option type**: boolean option
- **default**: False

Use “specialized tuples”, a custom implementation for some common kinds of tuples. Currently limited to tuples of length 2, in three variants: (int, int), (float, float), and a generic (object, object).

**objspace.std.withliststrategies**

- **name**: withliststrategies
- **description**: enable optimized ways to store lists of primitives
- **command-line**: –objspace-std-withliststrategies
- **command-line for negation**: –no-objspace-std-withliststrategies
- **option type**: boolean option
- **default**: True
Enable list strategies: Use specialized representations for lists of primitive objects, such as ints.

**objspace.std.withmethodcachecounter**

- **name**: withmethodcachecounter
- **description**: try to cache methods and provide a counter in __pypy__. for testing purposes only.
- **command-line**: –objspace-std-withmethodcachecounter
- **command-line for negation**: –no-objspace-std-withmethodcachecounter
- **option type**: boolean option
- **default**: False

Testing/debug option for the method cache.

**objspace.std.methodcachesizeexp**

- **name**: methodcachesizeexp
- **description**: $2^{**
  \text{methodcachesizeexp}}$ is the size of the of the method cache
- **command-line**: –objspace-std-methodcachesizeexp
- **option type**: integer option
- **default**: 11

Set the cache size (number of entries) for the method cache.

**objspace.std.intshortcut**

- **name**: intshortcut
- **description**: special case addition and subtraction of two integers in BINARY_ADD//BINARY_SUBTRACT and their inplace counterparts
- **command-line**: –objspace-std-intshortcut
- **command-line for negation**: –no-objspace-std-intshortcut
- **option type**: boolean option
- **default**: False

Optimize the addition and subtraction of two integers. Enabling this option gives small speedups.

**objspace.std.optimized_list_getitem**

- **name**: optimized_list_getitem
- **description**: special case the ‘list[integer]’ expressions
- **command-line**: –objspace-std-optimized_list_getitem
- **command-line for negation**: –no-objspace-std-optimized_list_getitem
- **option type**: boolean option
- **default**: False

Optimized list[int] a bit.

`objspace.std.newshortcut`

- **name**: newshortcut
- **description**: cache and shortcut calling `__new__` from builtin types
- **command-line**: –objc-space-std-newshortcut
- **command-line for negation**: –no-objc-space-std-newshortcut
- **option type**: boolean option
- **default**: False

Performance only: cache and shortcut calling `__new__` from builtin types

- **name**: std
- **description**: Standard Object Space Options
- **name**: objspace
- **description**: Object Space Options

**The `--opt` or `-O` translation option**

This meta-option selects a default set of optimization settings to use during a translation. Usage:

```bash
bin/rpython --opt=#
bin/rpython -O#
```

where # is the desired optimization level. The valid choices are:

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–opt=0</td>
<td>all optimizations off; fastest translation (*)</td>
</tr>
<tr>
<td>–opt=1</td>
<td>non-time-consuming optimizations on (*)</td>
</tr>
<tr>
<td>–opt=size</td>
<td>minimize the size of the final executable (*)</td>
</tr>
<tr>
<td>–opt=mem</td>
<td>minimize the run-time RAM consumption (in-progress)</td>
</tr>
<tr>
<td>–opt=2</td>
<td>all optimizations on; good run-time performance</td>
</tr>
<tr>
<td>–opt=3</td>
<td>same as –opt=2; remove asserts; gcc profiling (**)</td>
</tr>
<tr>
<td>–opt=jit</td>
<td>includes the JIT and tweak other optimizations for it</td>
</tr>
</tbody>
</table>

(*): The levels 0, 1 and size use the Boehm-Demers-Weiser garbage collector (Debian package `libgc-dev`). The translation itself is faster and consumes less memory; the final executable is smaller but slower. The other levels use one of our built-in custom garbage collectors.

(**): The level 3 enables gcc profile-driven recompilation when translating PyPy.

The exact set of optimizations enabled by each level depends on the backend. Individual translation targets can also select their own options based on the level: when translating PyPy, the level `mem` enables the memory-saving object implementations in the object space; levels 2 and 3 enable the advanced object implementations that give an increase in performance; level 3 also enables gcc profile-driven recompilation.

The default level is 2.
3.5.6 Command line reference

Manual pages

pypy

SYNOPSIS

pypy [options] [-c cmd| -m mod| file.py] [arg...]

OPTIONS

- i Inspect interactively after running script.
- O Skip assert statements.
- OO Remove docstrings when importing modules in addition to -O.
- c CMD Program passed in as CMD (terminates option list).
- S Do not import site on initialization.
- s Don’t add the user site directory to sys.path.
- u Unbuffered binary stdout and stderr.
- h, --help Show a help message and exit.
- m MOD Library module to be run as a script (terminates option list).
- E Ignore environment variables (such as PYTHONPATH).
- B Disable writing bytecode (.pyc) files.
- X track-resources Produce a ResourceWarning whenever a file or socket is closed by the garbage collector.
--version Print the PyPy version.
--info Print translation information about this PyPy executable.
--jit ARG Low level JIT parameters. Mostly internal. Run --jit help for more information.

ENVIRONMENT

PYTHONPATH Add directories to pypy’s module search path. The format is the same as shell’s PATH.

PYTHONSTARTUP A script referenced by this variable will be executed before the first prompt is displayed, in interactive mode.

PYTHONDONTWRITEBYTECODE If set to a non-empty value, equivalent to the -B option. Disable writing .pyc files.

PYTHONINSPECT If set to a non-empty value, equivalent to the -i option. Inspect interactively after running the specified script.

PYTHONIOENCODING If this is set, it overrides the encoding used for stdin/stdout/stderr. The syntax is encoding-name:errorhandler The errorhandler part is optional and has the same meaning as in str.encode.
**PYTHONNOUSERSITE** If set to a non-empty value, equivalent to the `-s` option. Don’t add the user site directory to `sys.path`.

**PYTHONWARNINGS** If set, equivalent to the `-W` option (warning control). The value should be a comma-separated list of `-W` parameters.

**PYPYLOG** If set to a non-empty value, enable logging, the format is:

- `fname` or `+fname` logging for profiling: includes all `debug_start/debug_stop` but not any nested `debug_print`. `fname` can be `-` to log to `stderr`. The `+fname` form can be used if there is a `:` in `fname`
- `:fname` Full logging, including `debug_print`
- `prefix:fname` Conditional logging. Multiple prefixes can be specified, comma-separated. Only sections whose name match the prefix will be logged.

- `PYPYLOG=jit-log-opt,jit-backend:logfile` will generate a log suitable for `jitviewer`, a tool for debugging performance issues under PyPy.

**PYPY_IRC_TOPIC** If set to a non-empty value, print a random #pypy IRC topic at startup of interactive mode.

PyPy’s default garbage collector is called incminimark - it’s an incremental, generational moving collector. Here we hope to explain a bit how it works and how it can be tuned to suit the workload.

Incminimark first allocates objects in so called nursery - place for young objects, where allocation is very cheap, being just a pointer bump. The nursery size is a very crucial variable - depending on your workload (one or many processes) and cache sizes you might want to experiment with it via `PYPY_GC_NURSERY` environment variable. When the nursery is full, there is performed a minor collection. Freed objects are no longer referencable and just die, just by not being referenced anymore; on the other hand, objects found to still be alive must survive and are copied from the nursery to the old generation. Either to arenas, which are collections of objects of the same size, or directly allocated with malloc if they’re larger. (A third category, the very large objects, are initially allocated outside the nursery and never move.)

Since Incminimark is an incremental GC, the major collection is incremental: the goal is not to have any pause longer than 1ms, but in practice it depends on the size and characteristics of the heap: occasionally, there can be pauses between 10-100ms.

### Semi-manual GC management

If there are parts of the program where it is important to have a low latency, you might want to control precisely when the GC runs, to avoid unexpected pauses. Note that this has effect only on major collections, while minor collections continue to work as usual.

As explained above, a full major collection consists of $N$ steps, where $N$ depends on the size of the heap; generally speaking, it is not possible to predict how many steps will be needed to complete a collection.

- `gc.enable()` and `gc.disable()` control whether the GC runs collection steps automatically. When the GC is disabled the memory usage will grow indefinitely, unless you manually call `gc.collect()` and `gc.collect_step()`.
- `gc.collect()` runs a full major collection.
- `gc.collect_step()` runs a single collection step. It returns an object of type `GcCollectStepStats`, the same which is passed to the corresponding `GC Hooks`. The following code is roughly equivalent to a `gc.collect()`:

```python
while True:
    if gc.collect_step().major_is_done:
        break
```
For a real-world example of usage of this API, you can look at the 3rd-party module `pypytools.gc.custom`, which also provides a `with customgc.nogc()` context manager to mark sections where the GC is forbidden.

### Fragmentation

Before we discuss issues of “fragmentation”, we need a bit of precision. There are two kinds of related but distinct issues:

- If the program allocates a lot of memory, and then frees it all by dropping all references to it, then we might expect to see the RSS to drop. (RSS = Resident Set Size on Linux, as seen by “top”; it is an approximation of the actual memory usage from the OS’s point of view.) This might not occur: the RSS may remain at its highest value. This issue is more precisely caused by the process not returning “free” memory to the OS. We call this case “unreturned memory”.

- After doing the above, if the RSS didn’t go down, then at least future allocations should not cause the RSS to grow more. That is, the process should reuse unreturned memory as long as it has got some left. If this does not occur, the RSS grows even larger and we have real fragmentation issues.

### gc.get_stats

There is a special function in the `gc` module called `get_stats(memory_pressure=False)`. `memory_pressure` controls whether or not to report memory pressure from objects allocated outside of the GC, which requires walking the entire heap, so it’s disabled by default due to its cost. Enable it when debugging mysterious memory disappearance.

Example call looks like that:

```python
>>> gc.get_stats(True)
Total memory consumed:
GC used: 4.2MB (peak: 4.2MB)
in arenas: 763.7kB
rawmalloced: 383.1kB
nursery: 3.1MB
raw assembler used: 0.0kB
memory pressure: 0.0kB
----------------------------
Total: 4.2MB

Total memory allocated:
GC allocated: 4.5MB (peak: 4.5MB)
in arenas: 763.7kB
rawmalloced: 383.1kB
nursery: 3.1MB
raw assembler allocated: 0.0kB
memory pressure: 0.0kB
----------------------------
Total: 4.5MB
```

In this particular case, which is just at startup, GC consumes relatively little memory and there is even less unused, but allocated memory. In case there is a lot of unreturned memory or actual fragmentation, the “allocated” can be much higher than “used”. Generally speaking, “peak” will more closely resemble the actual memory consumed as reported by RSS. Indeed, returning memory to the OS is a hard and not solved problem. In PyPy, it occurs only if an arena is entirely free—a contiguous block of 64 pages of 4 or 8 KB each. It is also rare for the “rawmalloced” category, at least for common system implementations of `malloc()`.
The details of various fields:

- **GC in arenas** - small old objects held in arenas. If the amount “allocated” is much higher than the amount “used”, we have unreturned memory. It is possible but unlikely that we have internal fragmentation here. However, this unreturned memory cannot be reused for any `malloc()`, including the memory from the “rawmalloced” section.

- **GC rawmalloced** - large objects allocated with malloc. This is gives the current (first block of text) and peak (second block of text) memory allocated with `malloc()`. The amount of unreturned memory or fragmentation caused by `malloc()` cannot easily be reported. Usually you can guess there is some if the RSS is much larger than the total memory reported for “GC allocated”, but do keep in mind that this total does not include malloc’ed memory not known to PyPy’s GC at all. If you guess there is some, consider using `jemalloc` as opposed to system malloc.

- **nursery** - amount of memory allocated for nursery, fixed at startup, controlled via an environment variable

- **raw assembler allocated** - amount of assembler memory that JIT feels responsible for

- **memory pressure, if asked for** - amount of memory we think got allocated via external malloc (eg loading cert store in SSL contexts) that is kept alive by GC objects, but not accounted in the GC

### GC Hooks

GC hooks are user-defined functions which are called whenever a specific GC event occur, and can be used to monitor GC activity and pauses. You can install the hooks by setting the following attributes:

- `gc.hook.on_gc_minor` Called whenever a minor collection occurs. It corresponds to `gc-minor` sections inside PYPYLOG.

- `gc.hook.on_gc_collect_step` Called whenever an incremental step of a major collection occurs. It corresponds to `gc-collect-step` sections inside PYPYLOG.

- `gc.hook.on_gc_collect` Called after the last incremental step, when a major collection is fully done. It corresponds to `gc-collect-done` sections inside PYPYLOG.

To uninstall a hook, simply set the corresponding attribute to `None`. To install all hooks at once, you can call `gc.hooks.set(obj)`, which will look for methods `on_gc_*` on `obj`. To uninstall all the hooks at once, you can call `gc.hooks.reset()`.

The functions called by the hooks receive a single `stats` argument, which contains various statistics about the event.

Note that PyPy cannot call the hooks immediately after a GC event, but it has to wait until it reaches a point in which the interpreter is in a known state and calling user-defined code is harmless. It might happen that multiple events occur before the hook is invoked: in this case, you can inspect the value `stats.count` to know how many times the event occurred since the last time the hook was called. Similarly, `stats.duration` contains the total time spent by the GC for this specific event since the last time the hook was called.

On the other hand, all the other fields of the `stats` object are relative only to the last event of the series.

The attributes for `GcMinorStats` are:

- **count** The number of minor collections occurred since the last hook call.

- **duration** The total time spent inside minor collections since the last hook call, in seconds.

- **duration_min** The duration of the fastest minor collection since the last hook call.

- **duration_max** The duration of the slowest minor collection since the last hook call.
total_memory_used  The amount of memory used at the end of the minor collection, in bytes. This include
the memory used in arenas (for GC-managed memory) and raw-malloced memory (e.g., the content of
numpy arrays).

pinned_objects  the number of pinned objects.
The attributes for GcCollectStepStats are:
count, duration, duration_min, duration_max  See above.
oldstate, newstate  Integers which indicate the state of the GC before and after the step.
major_is_done  Boolean which indicate whether this was the last step of the major collection
The value of oldstate and newstate is one of these constants, defined inside gc.GcCollectStepStats:
STATE_SCANNING, STATE_MARKING, STATE_SWEEPING, STATE_FINALIZING, STATE_USERDEL. It is
possible to get a string representation of it by indexing the GC_STATES tuple.
The attributes for GcCollectStats are:
count  See above.
num_major_collects  The total number of major collections which have been done since the start. Contrarily to
count, this is an always-growing counter and it’s not reset between invocations.
arenas_count_before, arenas_count_after  Number of arenas used before and after the major collec-
tion.
arenas_bytes  Total number of bytes used by GC-managed objects.
rawmalloc_bytes_before, rawmalloc_bytes_after  Total number of bytes used by raw-malloced ob-
jects, before and after the major collection.
Note that GcCollectStats has not got a duration field. This is because all the GC work is done inside
gc-collect-step: gc-collect-done is used only to give additional stats, but doesn’t do any actual work.
Here is an example of GC hooks in use:

```python
import sys
import gc

class MyHooks(object):
    done = False

    def on_gc_minor(self, stats):
        print 'gc-minor: count = %02d, duration = %d' % (stats.count, stats.duration)

    def on_gc_collect_step(self, stats):
        old = gc.GcCollectStepStats.GC_STATES[stats.oldstate]
        new = gc.GcCollectStepStats.GC_STATES[stats.newstate]
        print 'gc-collect-step: %s --> %s' % (old, new)
        print ' count = %02d, duration = %d' % (stats.count, stats.duration)

    def on_gc_collect(self, stats):
        print 'gc-collect-done: count = %02d' % stats.count
        self.done = True

hooks = MyHooks()
gc.hooks.set(hooks)
```

(continues on next page)
Environment variables

PyPy’s default incminimark garbage collector is configurable through several environment variables:

**PYPY_GC_NURSERY** The nursery size. Defaults to 1/2 of your last-level cache, or 4M if unknown. Small values (like 1 or 1KB) are useful for debugging.

**PYPY_GC_NURSERY_DEBUG** If set to non-zero, will fill nursery with garbage, to help debugging.

**PYPY_GC_INCREMENT_STEP** The size of memory marked during the marking step. Default is size of nursery times 2. If you mark it too high your GC is not incremental at all. The minimum is set to size that survives minor collection times 1.5 so we reclaim anything all the time.

**PYPY_GC_MAJOR_COLLECT** Major collection memory factor. Default is 1.82, which means trigger a major collection when the memory consumed equals 1.82 times the memory really used at the end of the previous major collection.

**PYPY_GC_GROWTH** Major collection threshold’s max growth rate. Default is 1.4. Useful to collect more often than normally on sudden memory growth, e.g. when there is a temporary peak in memory usage.

**PYPY_GC_MAX** The max heap size. If coming near this limit, it will first collect more often, then raise an RPython MemoryError, and if that is not enough, crash the program with a fatal error. Try values like 1.6GB.

**PYPY_GC_MAX_DELTA** The major collection threshold will never be set to more than \( \text{PYPY_GC_MAX} \) - \( \text{PYPY_GC_MAX_DELTA} \) the amount really used after a collection. Defaults to 1/8th of the total RAM size (which is constrained to be at most 2/3/4GB on 32-bit systems). Try values like 200MB.

**PYPY_GC_MIN** Don’t collect while the memory size is below this limit. Useful to avoid spending all the time in the GC in very small programs. Defaults to 8 times the nursery.

**PYPY_GC_DEBUG** Enable extra checks around collections that are too slow for normal use. Values are 0 (off), 1 (on major collections) or 2 (also on minor collections).

**PYPY_GC_MAX_PINNED** The maximal number of pinned objects at any point in time. Defaults to a conservative value depending on nursery size and maximum object size inside the nursery. Useful for debugging by setting it to 0.

### SEE ALSO

python(1)

pypy3

### SYNOPSIS

pypy3 [options] [-c cmd|–m modfile.pyl|–] [arg…]
OPTIONS

-\(i\) Inspect interactively after running script.
-\(O\) Skip assert statements.
-\(OO\) Remove docstrings when importing modules in addition to \(-O\).
-\(c\) CMD Program passed in as CMD (terminates option list).
-\(S\) Do not import site on initialization.
-\(s\) Don’t add the user site directory to sys.path.
-\(u\) Unbuffered binary stdout and stderr.
-\(h\), --help Show a help message and exit.
-\(m\) MOD Library module to be run as a script (terminates option list).
-\(W\) ARG Warning control (arg is action:message:category:module:lineno).
-\(E\) Ignore environment variables (such as PYTHONPATH).
-\(B\) Disable writing bytecode (.pyc) files.
-\(X\) track-resources Produce a ResourceWarning whenever a file or socket is closed by the garbage collector.

--version Print the PyPy version.
--info Print translation information about this PyPy executable.
--jit ARG Low level JIT parameters. Mostly internal. Run --jit help for more information.

ENVIRONMENT

PYTHONPATH Add directories to pypy3’s module search path. The format is the same as shell’s PATH.

PYTHONSTARTUP A script referenced by this variable will be executed before the first prompt is displayed, in interactive mode.

PYTHONDONTWRITEBYTECODE If set to a non-empty value, equivalent to the \(-B\) option. Disable writing .pyc files.

PYTHONINSPECT If set to a non-empty value, equivalent to the \(-i\) option. Inspect interactively after running the specified script.

PYTHONIOENCODING If this is set, it overrides the encoding used for stdin/stdout/stderr. The syntax is encoding-name:erro\r\r\nhandler The errorhandler part is optional and has the same meaning as in str.encode.

PYTHONNOSITE If set to a non-empty value, equivalent to the \(-s\) option. Don’t add the user site directory to sys.path.

PYTHONWARNINGS If set, equivalent to the \(-W\) option (warning control). The value should be a comma-separated list of \(-W\) parameters.

PYPYLOG If set to a non-empty value, enable logging, the format is:

\*fname or +fname* logging for profiling: includes all debug_start/debug_stop but not any nested debug_print. \*fname can be \(- to log to stderr. The +fname form can be used if there is a : in fname.

\*fname* Full logging, including debug_print.
prefix:fname  Conditional logging. Multiple prefixes can be specified, comma-separated. Only sections whose name match the prefix will be logged.

PYPYLOG=jit-log-opt,jit-backend:logfile will generate a log suitable for jitviewer, a tool for debugging performance issues under PyPy.

PYPY_IRC_TOPIC  If set to a non-empty value, print a random #pypy IRC topic at startup of interactive mode.

PyPy’s default garbage collector is called incminimark - it’s an incremental, generational moving collector. Here we hope to explain a bit how it works and how it can be tuned to suit the workload.

Incminimark first allocates objects in so called nursery - place for young objects, where allocation is very cheap, being just a pointer bump. The nursery size is a very crucial variable - depending on your workload (one or many processes) and cache sizes you might want to experiment with it via PYPY_GC_NURSERY environment variable. When the nursery is full, there is performed a minor collection. Freed objects are no longer referencable and just die, just by not being referenced any more; on the other hand, objects found to still be alive must survive and are copied from the nursery to the old generation. Either to arenas, which are collections of objects of the same size, or directly allocated with malloc if they’re larger. (A third category, the very large objects, are initially allocated outside the nursery and never move.)

Since Incminimark is an incremental GC, the major collection is incremental: the goal is not to have any pause longer than 1ms, but in practice it depends on the size and characteristics of the heap: occasionally, there can be pauses between 10-100ms.

Semi-manual GC management

If there are parts of the program where it is important to have a low latency, you might want to control precisely when the GC runs, to avoid unexpected pauses. Note that this has effect only on major collections, while minor collections continue to work as usual.

As explained above, a full major collection consists of N steps, where N depends on the size of the heap; generally speaking, it is not possible to predict how many steps will be needed to complete a collection.

gc.enable() and gc.disable() control whether the GC runs collection steps automatically. When the GC is disabled the memory usage will grow indefinitely, unless you manually call gc.collect() and gc.collect_step().

gc.collect() runs a full major collection.

gc.collect_step() runs a single collection step. It returns an object of type GcCollectStepStats, the same which is passed to the corresponding GC Hooks. The following code is roughly equivalent to a gc.collect():

```python
while True:
    if gc.collect_step().major_is_done:
        break
```

For a real-world example of usage of this API, you can look at the 3rd-party module pypytools.gc.custom, which also provides a with customgc.nogc() context manager to mark sections where the GC is forbidden.

Fragmentation

Before we discuss issues of “fragmentation”, we need a bit of precision. There are two kinds of related but distinct issues:

- If the program allocates a lot of memory, and then frees it all by dropping all references to it, then we might expect to see the RSS to drop. (RSS = Resident Set Size on Linux, as seen by “top”; it is an approximation of the actual memory usage from the OS’s point of view.) This might not occur: the RSS may remain at its highest
value. This issue is more precisely caused by the process not returning “free” memory to the OS. We call this case “unreturned memory”.

- After doing the above, if the RSS didn’t go down, then at least future allocations should not cause the RSS to grow more. That is, the process should reuse unreturned memory as long as it has got some left. If this does not occur, the RSS grows even larger and we have real fragmentation issues.

**gc.get_stats**

There is a special function in the `gc` module called `get_stats(memory_pressure=False)`.

`memory_pressure` controls whether or not to report memory pressure from objects allocated outside of the GC, which requires walking the entire heap, so it’s disabled by default due to its cost. Enable it when debugging mysterious memory disappearance.

Example call looks like that:

```python
>>> gc.get_stats(True)
Total memory consumed:
GC used: 4.2MB (peak: 4.2MB)
in arenas: 763.7kB
rawmalloced: 383.1kB
nursery: 3.1MB
raw assembler used: 0.0kB
memory pressure: 0.0kB
-----------------------------
Total: 4.2MB

Total memory allocated:
GC allocated: 4.5MB (peak: 4.5MB)
in arenas: 763.7kB
rawmalloced: 383.1kB
nursery: 3.1MB
raw assembler allocated: 0.0kB
memory pressure: 0.0kB
-----------------------------
Total: 4.5MB
```

In this particular case, which is just at startup, GC consumes relatively little memory and there is even less unused, but allocated memory. In case there is a lot of unreturned memory or actual fragmentation, the “allocated” can be much higher than “used”. Generally speaking, “peak” will more closely resemble the actual memory consumed as reported by RSS. Indeed, returning memory to the OS is a hard and not solved problem. In PyPy, it occurs only if an arena is entirely free—a contiguous block of 64 pages of 4 or 8 KB each. It is also rare for the “rawmalloced” category, at least for common system implementations of `malloc()`.

The details of various fields:

- **GC in arenas** - small old objects held in arenas. If the amount “allocated” is much higher than the amount “used”, we have unreturned memory. It is possible but unlikely that we have internal fragmentation here. However, this unreturned memory cannot be reused for any `malloc()`, including the memory from the “rawmalloced” section.

- **GC rawmalloced** - large objects allocated with malloc. This is gives the current (first block of text) and peak (second block of text) memory allocated with `malloc()`. The amount of unreturned memory or fragmentation caused by `malloc()` cannot easily be reported. Usually you can guess there is some if the RSS is much larger than the total memory reported for “GC allocated”, but do keep in mind that this total does not include malloc’ed memory not known to PyPy’s GC at all. If you guess there is some, consider using `jemalloc` as opposed to system malloc.
• nursery - amount of memory allocated for nursery, fixed at startup, controlled via an environment variable
• raw assembler allocated - amount of assembler memory that JIT feels responsible for
• memory pressure, if asked for - amount of memory we think got allocated via external malloc (e.g., loading certificate store in SSL contexts) that is kept alive by GC objects, but not accounted in the GC

**GC Hooks**

GC hooks are user-defined functions which are called whenever a specific GC event occurs, and can be used to monitor GC activity and pauses. You can install the hooks by setting the following attributes:

```
gc.hook.on_gc_minor  Called whenever a minor collection occurs. It corresponds to gc-minor sections inside PYPYLOG.
```

```
gc.hook.on_gc_collect_step  Called whenever an incremental step of a major collection occurs. It corresponds to gc-collect-step sections inside PYPYLOG.
```

```
gc.hook.on_gc_collect  Called after the last incremental step, when a major collection is fully done. It corresponds to gc-collect-done sections inside PYPYLOG.
```

To uninstall a hook, simply set the corresponding attribute to `None`. To install all hooks at once, you can call `gc.hooks.set(obj)`, which will look for methods `on_gc_*` on `obj`. To uninstall all the hooks at once, you can call `gc.hooks.reset()`.

The functions called by the hooks receive a single `stats` argument, which contains various statistics about the event. Note that PyPy cannot call the hooks immediately after a GC event, but it has to wait until it reaches a point in which the interpreter is in a known state and calling user-defined code is harmless. It might happen that multiple events occur before the hook is invoked: in this case, you can inspect the value `stats.count` to know how many times the event occurred since the last time the hook was called. Similarly, `stats.duration` contains the total time spent by the GC for this specific event since the last time the hook was called.

On the other hand, all the other fields of the `stats` object are relative only to the last event of the series.

The attributes for `GcMinorStats` are:

- **count** The number of minor collections occurred since the last hook call.
- **duration** The total time spent inside minor collections since the last hook call, in seconds.
- **duration_min** The duration of the fastest minor collection since the last hook call.
- **duration_max** The duration of the slowest minor collection since the last hook call.

- **total_memory_used** The amount of memory used at the end of the minor collection, in bytes. This includes the memory used in arenas (for GC-managed memory) and raw-malloced memory (e.g., the content of numpy arrays).

- **pinned_objects** The number of pinned objects.

The attributes for `GcCollectStepStats` are:

- **count, duration, duration_min, duration_max** See above.
- **oldstate, newstate** Integers which indicate the state of the GC before and after the step.
- **major_is_done** Boolean which indicates whether this was the last step of the major collection.

The value of `oldstate` and `newstate` is one of these constants, defined inside `gc.GcCollectStepStats`: `STATE_SCANNING`, `STATE_MARKING`, `STATE_SWEEPING`, `STATE_FINALIZING`, `STATE_USERDEL`. It is possible to get a string representation of it by indexing the `GC_STATES` tuple.

3.5. **Source Code Documentation**
The attributes for `GcCollectStats` are:

- **count**: See above.
- **num_major_collects**: The total number of major collections which have been done since the start. Contrarily to `count`, this is an always-growing counter and it’s not reset between invocations.
- **arenas_count_before**, **arenas_count_after**: Number of arenas used before and after the major collection.
- **arenas_bytes**: Total number of bytes used by GC-managed objects.
- **rawmalloc_bytes_before**, **rawmalloc_bytes_after**: Total number of bytes used by raw-malloced objects, before and after the major collection.

Note that `GcCollectStats` has not got a `duration` field. This is because all the GC work is done inside `gc-collect-step`: `gc-collect-done` is used only to give additional stats, but doesn’t do any actual work.

Here is an example of GC hooks in use:

```python
import sys
import gc

class MyHooks(object):
    done = False

    def on_gc_minor(self, stats):
        print 'gc-minor: count = %02d, duration = %d' % (stats.count, stats.duration)

    def on_gc_collect_step(self, stats):
        old = gc.GcCollectStepStats.GC_STATES[stats.oldstate]
        new = gc.GcCollectStepStats.GC_STATES[stats.newstate]
        print 'gc-collect-step: %s --> %s' % (old, new)
        print ' count = %02d, duration = %d' % (stats.count, stats.duration)

    def on_gc_collect(self, stats):
        print 'gc-collect-done: count = %02d' % stats.count
        self.done = True

hooks = MyHooks()
gc.hooks.set(hooks)

# simulate some GC activity
lst = []
while not hooks.done:
    lst = [lst, 1, 2, 3]
```

**Environment variables**

PyPy’s default incminimark garbage collector is configurable through several environment variables:

- **PYPY_GC_NURSERY**: The nursery size. Defaults to 1/2 of your last-level cache, or 4M if unknown. Small values (like 1 or 1KB) are useful for debugging.
- **PYPY_GC_NURSERY_DEBUG**: If set to non-zero, will fill nursery with garbage, to help debugging.
- **PYPY_GC_INCREMENT_STEP**: The size of memory marked during the marking step. Default is size of nursery times 2. If you mark it too high your GC is not incremental at all. The minimum is set to size that survives
PyPy, Release 7.2.0

minor collection times 1.5 so we reclaim anything all the time.

**PYPY_GC_MAJOR_COLLECT**  Major collection memory factor. Default is 1.82, which means trigger a major collection when the memory consumed equals 1.82 times the memory really used at the end of the previous major collection.

**PYPY_GC_GROWTH**  Major collection threshold’s max growth rate. Default is 1.4. Useful to collect more often than normally on sudden memory growth, e.g. when there is a temporary peak in memory usage.

**PYPY_GC_MAX**  The max heap size. If coming near this limit, it will first collect more often, then raise an RPython MemoryError, and if that is not enough, crash the program with a fatal error. Try values like 1.6GB.

**PYPY_GC_MAX_DELTA**  The major collection threshold will never be set to more than PYPY_GC_MAX_DELTA the amount really used after a collection. Defaults to 1/8th of the total RAM size (which is constrained to be at most 2/3/4GB on 32-bit systems). Try values like 200MB.

**PYPY_GC_MIN**  Don’t collect while the memory size is below this limit. Useful to avoid spending all the time in the GC in very small programs. Defaults to 8 times the nursery.

**PYPY_GC_DEBUG**  Enable extra checks around collections that are too slow for normal use. Values are 0 (off), 1 (on major collections) or 2 (also on minor collections).

**PYPY_GC_MAX_PINNED**  The maximal number of pinned objects at any point in time. Defaults to a conservative value depending on nursery size and maximum object size inside the nursery. Useful for debugging by setting it to 0.

SEE ALSO

python3(1)

### 3.5.7 PyPy directory cross-reference

Here is a fully referenced alphabetical two-level deep directory overview of PyPy:

<table>
<thead>
<tr>
<th>Directory</th>
<th>explanation/links</th>
</tr>
</thead>
<tbody>
<tr>
<td>pypy/bin/</td>
<td>command-line scripts, mainly pypy/bin/pyinteractive.py</td>
</tr>
<tr>
<td>pypy/config/</td>
<td>handles the numerous options for building and running PyPy</td>
</tr>
<tr>
<td>pypy/doc/</td>
<td>text versions of PyPy developer documentation</td>
</tr>
<tr>
<td>pypy/doc/config</td>
<td>documentation for the numerous translation options</td>
</tr>
<tr>
<td>pypy/doc/discussion</td>
<td>drafts of ideas and documentation</td>
</tr>
<tr>
<td>pypy/goal/</td>
<td>our main PyPy-translation scripts live here</td>
</tr>
<tr>
<td>pypy/interpreter</td>
<td>bytecode interpreter and related objects (frames, functions, modules,...)</td>
</tr>
<tr>
<td>pypy/interpreter/astcompiler</td>
<td>interpreter-level Python source parser</td>
</tr>
<tr>
<td>pypy/interpreter/bytecodecompiler</td>
<td>interpreter-level bytecode compiler, via an AST representation</td>
</tr>
<tr>
<td>pypy/module/</td>
<td>contains mixed modules implementing core modules with both application and interpreter level code. Not all are finished and working. Use the --withmod-xxx or --allworkingmodules translation options.</td>
</tr>
<tr>
<td>pypy/objspace</td>
<td>object space implementations</td>
</tr>
<tr>
<td>pypy/objspace/StdObjSpace</td>
<td>implementing CPython’s objects and types</td>
</tr>
<tr>
<td>pypy/tool/</td>
<td>various utilities and hacks used from various places</td>
</tr>
<tr>
<td>pypy/tool/pytest</td>
<td>support code for our testing methods</td>
</tr>
<tr>
<td>*/test/</td>
<td>many directories have a test subdirectory containing test modules (see Testing in PyPy)</td>
</tr>
<tr>
<td>_cache/</td>
<td>holds cache files from various purposes</td>
</tr>
</tbody>
</table>

3.5. Source Code Documentation 199
3.6 PyPy’s Release Process

3.6.1 Release Policy

We try to create a stable release a few times a year. These are released on a branch named like release-pypy3.5-v2.x or release-pypy3.5-v4.x, and each release is tagged, for instance release-pypy3.5-v4.0.1.

The release version number should be bumped. A micro release increment means there were no changes that justify rebuilding c-extension wheels, since the wheels are marked with only major.minor version numbers. It is often not clear what constitutes a “major” release versus a “minor” release, the release manager can make that call.

After release, inevitably there are bug fixes. It is the responsibility of the committer who fixes a bug to make sure this fix is on the release branch, so that we can then create a tagged bug-fix release, which will hopefully happen more often than stable releases.

3.6.2 How to Create a PyPy Release

As a meta rule setting up issues in the tracker for items here may help not forgetting things. A set of todo files may also work.

Check and prioritize all issues for the release, postpone some if necessary, create new issues also as necessary. An important thing is to get the documentation into an up-to-date state!

3.6.3 Release Steps

Make the release branch

This is needed only in case you are doing a new major version; if not, you can probably reuse the existing release branch.

We want to be able to freely merge default into the branch and vice-versa; thus we need to do a complicate dance to avoid to patch the version number when we do a merge:

```
$ hg up -r default
$ # edit the version to e.g. 7.0.0-final
$ hg ci
$ hg branch release-pypy2.7-v7.x && hg ci
$ hg up -r default
$ # edit the version to 7.1.0-alpha0
$ hg ci
$ hg up -r release-pypy2.7-v7.x
$ hg merge default
$ # edit the version to AGAIN 7.0.0-final
$ hg ci
```

Then, we need to do the same for the 3.x branch:

```
$ hg up -r py3.5
$ hg merge default # this brings the version fo 7.1.0-alpha0
$ hg branch release-pypy3.5-v7.x
$ # edit the version to 7.0.0-final
$ hg ci
$ hg up -r py3.5
$ hg merge release-pypy3.5-v7.x
```

(continues on next page)
To change the version, you need to edit three files:

- module/sys/version.py
- module/cpyext/include/patchlevel.h
- doc/conf.py

Other steps

- Make sure the RPython builds on the buildbot pass with no failures
- Maybe bump the SOABI number in module/imp/importing. This has many implications, so make sure the PyPy community agrees to the change. Wheels will use the major.minor release numbers in the name, so bump them if there is an incompatible change to cpyext.
- Update and write documentation
  - update pypy/doc/contributor.rst (and possibly LICENSE) pypy/doc/tool/makecontributor.py generates the list of contributors
  - rename pypy/doc/whatsnew_head.rst to whatsnew_VERSION.rst create a fresh whatsnew_head.rst after the release and add the new file to pypy/doc/index-of-whatsnew.rst
  - write release announcement pypy/doc/release-VERSION.rst The release announcement should contain a direct link to the download page
  - Add the new files to pypy/doc/index-{whatsnew,release-notes}.rst
- Build and upload the release tar-balls
  - go to pypy/tool/release and run force-builds.py <release branch> The following JIT binaries should be built, however, we need more buildbots windows, linux-32, linux-64, osx64, armlf-raspberrian, armel, freebsd64
  - wait for builds to complete, make sure there are no failures
  - send out a mailing list message asking for people to test before uploading to prevent having to upload more than once
  - add a tag on the pypy/jitviewer repo that corresponds to pypy release, so that the source tarball can be produced in the next steps
  - download the builds, repackage binaries. Tag the release-candidate version (it is important to mark this as a candidate since usually at least two tries are needed to complete the process) and download and repackage source from bitbucket. You may find it convenient to use the repackage.sh script in pypy/tool/release to do this.
    - Otherwise repackage and upload source “-src.tar.bz2” to bitbucket and to cobra, as some packagers prefer a clearly labeled source package (download e.g. https://bitbucket.org/pypy/pypy/get/release-2.5.x.tar.bz2, unpack, rename the top-level directory to “pypy-2.5.0-src”, repack, and upload)
  - Upload binaries to https://bitbucket.org/pypy/pypy/downloads
- Send out a mailing list message asking for last-minute comments and testing
- RELEASE !
– update pypy.org (under extradoc/pypy.org), rebuild and commit, using the hashes produced from the repackage.sh script or by hand
– post announcement on morepypyp.blogspot.com
– send announcements to twitter.com, pypy-dev, python-list, python-announce, python-dev . . .

• If all is OK, document the released version
  – add a tag on the codespeed web site that corresponds to pypy release
  – revise versioning at https://readthedocs.org/projects/pypy
  – tag the final release(s) with appropriate tags
4.1 Historical release notes

4.1.1 Combined releases

PyPy v7.1.1: release of 2.7, and 3.6-beta

The PyPy team is proud to release a bug-fix release version 7.1.1 of PyPy, which includes two different interpreters:

- PyPy2.7, which is an interpreter supporting the syntax and the features of Python 2.7
- PyPy3.6-beta: this is the second official release of PyPy to support 3.6 features, although it is still considered beta quality.

The interpreters are based on much the same codebase, thus the double release.

As always, this release is 100% compatible with the previous one and fixed several issues and bugs raised by the growing community of PyPy users. We strongly recommend updating.

The PyPy3.6 release is still not production quality so your mileage may vary. There are open issues with incomplete compatibility and c-extension support.

You can download the v7.1.1 releases here:

  http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project. If PyPy is not quite good enough for your needs, we are available for direct consulting work.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.
PyPy, Release 7.2.0

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7, 3.6. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This PyPy release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Unfortunately at the moment of writing our ARM buildbots are out of service, so for now we are not releasing any binary for the ARM architecture, although PyPy does support ARM 32 bit processors.

Changelog

Changes shared across versions:

- Improve performance of u'' .append
- Prevent a crash in zlib when flushing a closed stream
- Fix a few corner cases when encountering unicode values above 0x110000
- Teach the JIT how to handle very large constant lists, sets, or dicts
- Fix building on ARM32 (issue 2984)
- Fix a bug in register assignment in ARM32
- Package windows DLLs needed by cffi modules next to the cffi c-extensions (issue 2988)
- Cleanup and refactor JIT code to remove rpython .jit .metainterp .typesystem
- Fix memoryviews of ctype structures with padding, (cpython issue 32780)
- CFFI updated to as-yet-unreleased 1.12.3

Python 3.6 only:

- Override some errno .E* values that were added to MSVC in v2010 so that errno .E* == errno .WSAE* as in CPython
- Do the same optimization that CPython does for (1, 2, 3, *a) (but at the AST level)
- str .maketrans was broken (issue 2991)
- Raise a TypeError when using buffers and unicode such as '' .strip (buffer) and 'a' < buffer
- Support _overlapped and asyncio on win32
- Fix an issue where ''.join(list_of_strings) would rarely confuse utf8 and bytes (issue 2997)

PyPy v7.1.0: release of 2.7, and 3.6-beta

The PyPy team is proud to release the version 7.1.0 of PyPy, which includes two different interpreters:

- PyPy2.7, which is an interpreter supporting the syntax and the features of Python 2.7
• PyPy3.6-beta: this is the second official release of PyPy to support 3.6 features, although it is still considered beta quality.

The interpreters are based on much the same codebase, thus the double release.

This release, coming fast on the heels of 7.0 in February, finally merges the internal refactoring of unicode representation as UTF-8. Removing the conversions from strings to unicode internally lead to a nice speed bump.

We also improved the ability to use the buffer protocol with ctype structures and arrays.

Until we can work with downstream providers to distribute builds with PyPy, we have made packages for some common packages available as wheels.

The CFFI backend has been updated to version 1.12.2. We recommend using CFFI rather than c-extensions to interact with C, and cppy for interacting with C++ code.

As always, this release is 100% compatible with the previous one and fixed several issues and bugs raised by the growing community of PyPy users. We strongly recommend updating.

The PyPy3.6 release is still not production quality so your mileage may vary. There are open issues with incomplete compatibility and c-extension support.

You can download the v7.1 releases here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project. If PyPy is not quite good enough for your needs, we are available for direct consulting work.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7, 3.6. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This PyPy release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Unfortunately at the moment of writing our ARM buildbots are out of service, so for now we are not releasing any binary for the ARM architecture, although PyPy does support ARM 32 bit processors.

**Changelog**

Changes shared across versions

- Use utf8 internally to represent unicode, with the goal of never using rpython-level unicode
- Update cffi to 1.12.2
- Improve performance of long operations where one of the operands fits into an int
• Since _ctypes is implemented in pure python over libffi, add interfaces and methods to support the buffer interface from python. Specifically, add a __pypy__.newmemoryview function to create a memoryview and extend the use of the PyPy-specific __buffer__ class method. This enables better buffer sharing between ctypes and NumPy.

• Add copying to zlib

• Improve register allocation in the JIT by using better heuristics

• Include <sys/sysmacros.h> on Gnu/Hurd

• Mostly for completeness sake: support for rlib.jit.promote_unicode, which behaves like promote_string, but for python unicode objects

• Correctly initialize the d_type and d_name members of builtin descriptors to fix a segfault related to class-methods in Cython

• Expand documentation of __pypy__ module

C-API (cpyext) improvements shared across versions

• Move PyTuple_Type.tp_new to C

• Call internal methods from PyDict_XXXItem() instead of going through dunder methods (CPython cpyext compatibility)

Python 3.6 only

• Support for os.PathLike in the posix module

• Update idellib for 3.6.1

• Make BUILD_CONST_KEY_MAP JIT-friendly

• Adapt code that optimizes sys.exc_info() to wordcode

• Fix annotation bug found by attrs

PyPy v7.0.0: triple release of 2.7, 3.5 and 3.6-alpha

The PyPy team is proud to release the version 7.0.0 of PyPy, which includes three different interpreters:

• PyPy2.7, which is an interpreter supporting the syntax and the features of Python 2.7

• PyPy3.5, which supports Python 3.5

• PyPy3.6-alpha: this is the first official release of PyPy to support 3.6 features, although it is still considered alpha quality.

All the interpreters are based on much the same codebase, thus the triple release.

Until we can work with downstream providers to distribute builds with PyPy, we have made packages for some common packages available as wheels.

The GC hooks, which can be used to gain more insights into its performance, has been improved and it is now possible to manually manage the GC by using a combination of gc.disable and gc.collect_step. See the GC blog post.

We updated the cffi module included in PyPy to version 1.12, and the cppyy backend to 1.4. Please use these to wrap your C and C++ code, respectively, for a JIT friendly experience.

As always, this release is 100% compatible with the previous one and fixed several issues and bugs raised by the growing community of PyPy users. We strongly recommend updating.
The PyPy3.6 release and the Windows PyPy3.5 release are still not production quality so your mileage may vary. There are open issues with incomplete compatibility and c-extension support.

The utf8 branch that changes internal representation of unicode to utf8 did not make it into the release, so there is still more goodness coming. You can download the v7.0 releases here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project. If PyPy is not quite good enough for your needs, we are available for direct consulting work.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7, 3.5 and 3.6. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

The PyPy release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Unfortunately at the moment of writing our ARM buildbots are out of service, so for now we are not releasing any binary for the ARM architecture.

Changelog

If not specified, the changes are shared across versions

- Support __set_name__, __init_subclass__ (Py3.6)
- Support cppyy in Py3.5 and Py3.6
- Use implementation-specific site directories in sysconfig (Py3.5, Py3.6)
- Adding detection of gcc to sysconfig (Py3.5, Py3.6)
- Fix multiprocessing regression on newer glibcs
- Make sure ‘blocking-ness’ of socket is set along with default timeout
- Include crypt.h for crypt() on Linux
- Improve and re-organize the contributing documentation
- Make the __module__ attribute writable, fixing an incompatibility with NumPy 1.16
- Implement Py_ReprEnter, Py_ReprLeave(), `PyMarshal_ReadObjectFromString, PyMarshal_WriteObjectToString, PyObject_DelItemString, PyMapping_DelItem, PyMapping_DelItemString, PyEval_GetFrame, PyOS_InputHook, PyErr_FormatFromCause (Py3.6),
- Implement new wordcode instruction encoding (Py3.6)
• Log additional gc-minor and gc-collect-step info in the PYPYLOG

• The reverse-debugger (revdb) branch has been merged to the default branch, so it should always be up-to-date. You still need a special pypy build, but you can compile it from the same source as the one we distribute for the v7.0.0 release. For more information, see https://bitbucket.org/pypy/revdb

• Support underscores in numerical literals like '4_2' (Py3.6)

• Pre-emptively raise MemoryError if the size of dequeue in _collections.deque is too large (Py3.5)

• Fix multithreading issues in calls to os.setenv

• Add missing defines and typedefs for numpy and pandas on MSVC

• Add CPython macros like Py_NAN to header files

• Rename the MethodType to instancemethod, like CPython

• Better support for async with in generators (Py3.5, Py3.6)

• Improve the performance of pow(a, b, c) if c is a large integer

• Now vmprof works on FreeBSD

• Support GNU Hurd, fixes for FreeBSD

• Add deprecation warning if type of result of __float__ is float inherited class (Py3.6)

• Fix async generator bug when yielding a StopIteration (Py3.6)

• Speed up max(list-of-int) from non-jitted code

• Fix Windows os.listdir() for some cases (see CPython #32539)

• Add select.PIPE_BUF

• Use subprocess to avoid shell injection in shutil module - backport of https://bugs.python.org/issue34540

• Rename _Py_ZeroStruct to _Py_FalseStruct (Py3.5, Py3.6)

• Remove some cpyext names for Py3.5, Py3.6

• Enable use of unicode file names in dlopen

• Backport CPython fix for thread.RLock

• Make GC hooks measure time in seconds (as opposed to an opaque unit)

• Refactor and reorganize tests in test_lib_pypy

• Check error values in socket.setblocking (Py3.6)

• Add support for FsPath to os.unlink() (Py3.6)

• Fix freezing builtin modules at translation

• Tweak W_UnicodedictionaryStrategy which speeds up dictionaries with only unicode keys

We also refactored many parts of the JIT bridge optimizations, as well as cpyext internals, and together with new contributors fixed issues, added new documentation, and cleaned up the codebase.

PyPy2.7 and PyPy3.5 v6.0 dual release

The PyPy team is proud to release both PyPy2.7 v6.0 (an interpreter supporting Python 2.7 syntax), and a PyPy3.5 v6.0 (an interpreter supporting Python 3.5 syntax). The two releases are both based on much the same codebase, thus the dual release.
This release is a feature release following our previous 5.10 incremental release in late December 2017. Our C-API compatibility layer cpyext is now much faster (see the blog post) as well as more complete. We have made many other improvements in speed and CPython compatibility. Since the changes affect the included python development header files, all c-extension modules must be recompiled for this version.

Until we can work with downstream providers to distribute builds with PyPy, we have made packages for some common packages available as wheels. You may compile yourself using `pip install --no-build-isolation <package>`, the no-build-isolation is currently needed for pip v10.

First-time python users are often stumped by silly typos and omissions when getting started writing code. We have improved our parser to emit more friendly syntax errors, making PyPy not only faster but more friendly.

The GC now has hooks to gain more insights into its performance.

The Matplotlib TkAgg backend now works with PyPy, as do pygame and pygobject.

We updated the cffi module included in PyPy to version 1.11.5, and the cppyy backend to 0.6.0. Please use these to wrap your C and C++ code, respectively, for a JIT friendly experience.

As always, this release is 100% compatible with the previous one and fixed several issues and bugs raised by the growing community of PyPy users. We strongly recommend updating.

The Windows PyPy3.5 release is still considered beta-quality. There are open issues with unicode handling especially around system calls and c-extensions.

The utf8 branch that changes internal representation of unicode to utf8 did not make it into the release, so there is still more goodness coming. We also began working on a Python3.6 implementation, help is welcome.

You can download the v6.0 releases here:

```
http://pypy.org/download.html
```

We would like to thank our donors for the continued support of the PyPy project. If PyPy is not quite good enough for your needs, we are available for direct consulting work.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7 and CPython 3.5. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

The PyPy release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Changelog

- Speed up C-API method calls, and make most Py*_Check calls C macros
PyPy, Release 7.2.0

- Speed up C-API slot method calls
- Enable TkAgg backend support for matplotlib
- Support `hastzinfo` and `tzinfo` in the C-API `PyDateTime*` structures
- `datetime.h` is now more similar to CPython
- We now support `PyUnicode_AsUTF{16,32}String`, `_PyLong_AsByteArray`, `_PyLong_AsByteArrayO`
- PyPy3.5 on Windows is compiled with the Microsoft Visual Compiler v14, like CPython
- Fix performance of attribute lookup when more than 80 attributes are used
- Improve performance on passing built-in types to C-API C code
- Improve the performance of datetime and timedelta by skipping the consistency checks of the datetime values (they are correct by construction)
- Improve handling of bigint's, including fixing `int_divmod`
- Improve reporting of GC statistics
- Accept unicode filenames in `dbm.open()`
- Improve RPython support for half-floats
- Added missing attributes to C-API `instancemethod` on pypy3
- Store error state in thread-local storage for C-API.
- Fix JIT bugs exposed in the sre module
- Improve speed of Python parser, improve ParseError messages and SyntaxError
- Handle JIT hooks more efficiently
- Fix a rare GC bug exposed by intensive use of cpyext Buffers

We also refactored many parts of the JIT bridge optimizations, as well as cpyext internals, and together with new contributors fixed issues, added new documentation, and cleaned up the codebase.

PyPy 5.10.1

We have released a bugfix PyPy3.5-v5.10.1 due to the following issues:
- Fix `time.sleep(float('nan'))` which would hang on windows
- Fix missing `errno` constants on windows
- Fix issue 2718 for the REPL on linux
- Fix an overflow in converting 3 secs to nanoseconds (issue 2717)
- Flag `kwarg` to `os.setxattr` had no effect
- Fix the winreg module for unicode entries in the registry on windows

Note that many of these fixes are for our new beta version of PyPy3.5 on windows. There may be more unicode problems in the windows beta version especially around the subject of directory- and file-names with non-ascii characters.

Our downloads are available now. On macos, we recommend you wait for the Homebrew package.

Thanks to those who reported the issues.
What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7 and CPython 3.5. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This PyPy 3.5 release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Please update, and continue to help us make PyPy better.

Cheers

The PyPy Team

PyPy2.7 and PyPy3.5 v5.10 dual release

The PyPy team is proud to release both PyPy2.7 v5.10 (an interpreter supporting Python 2.7 syntax), and a final PyPy3.5 v5.10 (an interpreter for Python 3.5 syntax). The two releases are both based on much the same codebase, thus the dual release.

This release is an incremental release with very few new features, the main feature being the final PyPy3.5 release that works on linux and OS X with beta windows support. It also includes fixes for vmprof cooperation with greenlets.

Compared to 5.9, the 5.10 release contains mostly bugfixes and small improvements. We have in the pipeline big new features coming for PyPy 6.0 that did not make the release cut and should be available within the next couple months.

As always, this release is 100% compatible with the previous one and fixed several issues and bugs raised by the growing community of PyPy users. As always, we strongly recommend updating.

There are quite a few important changes that are in the pipeline that did not make it into the 5.10 release. Most important are speed improvements to cpyext (which will make numpy and pandas a bit faster) and utf8 branch that changes internal representation of unicode to utf8, which should help especially the Python 3.5 version of PyPy.

This release concludes the Mozilla Open Source grant for having a compatible PyPy 3.5 release and we’re very grateful for that. Of course, we will continue to improve PyPy 3.5 and probably move to 3.6 during the course of 2018.

You can download the v5.10 releases here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7 and CPython 3.5. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

4.1. Historical release notes
PyPy, Release 7.2.0

We also welcome developers of other dynamic languages to see what RPython can do for them.

The PyPy release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Changelog

- improve ssl handling on windows for pypy3 (makes pip work)
- improve unicode handling in various error reporters
- fix vmprof cooperation with greenlets
- fix some things in cpyext
- test and document the cmp(nan, nan) == 0 behaviour
- don’t crash when calling sleep with inf or nan
- fix bugs in _io module
- inspect.isbuiltin() now returns True for functions implemented in C
- allow the sequences future-import, docstring, future-import for CPython bug compatibility
- Issue #2699: non-ascii messages in warnings
- posix.lockf
- fixes for FreeBSD platform
- add .debug files, so builds contain debugging info, instead of being stripped
- improvements to cppy
- issue #2677 copy pure c PyBuffer_{From,To}Contiguous from cpython
- issue #2682, split firstword on any whitespace in sqlite3
- ctypes: allow ptr[0] = foo when ptr is a pointer to struct
- matplotlib will work with tkagg backend once matplotlib pr #9356 is merged
- improvements to utf32 surrogate handling
- cffi version bump to 1.11.2

PyPy2.7 and PyPy3.5 v5.9 dual release

The PyPy team is proud to release both PyPy2.7 v5.9 (an interpreter supporting Python 2.7 syntax), and a beta-quality PyPy3.5 v5.9 (an interpreter for Python 3.5 syntax). The two releases are both based on much the same codebase, thus the dual release. Note that PyPy3.5 supports Linux 64bit only for now.

This new PyPy2.7 release includes the upstream stdlib version 2.7.13, and PyPy3.5 includes the upstream stdlib version 3.5.3.
NumPy and Pandas now work on PyPy2.7 (together with Cython 0.27.1). Issues that appeared as excessive memory use were cleared up and other incompatibilities were resolved. The C-API compatibility layer does slow down code which crosses the python-c interface often, we have ideas on how it could be improved, and still recommend using pure python on PyPy or interfacing via CFFI. Many other modules based on C-API exentions now work on PyPy as well.

Cython 0.27.1 (released very recently) supports more projects with PyPy, both on PyPy2.7 and PyPy3.5 beta. Note version 0.27.1 is now the minimum version that supports this version of PyPy, due to some interactions with updated C-API interface code.

We optimized the JSON parser for recurring string keys, which should decrease memory use to 50% and increase parsing speed by up to 15% for large JSON files with many repeating dictionary keys (which is quite common).

CFFI, which is part of the PyPy release, has been updated to 1.11.1, improving an already great package for interfacing with C. CFFI now supports complex arguments in API mode, as well as char16_t and char32_t and has improved support for callbacks.

Please let us know if your use case is slow, we have ideas how to make things faster but need real-world examples (not micro-benchmarks) of problematic code.

Work sponsored by a Mozilla grant continues on PyPy3.5; numerous fixes from CPython were ported to PyPy. Of course the bug fixes and performance enhancements mentioned above are part of both PyPy2.7 and PyPy3.5 beta.

As always, this release fixed many other issues and bugs raised by the growing community of PyPy users. We strongly recommend updating.

You can download the v5.9 releases here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7 and CPython 3.5. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

The PyPy 2.7 release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Highlights of the PyPy2.7, cpyext, and RPython changes (since 5.8 released June, 2017)

See also issues that were resolved

Note that these are also merged into PyPy 3.5

- New features and cleanups

4.1. Historical release notes
– Add support for PyFrozenSet_New, PyObject_HashNotImplemented, PyObject_Print(NULL, ...), PyObject_RichCompareBool(a, a, ...), PyType_IS_GC (does nothing), PyUnicode_FromFormat
– ctypes char_p and unichar_p indexing now CPython compatible
– gcdump now reports largest object
– More complete support in the _curses CFFI module
– Add cPickle.Unpickler.find_global (issue 1853)
– Fix PyErr_Fetch + PyErr_NormalizeException with no exception set
– Simplify gc.get_referrers() to return the opposite of gc.get_referents()
– Update RevDB to version pypy2.7-v5.6.2
– Previously, instance.method would return always the same bound method object, when gotten from the same instance (as far as is and id() can tell). CPython doesn't do that. Now PyPy, like CPython, returns a different bound method object every time. For type.method, PyPy2 still returns always the same unbound method object; CPython does it for built-in types but not for user-defined types
– Link to disable PaX protection for the JIT when needed
– Update build instructions and an rarely used Makefile
– Recreate support for using leakfinder in cpyext tests which had suffered bit-rot, disable due to many false positives
– Add more functionality to sysconfig
– Added __swappedbytes__ support for ctypes.Structure
– Better support the inspect module on frames

• Bug Fixes
– Fix issue 2592 - cpyext PyListObject.pop, pop_end must return a value
– Implement PyListObject.getstorage_copy
– Fix for reversed(dictproxy) issue 2601
– Fix for duplicate names in ctypes’ _fields__, issue 2621
– Update built-in pyexpat module on win32 to use UTF-8 version not UTF-16
– gc.get_objects now handles objects with finalizers more consistently
– Fixed memory leak in SSLContext.getpeercert returning validated certificates and SSLContext.get_ca_certs(binary_mode=True) (_get_crl_dp) CPython issue 29738

• Performance improvements:
– Improve performance of bytearray.extend by rewriting portions in app-level
– Optimize list accesses with constant indexes better by retaining more information about them
– Add a jit driver for array.count and array.index
– Improve information retained in a bridge wrt array
– Move some dummy CAPI functions and Py*_Check functions from RPython into pure C macros
– In the fast zip(intlist1, intlist2) implementation, don’t wrap and unwrap all the ints
– Cache string keys that occur in JSON dicts, as they are likely to repeat
• RPython improvements
  – Do not preallocate a RPython list if we only know an upper bound on its size
  – Issue 2590: fix the bounds in the GC when allocating a lot of objects with finalizers
  – Replace magical NOT RPYTHON comment with a decorator
  – Implement `socket.sendmsg()` / `recvmsg()` for py3.5
  – Add `memory_pressure` for `_SSLSocket` objects

• Degradations
  – Disable vmprof on win32, due to upstream changes that break the internal `_vmprof` module

Highlights of the PyPy3.5 release (since 5.8 beta released June 2017)

• New features
  – Add support for `_PyNamespace_New`, `PyMemoryView_FromMemory`, `Py_EnterRecursiveCall` raising `RecursionError`, `PyObject_LengthHint`, `PyUnicode_FromKindAndData`, `PyDict_SetDefault`, `PyGenObject`, `PyGenObject`, `PyUnicode_Substring`, `PyLong_FromUnicodeObject`
  – Implement `PyType_FromSpec` (PEP 384) and fix issues with PEP 489 support
  – Support the new version of `os.stat()` on win32
  – Use `stat3()` on Posix
  – Accept buffer objects as filenames, except for `os.listdir`
  – Make slices of array `memoryview`s usable as writable buffers if contiguous
  – Better handling of '%s' formatting for byte strings which might be utf-8 encoded
  – Update the macros `Py_DECREF` and similar to use the CPython 3.5 version
  – Ensure that `mappingproxy` is recognised as a mapping, not a sequence
  – Enable PGO for CLang
  – Rework `cppyy` packaging and rename the backend to `_cppyy`
  – Support for librssl 2.5.4
  – Mirror CPython `classmethod __reduce__` which fixes pickling test
  – Use utf-8 for readline history file
  – Allow assigning '__class__' between `ModuleType` and its subclasses
  – Add async slot functions in `cpyext`

• Bug Fixes
  – Try to make `openssl` CFFI bindings more general and future-proof
  – Better support `importlib` by only listing built-in modules in `sys.builtin`
  – Add `memory_pressure` to large CFFI allocations in `_lzma`, issue 2579
  – Fix for reversed(mapping object) issue 2601
  – Fixing regression with non-started generator receiving non-None, should always raise `TypeError`
  – `itertools.islice`: use same logic as CPython, fixes 2643

4.1. Historical release notes
• Performance improvements:
  –
  • The following features of Python 3.5 are not implemented yet in PyPy:
    – PEP 442: Safe object finalization

Please update, and continue to help us make PyPy better.

Cheers

**PyPy2.7 and PyPy3.5 v5.8 dual release**

The PyPy team is proud to release both PyPy2.7 v5.8 (an interpreter supporting Python 2.7 syntax), and a beta-quality PyPy3.5 v5.8 (an interpreter for Python 3.5 syntax). The two releases are both based on much the same codebase, thus the dual release. Note that PyPy3.5 supports Linux 64bit only for now.

This new PyPy2.7 release includes the upstream stdlib version 2.7.13, and PyPy3.5 includes the upstream stdlib version 3.5.3.

We fixed critical bugs in the shadowstack rootfinder garbage collector strategy that crashed multithreaded programs and very rarely showed up even in single threaded programs.

We added native PyPy support to profile frames in the vmprof statistical profiler.

The `struct` module functions `pack*` and `unpack*` are now much faster, especially on raw buffers and bytearrays. Microbenchmarks show a 2x to 10x speedup. Thanks to Gambit Research for sponsoring this work.

This release adds (but disables by default) link-time optimization and profile guided optimization of the base interpreter, which may make unjitted code run faster. To use these, translate with appropriate options. Be aware of issues with gcc toolchains, though.

Please let us know if your use case is slow, we have ideas how to make things faster but need real-world examples (not micro-benchmarks) of problematic code.

Work sponsored by a Mozilla grant continues on PyPy3.5; numerous fixes from CPython were ported to PyPy and PEP 489 was fully implemented. Of course the bug fixes and performance enhancements mentioned above are part of both PyPy 2.7 and PyPy 3.5.

**CFFI**, which is part of the PyPy release, has been updated to an unreleased 1.10.1, improving an already great package for interfacing with C.

As always, this release fixed many other issues and bugs raised by the growing community of PyPy users. We strongly recommend updating.

You can download the v5.8 releases here:

```
http://pypy.org/download.html
```

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7 and CPython 3.5. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.
The PyPy 2.7 release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

**Highlights of the PyPy2.7, cpyext, and RPython changes (since 5.7 released March, 2017)**

See also issues that were resolved

Note that these are also merged into PyPy 3.5

- New features and cleanups
  - Implement `PyModule_New`, `Py_GetRecursionLimit`, `Py_SetRecursionLimit`, `Py_EnterRecursiveCall`, `Py_LeaveRecursiveCall`, populate `tp_descr_get` and `tp_descr_set` slots, add conversions of `__len__`, `__setitem__`, `__delitem__` to appropriate C-API slots
  - Fix for multiple inheritance in app-level for C-API defined classes
  - Revert a change that removed `tp_getattr` (Part of the 5.7.1 bugfix release)
  - Document more differences with CPython here
  - Add native PyPy support to profile frames in `vmprof`
  - Fix an issue with Exception order on failed import
  - Fix for a corner case of `__future__` imports
  - Update packaged Windows zlib, sqlite, expat and OpenSSL to versions used by CPython
  - Allow windows builds to use `jom.exe` for compiling in parallel
  - Rewrite `itertools.groupby()`, following CPython
  - Backport changes from PyPy 3.5 to minimize the code differences
  - Improve support for BSD using patches contributed by downstream
  - Support profile-guided optimization, enabled with `--profopt`, and specify training data `profoptpath`

- Bug Fixes
  - Correctly handle dict.pop where the popping key is not the same type as the dict’s and pop is called with a default (Part of the 5.7.1 bugfix release)
  - Improve our file’s universal newline .readline implementation for \n, \r confusion
  - Tweak issue where ctypes array `__base` was set on empty arrays, now it is closer to the implementation in CPython
  - Fix critical bugs in shadowstack that crashed multithreaded programs and very rarely showed up even in single threaded programs
  - Remove flaky fastpath function call from ctypes
  - Support passing a buffersize of 0 to socket.getsockopt
  - Avoid hash() returning -1 in cpyext

- Performance improvements:
- Tweaks made to improve performance by reducing the number of guards inserted in jitted code, based on feedback from users
- Add garbage collector memory pressure to some c-level allocations
- Speed up strack.pack, strack.pack_into
- Performance tweaks to round(x, n) for the case n == 0
- Improve zipfile performance by not doing repeated string concatenation

- RPython improvements
  - Improve the default shadowstack garbage collector, fixing a crash with multithreaded code and other issues
  - Make sure lstrip consumes the entire string
  - Support posix_fallocate and posix_fadvise, expose them on PyPy3.5
  - Test and fix for int_and() propagating wrong bounds
  - Improve the generated machine code by tracking the (constant) value of r11 across instructions. This lets us avoid reloading r11 with another (apparently slowish) “movabs” instruction, replacing it with either nothing or a cheaper variant.
  - Performance tweaks in the x86 JIT-generated machine code: rarely taken blocks are moved off-line. Also, the temporary register used to contain large constants is reused across instructions. This helps CPUs branch predictor
  - Refactor rpython.rtyper.controllerentry to use @specialize instead of _annspecialcase_
  - Refactor handling of buffers and memoryviews. Memoryviews will now be accepted in a few more places, e.g. in compile()

Highlights of the PyPy3.5 release (since 5.7 beta released March 2017)

- New features
  - Implement main part of PEP 489 (multi-phase extension module initialization)
  - Add docstrings to various modules and functions
  - Adapt many CPython bug/feature fixes from CPython 3.5 to PyPy3.5
  - Translation succeeds on Mac OS X, unfortunately our buildbot slave cannot be updated to the proper development versions of OpenSSL to properly package a release.
  - Implement "_SSL.Socket.server_side"
  - Do not silently ignore _swappedbytes_ in ctypes. We now raise a NotImplementedError
  - Implement and expose msvrct.SetErrorMode
  - Implement PyModule_GetState

- Bug Fixes
  - Fix inconsistencies in the xml.etree.ElementTree.Element class, which on CPython is hidden by the C version from ‘_elementtree’.
  - OSError(None,None) is different from OSError()
  - Get closer to supporting 32 bit windows, translation now succeeds and most lib-python/3/test runs
  - Call sys.__interactivehook__ at startup
- Let `OrderedDict.__init__` behave like CPython wrt. subclasses overriding `__setitem__`

- **Performance improvements:**
  - Use `<python> -m test` to run the CPython test suite, as documented by CPython, instead of our outdated `regverbose.py` script
  - Change `_cffi_src/openssl/callbacks.py` to stop relying on the CPython C API.
  - Avoid importing the full locale module during `_io` initialization, CPython change fbbf8b160e8d
  - Avoid freezing many app-level modules at translation, avoid importing many modules at startup
  - Refactor buffers, which allows an optimization for `bytearray()[:n].tobytes()`

- The following features of Python 3.5 are not implemented yet in PyPy:
  - PEP 442: Safe object finalization

Please update, and continue to help us make PyPy better.

Cheers

**PyPy 5.7.1**

We have released a bugfix PyPy2.7-v5.7.1 and PyPy3.5-v5.7.1 beta (Linux 64bit), due to the following issues:

- correctly handle an edge case in `dict.pop` (issue 2508)
- fix a regression to correctly handle multiple inheritance in a C-API type where the second base is an app-level class with a `__new__` function
- fix a regression to fill a C-API type's `tp_getattr` slot from a `__getattr__` method (issue 2523)

Thanks to those who reported the issues.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7 and CPython 3.5. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

The PyPy 2.7 release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Please update, and continue to help us make PyPy better.

Cheers

The PyPy Team
PyPy2.7 and PyPy3.5 v5.7 - two in one release

The PyPy team is proud to release both PyPy2.7 v5.7 (an interpreter supporting Python v2.7 syntax), and a beta-quality PyPy3.5 v5.7 (an interpreter for Python v3.5 syntax). The two releases are both based on much the same codebase, thus the dual release. Note that PyPy3.5 supports Linux 64bit only for now.

This new PyPy2.7 release includes the upstream stdlib version 2.7.13, and PyPy3.5 (our first in the 3.5 series) includes the upstream stdlib version 3.5.3.

We continue to make incremental improvements to our C-API compatibility layer (cpyext). PyPy2 can now import and run many C-extension packages, among the most notable are Numpy, Cython, and Pandas. Performance may be slower than CPython, especially for frequently-called short C functions. Please let us know if your use case is slow, we have ideas how to make things faster but need real-world examples (not micro-benchmarks) of problematic code.

Work proceeds at a good pace on the PyPy3.5 version due to a grant from the Mozilla Foundation, hence our first 3.5.3 beta release. Thanks Mozilla !!! While we do not pass all tests yet, asyncio works and as these benchmarks show it already gives a nice speed bump. We also backported the \texttt{format}ing from 3.6 (as an exception; otherwise \texttt{PyPy3.5} supports the Python 3.5 language).

CFFI has been updated to 1.10, improving an already great package for interfacing with C.

We now use shadowstack as our default gcrootfinder even on Linux. The alternative, asmgcc, will be deprecated at some future point. While about 3% slower, shadowstack is much more easily maintained and debuggable. Also, the performance of shadowstack has been improved in general: this should close the speed gap between Linux and other platforms.

As always, this release fixed many issues and bugs raised by the growing community of PyPy users. We strongly recommend updating.

You can download the v5.7 release here:

\url{http://pypy.org/download.html}

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7 and CPython 3.5. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

The PyPy 2.7 release supports:

\begin{itemize}
  \item \texttt{x86} machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
  \item newer \texttt{ARM} hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
  \item big- and little-endian variants of \texttt{PPC64} running Linux,
  \item \texttt{s390x} running Linux
\end{itemize}

Highlights of the PyPy2.7, cpyext, and RPython changes (since 5.6 released Nov, 2016)

See also issues that were resolved
• New features and cleanups
  – update the format of the PYPYLOG file and improvements to vmprof
  – emit more sysconfig values for downstream cextension packages including properly setting purelib and platlib to site-packages
  – add PyAnySet_Check, PyModule_GetName, PyWeakref_Check*, _PyImport_{Acquire, Release}Lock, PyGen_Check*, PyOS_AfterFork.
  – detect and raise on recreation of a PyPy object from a PyObject during tp_dealloc
  – refactor and clean up poor handling of unicode exposed in work on py3.5
  – builtin module cppy supports C++ 11, 14, etc. via cling (reflex has been removed)
  – adapt weakref according to CPython issue 19542, will be in CPython 2.7.14
  – support translations with cpyext and the Boehm GC (for special cases like RevDB
  – implement StringBuffer.get_raw_address for the buffer protocol, it is now possible to obtain the address of any readonly object without pinning it
  – refactor the initialization code in translating cpyext
  – use a cffi-style C parser to create rffi objects in cpyext, now the translating Python must have either cffi or pycparser available
  – implement move_to_end(last=True/False) on RPython ordered dicts, make available as __pypy__.move_to_end and, on py3.5, OrderedDict.move_to_end()
  – remove completely RPython space.wrap in a major cleanup, differentiate between space.newtext and space.newbytes on py3.5
  – any uncaught RPython exception in the interpreter is turned into a SystemError (rather than a segfault)
  – add translation time –disable_entrypoints option for embedding PyPy together with another RPython VM

• Bug Fixes
  – fix "".replace("", "x", num) to give the same result as CPython
  – create log files without the executable bit
  – disable clock_gettime() on OS/X, since we support 10.11 and it was only added in 10.12
  – support HAVE_FSTATVFS which was unintentionally always false
  – fix user-created C-API heap type, issue 2434
  – fix PyDict_Update is not actually the same as dict.update
  – assign tp_doc on PyTypeObject and tie it to the app-level __doc__ attribute issue 2446
  – clean up memory leaks around PyObject_GetBuffer, PyMemoryView_GET_BUFFER, PyMemoryView_FromBuffer, and PyBuffer_Release
  – improve support for creating C-extension objects from app-level classes, filling more slots, especially tp_new and tp_dealloc
  – fix for ctypes.c_bool returning bool restype, issue 2475
  – fix in corner cases with the GIL and C-API functions
  – allow overriding thread.local._init_ in a subclass, issue 2501
  – allow PyClass_New to be called with NULL as the first arguemnt, issue 2504

• Performance improvements:

4.1. Historical release notes
PyPy, Release 7.2.0

- clean-ups in the jit optimizeopt
- optimize if x is not None: return x or if x != 0: return x
- add jit.conditional_call_elidable(), a way to tell the JIT “conditionally call this function” returning a result
- try harder to propagate can_be_None=False information
- add rarithmetic.ovfcheck_int32_add/sub/mul
- add and use rgc.may_ignore_finalizer(): an optimization hint that makes the GC stop tracking the object
- replace malloc+memset with a single calloc, useful for large allocations?
- linux: try to implement os.urandom() as the syscall getrandom() if available
- propagate debug.ll_assert_not_none() through the JIT to reduce number of guards
- improve the performance of PyDict_Next
- improve dict.pop()
- improve the optimization of branchy Python code by retaining more information across failing guards
- add optimized “zero-copy” path for io.FileIO.readinto

- RPython improvements
  - improve the consistency of RPython annotation unions
  - add translation option –keepgoing to continue after the first AnnotationError
  - improve shadowstack to where it is now the default in place of asmgcc
  - add a rpython implementation of siphash24, allow choosing hash algorithm randomizing the seed
  - add rstack.stack_almost_full() and use it to avoid stack overflow due to the JIT where possible

Highlights of the PyPy3.5 release (since 5.5 alpha released Oct, 2016)

Development moved from the py3k branch to the py3.5 branch in the PyPy bitbucket repo.

- New features
  - this first PyPy3.5 release implements most of Python 3.5.3, exceptions are listed below
  - PEP 456 allowing secure and interchangable hash algorithms
  - use cryptography’s cffi backend for SSL

- Bug Fixes
  - implement fixes for some CPython issues that arose since the last release
  - solve deadlocks in thread locking mechanism

- Performance improvements:
  - do not create a list whenever descr_new of a bytesobject is called

- The following features of Python 3.5 are not implemented yet in PyPy:
  - PEP 442: Safe object finalization
  - PEP 489: Multi-phase extension module initialization
Please update, and continue to help us make PyPy better.

Cheers

4.1.2 CPython 2.7 compatible versions

**PyPy2.7 v5.6**

We have released PyPy2.7 v5.6, about two months after PyPy2.7 v5.4. This new PyPy2.7 release includes the upstream stdlib version 2.7.12.

We continue to make incremental improvements to our C-API compatibility layer (cpyext). We pass all but a few of the tests in the upstream numpy test suite.

Work proceeds at a good pace on the PyPy3.5 version due to a grant from the Mozilla Foundation, and some of those changes have been backported to PyPy2.7 where relevant.

The PowerPC and s390x backend have been enhanced with the capability use SIMD instructions for micronumpy loops.

We changed `timeit` to now report average +- standard deviation, which is better than the misleading minimum value reported in CPython.

We now support building PyPy with OpenSSL 1.1 in our built-in `_ssl` module, as well as maintaining support for previous versions.

CFFI has been updated to 1.9, improving an already great package for interfacing with C.

As always, this release fixed many issues and bugs raised by the growing community of PyPy users. We strongly recommend updating.

You can download the PyPy2.7 v5.6 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports:

- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux
Other Highlights (since 5.4 released Aug 31, 2016)

- New features
  - Allow tests run with -A to find libm.so even if it is a script not a dynamically loadable file
  - Backport fixes to rposix on windows from py3.5
  - Allow user-defined __getitem__ on subclasses of str and unicode
  - Add inode to scandir() on posix systems
  - Support more attributes on super
  - Issue #2386: non-latin1 unicode keys were ignored in unicode.format(**d)
  - Restore the ability to translate with CPython
  - Update to CFFI 1.9.0
  - Support the new buffer protocol in cpyext and numpypy
  - Add rposix.sync()
  - Support full-precision nanosecond times in os.stat()
  - Add documentation about the assembler backends to RPYthon
  - Search for the stdlibs from the libpypy shared object rather than the pypy-c exe, changes downstream packaging requirements
  - Add try_inline, like always_inline and dont_inline to RPYthon
  - Reject 'a'.strip(buffer(' ')) like cpython (the argument to strip must be str or unicode)
  - Allow warning.warn(('something', 1), Warning) like on CPython
  - Refactor rclock and add some more CLOCK_xxx constants on relevant platforms
  - Backport the faulthandler module from py3.5
  - Improve the error message when trying to call a method where the self parameter is missing in the definition
  - Implement rposix.cpu_count
  - Support translation on FreeBSD running on PowerPC
  - Implement __rmod__ on str and unicode types
  - Issue warnings for stricter handling of __new__, __init__ args
  - When using struct.unpack('q', ... try harder to prefer int to long
  - Support OpenSSL version 1.1 (in addition to version 1.0)

- Bug Fixes
  - Tweak a float comparison with 0 in backendopt.inline to avoid rounding errors
  - Fix translation of the sandbox
  - Fix for an issue where unicode.decode('utf8', 'custom_replace') messed up the last byte of a unicode string sometimes
  - fix some calls to functions through window’s COM interface
  - fix minor leak when the C call to socketpair() fails
  - make sure (-1.0 + 0j).__hash__() , (-1.0).__hash__() returns -2
PyPy, Release 7.2.0

- Fix for an issue where PyBytesResize was called on a fresh pyobj
- Fix bug in codewriter about passing the exitswitch variable to a call
- Don’t crash in merge_if_blocks if the values are symbolics
- Issue #2325/2361: __class__ assignment between two classes with the same slots
- Issue #2409: don’t leak the file descriptor when doing open('some-dir')
- Windows fixes around vmprof
- Don’t use sprintf() from inside a signal handler
- Test and fix bug from the guard_not_forced_2 branch, which didn’t save the floating-point register
- _numpypy.add.reduce returns a scalar now

- Performance improvements:
  - Improve method calls on oldstyle classes
  - Clean and refactor code for testing cpyext to allow sharing with py3.5
  - Refactor a building the map of reflected ops in _numpypy
  - Improve merging of virtual states in the JIT in order to avoid jumping to the preamble
  - In JIT residual calls, if the called function starts with a fast-path like if x.foo != 0: return x.foo, then inline the check before doing the CALL.
  - Ensure make_inputargs fails properly when given arguments with type information
  - Makes optimiseopt iterative instead of recursive so it can be reasoned about more easily and debugging is faster
  - Refactor and remove dead code from optimiseopt, resume

Please update, and continue to help us make PyPy better.

Cheers

PyPy 5.4.1

We have released a bugfix for PyPy2.7-v5.4.0, released last week, due to the following issues:

- Update list of contributors in documentation and LICENSE file, this was unfortunately left out of 5.4.0. My apologies to the new contributors
- Allow tests run with -A to find libm.so even if it is a script not a dynamically loadable file
- Bump sys.setrecursionlimit() when translating PyPy, for translating with CPython
- Tweak a float comparison with 0 in backendopt.inline to avoid rounding errors
- Fix for an issue for translating the sandbox
- Fix for and issue where unicode.decode('utf8', 'custom_replace') messed up the last byte of a unicode string sometimes
- Update built-in cffi to version 1.8.1
- Explicitly detect that we found as-yet-unsupported OpenSSL 1.1, and crash translation with a message asking for help porting it
- Fix a regression where a PyBytesObject was forced (converted to a RPython object) when not required, reported as issue #2395
Thanks to those who reported the issues.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast ([PyPy and CPython 2.7.x performance comparison](#)) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports:

- **x86** machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows 32, OpenBSD, FreeBSD),
- newer **ARM** hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of **PPC64** running Linux,
- **s390x** running Linux

Please update, and continue to help us make PyPy better.

Cheers

The PyPy Team

**PyPy2.7 v5.4**

We have released PyPy2.7 v5.4, a little under two months after PyPy2.7 v5.3. This new PyPy2.7 release includes incremental improvements to our C-API compatibility layer (cpyext), enabling us to pass over 99% of the upstream numpy test suite. We updated built-in cffi support to version 1.8, which now supports the “limited API” mode for c-extensions on CPython >=3.2.

We improved tooling for the PyPy JIT, and expanded VMProf support to OpenBSD and Dragon Fly BSD.

As always, this release fixed many issues and bugs raised by the growing community of PyPy users. We strongly recommend updating.

You can download the PyPy2.7 v5.4 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast ([PyPy and CPython 2.7.x performance comparison](#)) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports:

- **x86** machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
- newer **ARM** hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
• big- and little-endian variants of **PPC64** running Linux,
• **s390x** running Linux

**Other Highlights (since 5.3 released in June 2016)**

**New features:**

- Add `sys.{get,set}dlopenflags`
- Improve CPython compatibility of ‘is’ for small and empty strings
- Support for `rgc.FinalizerQueue` in the Boehm garbage collector
- (RPython) support `spawnv()` if it is called in C `_spawnv` on windows
- Fill in more slots when creating a `PyTypeObject` from a `W_TypeObject`, like `__hex__, __sub__, __pow__`
- Copy CPython’s logic more closely for `isinstance()` and `issubclass()` as well as `type.__instancecheck__()` and `type.__subclasscheck__()`
- Expose the name of CDLL objects
- Rewrite the win32 dependencies of `subprocess` to use `cffi` instead of `ctypes`
- Improve the **JIT logging** facilities
- (RPython) make `int * string` work
- Allocate all RPython strings with one extra byte, normally unused. This now allows `ffi.from_buffer(string)` in CFFI with no copy
- Adds a new commandline option `-X track-resources` that will produce a `ResourceWarning` when the GC closes a file or socket. The traceback for the place where the file or socket was allocated is given as well, which aids finding places where `close()` is missing
- Add missing `PyObject_Realloc`, `PySequence_GetSlice`
- `type.__dict__` now returns a `dict_proxy` object, like on CPython. Previously it returned what looked like a regular dict object (but it was already read-only)
- (RPython) add `rposix.{get,set}_inheritable()`, needed by Python 3.5
- (RPython) add `rposix_scandir` portably, needed for Python 3.5
- Increased but incomplete support for memoryview attributes (format, itemsize, ...) which also adds support for `PyMemoryView_FromObject`

**Bug Fixes**

- Reject `mkdir()` in read-only sandbox filesystems
- Add include guards to `pymem.h` to enable c++ compilation
- Fix build breakage on OpenBSD and FreeBSD
- Support OpenBSD, Dragon Fly BSD in VMProf
- Fix for `bytearray()` .replace(‘a’, ‘ab’) for empty strings
- Sync internal state before calling `PyFile_AsFile()`
- Allow writing to a char* from `PyString_AsString()` until it is forced, also refactor `PyStringObject` to look like CPython’s and allow subclasing `PyString_Type` and `PyUnicode_Type`
- Rpython rffi’s socket(2) wrapper did not preserve errno
Refactor *PyTupleObject* to look like CPython’s and allow subclassing *PyTuple_Type*

Allow c-level assignment to a function pointer in a C-API user-defined type after calling PyTypeReady by retrieving a pointer to the function via offsets rather than storing the function pointer itself

Use `madvise(MADV_FREE)`, or if that doesn’t exist `MADV_DONTNEED` on freed arenas to release memory back to the OS for resource monitoring

Fix overflow detection in conversion of float to 64-bit integer in timeout argument to various thread/threading primitives

Fix win32 outputting `rrn` in some cases

Make `hash(-1)` return -2, as CPython does, and fix all the ancillary places this matters

Fix `PyNumber_Check()` to behave more like CPython

(VMProf) Try hard to not miss any Python-level frame in the captured stacks, even if there is metainterpid or blackhole interp involved. Also fix the stacklet (greenlet) support

Fix a critical JIT bug where `raw_malloc`-equivalent functions lost the additional flags

Fix the mapdict cache for subclasses of builtin types that provide a dict

Issues reported with our previous release were resolved after reports from users on our issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy

• Performance improvements:

  Add a before_call()-like equivalent before a few operations like `malloc_nursery`, to move values from registers into other registers instead of to the stack.

  More tightly pack the stack when calling with release gil

  Support `int_floordiv()`, `int_mod()` in the JIT more efficiently and add `rarithmetic.int_c_div()`, `rarithmetic.int_c_mod()` as explicit interfaces. Clarify that `int_floordiv()` does python-style rounding, unlike `llop.int_floordiv()`.

  Use `ll_assert` (more often) in incminimark

  (Testing) Simplify handling of interp-level tests and make it more forward-compatible. Don’t use interp-level RPython machinery to test building app-level extensions in cpyext

  Constant-fold `ffi.offsetof(“structname”, “fieldname”)` in cffi backend

  Avoid a case in the JIT, where successive guard failures in the same Python function end up as successive levels of RPython functions, eventually exhausting the stack, while at app-level the traceback is very short

  Check for NULL returns from calls to the raw-malloc and raise, rather than a guard

  Improve `socket.recvfrom()` so that it copies less if possible

  When generating C code, inline `goto` to blocks with only one predecessor, generating less lines of code

  When running the final backend-optimization phase before emitting C code, constant-fold calls to `we_are_jitted` to return False. This makes the generated C code a few percent smaller

  Refactor the `uid_t/gid_t` handling in `rlib.rposix` and in `interp_posix.py`, based on the clean-up of CPython 2.7.x

Please update, and continue to help us make PyPy better.

Cheers
PyPy 5.3.1

We have released a bugfix for PyPy2.7-v5.3.0, released last week, due to issues reported by users. Thanks to those who reported the issues.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports:
- x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows 32, OpenBSD, FreeBSD),
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Please update, and continue to help us make PyPy better.

Cheers

The PyPy Team

PyPy2.7 v5.3

We have released PyPy2.7 v5.3, about six weeks after PyPy 5.1 and a week after PyPy3.3 v5.2 alpha 1, the first PyPy release targetting 3.3 compatibility. This new PyPy2.7 release includes further improvements for the CAPI compatibility layer which we call cpyext. In addition to complete support for lxml, we now pass most (more than 90%) of the upstream numpy test suite, and much of SciPy is supported as well.

We updated cffi to version 1.7 (small changes, documented here).

You can download the PyPy2.7 v5.3 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports:
- x86 machines on most common operating systems (Linux 32/64 bits, Mac OS X 64 bits, Windows 32 bits, OpenBSD, FreeBSD)
PyPy, Release 7.2.0

- newer **ARM** hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of **PPC64** running Linux,
- **s390x** running Linux

**Other Highlights (since 5.1 released in April 2016)**

- New features:
  - Merge a major expansion of the C-API support in cpyext, here are some of the highlights:
    - allow c-snippet tests to be run with -A so we can verify we are compatible
    - fix many edge cases exposed by fixing tests to run with -A
    - issequence() logic matches cpython
    - make PyStringObject and PyUnicodeObject field names compatible with cpython
    - add preliminary support for PyDateTime_*
    - support PyComplexObject, PyFloatObject, PyDict_Merge, PyDictProxy, PyMemoryView_*, _Py_HashDouble, PyFile_AsFile, PyFile_FromFile, PyAnySet_CheckExact, PyUnicode_Concat, PyDateTime_TZInfo
    - improve support for PyGILState_Ensure, PyGILState_Release, and thread primitives, also find a case where CPython will allow thread creation before PyEval_InitThreads is run, dissallow on PyPy
    - create a PyObject-specific list strategy
    - rewrite slot assignment for typeobjects
    - improve tracking of PyObject to rpython object mapping
    - support tp_as_{number, sequence, mapping, buffer} slots
    - support ByteArrayObject via the new resizable_list_supporting_raw_ptr
    - implement PyList_SET_ITEM with CPython’s behavior, instead of SetItem’s
    - fix the signature of PyUFunc_FromFuncAndDataAndSignature
    - implement many PyWhatever_FOO() as a macro taking a void *
  - CPyExt tweak: instead of “GIL not held when a CPython C extension module calls PyXxx”, we now silently acquire/release the GIL. Helps with CPython C extension modules that call some PyXxx() functions without holding the GIL (arguably, they are theoretically buggy).
  - Add rgc.FinalizerQueue, documented in pypy/doc/discussion/finalizer-order.rst. It is a more flexible way to make RPython finalizers. Use this mechanism to clean up handling of __del__ methods, fixing issue #2287
  - Generalize cpyext old-style buffers to more than just str/buffer, add support for mmap
  - Support command line -v to trace import statements
  - Add rposix functions for PyPy3.3 support
  - Give super an __init__ and a simple __new__ for CPython compatibility
  - Revive traceviewer, a tool to use pygame to view traces

- Bug Fixes
– Fix issue #2277: only special-case two exact lists in zip(), not list subclasses, because an overridden 
  __iter__() should be called (probably)
– Fix issue #2226: Another tweak in the incremental GC- this should ensure that progress in the major GC 
  occurs quickly enough in all cases.
– Clarify and refactor documentation on http://doc.pypy.org
– Use “must be unicode, not %T” in unicodedata TypeErrors.
– Manually reset sys.settrace() and sys.setprofile() when we’re done running. This is not exactly what 
  CPython does, but if we get an exception, unlike CPython, we call functions from the ‘traceback’ module, 
  and these would call more the trace/profile function. That’s unexpected and can lead to more crashes at 
  this point.
– Use the appropriate tp_dealloc on a subclass of a builtin type, and call tp_new for a python-subclass of a 
  C-API type
– Fix for issue #2285 - rare vmprof segfaults on OS/X
– Fixed issue #2172 - where a test specified an invalid parameter to mmap on powerpc
– Fix issue #2311 - grab the __future__ flags imported in the main script, in -c, or in PYTHON_STARTUP, 
  and expose them to the -i console
– Issues reported with our previous release were resolved after reports from users on our issue tracker at 
  https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy

• Numpy:
  – Implement ufunc.outer on numpypy
  – Move PyPy-specific numpy headers to a subdirectory (also changed the repo accordingly)
• Performance improvements:
  – Use bitstrings to compress lists of descriptors that are attached to an EffectInfo
  – Remove most of the _ovf, _zer and _val operations from RPython. Kills quite some code internally, and 
    allows the JIT to do better optimizations: for example, app-level code like \[x / 2\] or \[x \% 2\] can now be 
    turned into \[x >> 1\] or \[x \& 1\], even if x is possibly negative.
  – Copy CPython’s ‘optimization’: ignore __iter__ etc. for \[f(**dict_subclass())\]
  – Use the __builtin_add_overflow built-ins if they are available
  – Rework the way registers are moved/spilled in before_call()
• Internal refactorings:
  – Refactor code to better support Python3-compatible syntax
  – Document and refactor OperationError -> oefmt
  – Reduce the size of generated C sources during translation by eliminating many many unused struct decla- 
      rations (Issue #2281)
  – Remove a number of translation-time options that were not tested and never used. Also fix a performance 
    bug in the method cache
  – Reduce the size of generated code by using the same function objects in all generated subclasses
  – Share cpyext Py* function wrappers according to the signature, shrinking the translated libpypy.so by 
    about 10% (measured without the JIT)
  – Compile c snippets with -Werror, and fix warnings it exposed
PyPy, Release 7.2.0

Please update, and continue to help us make PyPy better.

Cheers

The PyPy Team

PyPy 5.1.1

We have released a bugfix for PyPy 5.1, due to a regression in installing third-party packages dependant on numpy (using our numpy fork available at https://bitbucket.org/pypy/numpy).

Thanks to those who reported the issue. We also fixed a regression in translating PyPy which increased the memory required to translate. Improvement will be noticed by downstream packagers and those who translate rather than download pre-built binaries.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports:

- x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows 32, OpenBSD, FreeBSD),
- newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
- big- and little-endian variants of PPC64 running Linux,
- s390x running Linux

Please update, and continue to help us make PyPy better.

Cheers

The PyPy Team

PyPy 5.1

We have released PyPy 5.1, about a month after PyPy 5.0.

This release includes more improvement to warmup time and memory requirements. We have seen about a 20% memory requirement reduction and up to 30% warmup time improvement, more detail in the blog post.

We also now have fully support for the IBM s390x. Since this support is in RPython, any dynamic language written using RPython, like PyPy, will automagically be supported on that architecture.

We updated cffi to 1.6, and continue to improve support for the wider python ecosystem using the PyPy interpreter.

You can download the PyPy 5.1 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.
What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports:

• x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows 32, OpenBSD, FreeBSD),
• newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
• big- and little-endian variants of PPC64 running Linux,
• s390x running Linux

Other Highlights (since 5.0 released in March 2015)

• New features:
  – A new jit backend for the IBM s390x, which was a large effort over the past few months.
  – Add better support for PyUnicodeObject in the C-API compatibility layer
  – Support GNU/kFreeBSD Debian ports in vmprof
  – Add __pypy__.promote
  – Make attrgetter a single type for CPython compatibility

• Bug Fixes
  – Catch exceptions raised in an exit function
  – Fix a corner case in the JIT
  – Fix edge cases in the cpyext refcounting-compatible semantics (more work on cpyext compatibility is coming in the cpyext-ext branch, but isn’t ready yet)
  – Try harder to not emit NEON instructions on ARM processors without NEON support
  – Improve the rpython posix module system interaction function calls
  – Detect a missing class function implementation instead of calling a random function
  – Check that PyTupleObjects do not contain any NULLs at the point of conversion to W_TupleObjects
  – In cotypes, fix _anonymous_ fields of instances
  – Fix JIT issue with unpack() on a Trace which contains half-written operations
  – Fix sandbox startup (a regression in 5.0)
  – Fix possible segfault for classes with mangled mro or __metaclass__
  – Fix istinstance(deque(), Hashable) on the pure python deque
  – Fix an issue with forkpty()
  – Issues reported with our previous release were resolved after reports from users on our issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy

• Numpy:
  – Implemented numpy.where for a single argument
– Indexing by a numpy scalar now returns a scalar
– Fix transpose(arg) when arg is a sequence
– Refactor include file handling, now all numpy ndarray, ufunc, and umath functions exported from libpypy.so are declared in pypy_numpy.h, which is included only when building our fork of numpy
– Add broadcast

• Performance improvements:
  – Improve str.endswith([tuple]) and str.startswith([tuple]) to allow JITting
  – Merge another round of improvements to the warmup performance
  – Cleanup history rewriting in pyjitpl
  – Remove the forced minor collection that occurs when rewriting the assembler at the start of the JIT backend
  – Port the resource module to cffi

• Internal refactorings:
  – Use a simpler logger to speed up translation
  – Drop vestiges of Python 2.5 support in testing
  – Update rpython functions with ones needed for py3k

Please update, and continue to help us make PyPy better.

Cheers
The PyPy Team

**PyPy 5.0.1**

We have released a bugfix for PyPy 5.0, after reports that the newly released lxml 3.6.0, which now supports PyPy 5.0 +, can crash on large files. Thanks to those who reported the crash. Please update, downloads are available at pypy.org/download.html

The changes between PyPy 5.0 and 5.0.1 are only two bug fixes: one in cpyext, which fixes notably (but not only) lxml; and another for a corner case of the JIT.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows 32, OpenBSD, FreeBSD), newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux, and the big- and little-endian variants of PPC64 running Linux.

Please update, and continue to help us make PyPy better.

Cheers
The PyPy Team
**PyPy 5.0**

We have released PyPy 5.0, about three months after PyPy 4.0.1. We encourage all users of PyPy to update to this version. Apart from the usual bug fixes, there is an ongoing effort to improve the warmup time and memory usage of JIT-related metadata. The exact effects depend vastly on the program you’re running and can range from insignificant to warmup being up to 30% faster and memory dropping by about 30%.

We also merged a major upgrade to our C-API layer (cpyext), simplifying the interaction between c-level objects and PyPy interpreter level objects. As a result, lxml (prerelease) with its cython compiled component passes all tests on PyPy. The new cpyext is also much faster.

**vmprof** has been a go-to profiler for PyPy on linux for a few releases and we’re happy to announce that thanks to the cooperation with jetbrains, vmprof now works on Linux, OS X and Windows on both PyPy and CPython.

You can download the PyPy 5.0 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

**CFFI**

While not applicable only to PyPy, **cffi** is arguably our most significant contribution to the python ecosystem. PyPy 5.0 ships with **cffi-1.5.2** which now allows embedding PyPy (or cpython) in a C program.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (PyPy and CPython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other **dynamic languages** to see what RPython can do for them.

This release supports **x86** machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows 32, OpenBSD, FreeBSD), newer **ARM** hardware (ARMv6 or ARMv7, with VFPv3) running Linux, and the big- and little-endian variants of **PPC64** running Linux.

**Other Highlights (since 4.0.1 released in November 2015)**

- New features:
  - Support embedding PyPy in a C-program via cffi and static callbacks in cffi. This deprecates the old method of embedding PyPy
  - Refactor vmprof to work cross-operating-system, deprecate using buggy libunwind on Linux platforms. Vmprof even works on Windows now.
  - Support more of the C-API type slots, like tp_getattro, and fix C-API macros, functions, and structs such as _PyLong_FromByteArray(), PyString_GET_SIZE, f_locals in PyFrameObject, Py_NAN, co_filename in PyCodeObject
  - Use a more stable approach for allocating PyObjects in cpyext. (see [blog post](http://pypy.org/download.html)). Once the PyObject corresponding to a PyPy object is created, it stays around at the same location until the death of the PyPy object. Done with a little bit of custom GC support. It allows us to kill the notion of “borrowing” inside
cpyext, reduces 4 dictionaries down to 1, and significantly simplifies the whole approach (which is why it is a new feature while technically a refactoring) and allows PyPy to support the popular lxml module (as of the next release) with no PyPy specific patches needed

- Make the default filesystem encoding ASCII, like CPython
- Use hypothesis in test creation, which is great for randomizing tests

**Bug Fixes**

- Backport always using os.urandom for uuid4 from cpython and fix the JIT as well (issue #2202)
- More completely support datetime, optimize timedelta creation
- Fix for issue #2185 which caused an inconsistent list of operations to be generated by the unroller, appeared in a complicated DJango app
- Fix an elusive issue with stacklets on shadowstack which showed up when forgetting stacklets without resuming them
- Fix entrypoint() which now acquires the GIL
- Fix direct ffi call() so failure does not bail out before setting CALL_MAY_FORCE
- Fix (de)pickling long values by simplifying the implementation
- Fix RPython rthread so that objects stored as threadlocal do not force minor GC collection and are kept alive automatically. This improves performance of short-running Python callbacks and prevents resetting such object between calls
- Support floats as parameters to itertools.isslice()
- Check for the existence of CODESET, ignoring it should have prevented PyPy from working on FreeBSD
- Fix for corner case (likely shown by Krakatau) for consecutive guards with interdependencies
- Fix applevel bare class method comparisons which should fix pretty printing in IPython
- Issues reported with our previous release were resolved after reports from users on our issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy

**Numpy:**

- Updates to numpy 1.10.2 (incompatibilities and not-implemented features still exist)
- Support dtype=(‘O’, spec) union while disallowing record arrays with mixed object, non-object values
- Remove all traces of micronumpy from cpyext if –withoutmod-micronumpy option used
- Support indexing filtering with a boolean ndarray
- Support partition() as an app-level function, together with a cffi wrapper in pypy/numpy, this now provides partial support for partition()

**Performance improvements:**

- Optimize global lookups
- Improve the memory signature of numbering instances in the JIT. This should massively decrease the amount of memory consumed by the JIT, which is significant for most programs. Also compress the numberings using variable-size encoding
- Optimize string concatenation
- Use INT_LSHIFT instead of INT_MUL when possible
PyPy, Release 7.2.0

- Improve struct.unpack by casting directly from the underlying buffer. Unpacking floats and doubles is about 15 times faster, and integer types about 50% faster (on 64 bit integers). This was then subsequently improved further in optimizeopt.py.
- Optimize two-tuple lookups in mapdict, which improves warmup of instance variable access somewhat
- Reduce all guards from int_floordiv_ovf if one of the arguments is constant
- Identify permutations of attributes at instance creation, reducing the number of bridges created
- Greatly improve re.sub() performance

Internal refactorings:
- Refactor and improve exception analysis in the annotator
- Remove unnecessary special handling of space.wrap().
- Support list-resizing setslice operations in RPython
- Tweak the trace-too-long heuristic for multiple jit drivers
- Refactor bookkeeping (such a cool word - three double letters) in the annotater
- Refactor wrappers for OS functions from rtyper to rlib and simplify them
- Simplify backend loading instructions to only use four variants
- Simplify GIL handling in non-jitted code
- Refactor naming in optimizeopt
- Change GraphAnalyzer to use a more precise way to recognize external functions and fix null pointer handling, generally clean up external function handling
- Remove pure variants of getfield_gc_* operations from the JIT by determining purity while tracing
- Refactor databasing
- Simplify bootstrapping in cpyext
- Refactor rtyper debug code into python.rtyper.debug
- Separate structmember.h from Python.h Also enhance creating api functions to specify which header file they appear in (previously only pypy_decl.h)
- Fix tokenizer to enforce universal newlines, needed for Python 3 support

Please update, and continue to help us make PyPy better.

Cheers

The PyPy Team

PyPy 4.0.1

We have released PyPy 4.0.1, three weeks after PyPy 4.0.0. We have fixed a few critical bugs in the JIT compiled code, reported by users. We therefore encourage all users of PyPy to update to this version. There are a few minor enhancements in this version as well.

You can download the PyPy 4.0.1 release here:

http://pypy.org/download.html

4.1. Historical release notes
We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

CFFI

While not applicable only to PyPy, cffi is arguably our most significant contribution to the python ecosystem. PyPy 4.0.1 ships with cffi-1.3.1 with the improvements it brings.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy and cpython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows 32, OpenBSD, freebsd), newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux, and the big- and little-endian variants of ppc64 running Linux.

Other Highlights (since 4.0.0 released three weeks ago)

• Bug Fixes
  – Fix a bug when unrolling double loops in JITted code
  – Fix multiple memory leaks in the ssl module, one of which affected cpython as well (thanks to Alex Gaynor for pointing those out)
  – Use pkg-config to find ssl headers on OS-X
  – Issues reported with our previous release were resolved after reports from users on our issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy

• New features:
  – Internal cleanup of RPython class handling
  – Support stackless and greenlets on PPC machines
  – Improve debug logging in subprocesses: use PYPYLOG=jit:log.%d for example to have all subprocesses write the JIT log to a file called ‘log.%d’, with ‘%d’ replaced with the subprocess’ PID.
  – Support PyOS_double_to_string in our cpyext capi compatibility layer

• Numpy:
  – Improve support for __array_interface__
  – Propagate NAN mantissas through float16-float32-float64 conversions

• Performance improvements and refactorings:
  – Improvements in slicing byte arrays
  – Improvements in enumerate()
  – Silence some warnings while translating
Please update, and continue to help us make PyPy better.

Cheers

The PyPy Team

PyPy 4.0.0

We’re pleased and proud to unleash PyPy 4.0.0, a major update of the PyPy python 2.7.10 compatible interpreter with a Just In Time compiler. We have improved warmup time and memory overhead used for tracing, added vectorization for numpy and general loops where possible on x86 hardware (disabled by default), refactored rough edges in rpython, and increased functionality of numpy.

You can download the PyPy 4.0.0 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project.

We would also like to thank our contributors (7 new ones since PyPy 2.6.0) and encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

New Version Numbering

Since the past release, PyPy 2.6.1, we decided to update the PyPy 2.x.x versioning directly to PyPy 4.x.x, to avoid confusion with CPython 2.7 and 3.5. Note that this version of PyPy uses the stdlib and implements the syntax of CPython 2.7.10.

Vectorization

Richard Plangger began work in March and continued over a Google Summer of Code to add a vectorization step to the trace optimizer. The step recognizes common constructs and emits SIMD code where possible, much as any modern compiler does. This vectorization happens while tracing running code, so it is actually easier at run-time to determine the availability of possible vectorization than it is for ahead-of-time compilers.

Availability of SIMD hardware is detected at run time, without needing to precompile various code paths into the executable.

The first version of the vectorization has been merged in this release, since it is so new it is off by default. To enable the vectorization in built-in JIT drivers (like numpy ufuncs), add –jit vec=1, to enable all implemented vectorization add –jit vec_all=1

Benchmarks and a summary of this work appear here

Internal Refactoring: Warmup Time Improvement and Reduced Memory Usage

Maciej Fijalkowski and Armin Rigo refactored internals of rpython that now allow PyPy to more efficiently use guards in jitted code. They also rewrote unrolling, leading to a warmup time improvement of 20% or so. The reduction in guards also means a reduction in the use of memory, also a savings of around 20%. 

4.1. Historical release notes
Numpy

Our implementation of numpy continues to improve. ndarray and the numeric dtypes are very close to feature-complete; record, string and unicode dtypes are mostly supported. We have reimplemented numpy linalg, random and fft as cffi-1.0 modules that call out to the same underlying libraries that upstream numpy uses. Please try it out, especially using the new vectorization (via --jit vec=1 on the command line) and let us know what is missing for your code.

CFFI

While not applicable only to PyPy, cffi is arguably our most significant contribution to the python ecosystem. Armin Rigo continued improving it, and PyPy reaps the benefits of cffi-1.3: improved management of object lifetimes, __stdcall on Win32, ffi.memmove(), and percolate const, restrict keywords from cdef to C code.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy and cpython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows 32, OpenBSD, freebsd), as well as newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux.

We also introduce support for the 64 bit PowerPC hardware, specifically Linux running the big- and little-endian variants of ppc64.

Other Highlights (since 2.6.1 release two months ago)

- Bug Fixes
  - Applied OPENBSD downstream fixes
  - Fix a crash on non-linux when running more than 20 threads
  - In cffi, ffi.new_handle() is more cpython compliant
  - Accept unicode in functions inside the _curses cffi backend exactly like cpython
  - Fix a segfault in itertools.islice()
  - Use gcrootfinder=shadowstack by default, asmgcc on linux only
  - Fix ndarray.copy() for upstream compatibility when copying non-contiguous arrays
  - Fix assumption that lltype.UniChar is unsigned
  - Fix a subtle bug with stacklets on shadowstack
  - Improve support for the cpython capi in cpyext (our capi compatibility layer). Fixing these issues inspired some thought about cpyext in general, stay tuned for more improvements
  - When loading dynamic libraries, in case of a certain loading error, retry loading the library assuming it is actually a linker script, like on Arch and Gentoo
  - Issues reported with our previous release were resolved after reports from users on our issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy

- New features:
PyPy, Release 7.2.0

- Add an optimization pass to vectorize loops using x86 SIMD intrinsics.
- Support __stdcall on Windows in CFFI
- Improve debug logging when using PYPYLOG=???
- Deal with platforms with no RAND_egd() in OpenSSL

**Numpy:**
- Add support for ndarray.ctypes
- Fast path for mixing numpy scalars and floats
- Add support for creating Fortran-ordered ndarrays
- Fix casting failures in linalg (by extending ufunc casting)
- Recognize and disallow (for now) pickling of ndarrays with objects embedded in them

**Performance improvements and refactorings:**
- Reuse hashed keys across dictionaries and sets
- Refactor JIT internals to improve warmup time by 20% or so at the cost of a minor regression in JIT speed
- Recognize patterns of common sequences in the JIT backends and optimize them
- Make the garbage collector more incremental over external_malloc() calls
- Share guard resume data where possible which reduces memory usage
- Fast path for zip(list, list)
- Reduce the number of checks in the JIT for lst[a:]
- Move the non-optimizable part of callbacks outside the JIT
- Factor in field immutability when invalidating heap information
- Unroll itertools.izip_longest() with two sequences
- Minor optimizations after analyzing output from vmprof and trace logs
- Remove many class attributes in rpython classes
- Handle getfield_gc_pure* and getfield_gc_* uniformly in heap.py
- Improve simple trace function performance by lazily calling fast2locals and locals2fast only if truly necessary

Please try it out and let us know what you think. We welcome feedback, we know you are using PyPy, please tell us about it!

Cheers

The PyPy Team

**PyPy 2.6.1**

We’re pleased to announce PyPy 2.6.1, an update to PyPy 2.6.0 released June 1. We have updated stdlib to 2.7.10, cffi to version 1.3, extended support for the new vmprof statistical profiler for multiple threads, and increased functionality of numpy.

You can download the PyPy 2.6.1 release here:

http://pypy.org/download.html

4.1. Historical release notes
We would like to thank our donors for the continued support of the PyPy project, and our volunteers and contributors. We would also like to encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast ([pypy and cpython 2.7.x performance comparison](https://www.pypy.org/performance.html)) due to its integrated tracing JIT compiler.

This release supports **x86** machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows 32, OpenBSD, freebsd), as well as newer **ARM** hardware (ARMv6 or ARMv7, with VFPv3) running Linux.

We also welcome developers of other [dynamic languages](https://en.wikipedia.org/wiki/Dynamic_programming_language) to see what RPython can do for them.

**Highlights**

- **Bug Fixes**
  - Revive non-SSE2 support
  - Fixes for detaching `_io.Buffer`*
  - On Windows, close (and flush) all open sockets on exiting
  - Drop support for ancient macOS v10.4 and before
  - Clear up contention in the garbage collector between trace-me-later and pinning
  - Issues reported with our previous release were resolved after reports from users on our issue tracker at [https://bitbucket.org/pypy/pypy/issues](https://bitbucket.org/pypy/pypy/issues) or on IRC at #pypy.

- **New features:**
  - cffi was updated to version 1.3
  - The python stdlib was updated to 2.7.10 from 2.7.9
  - vmprof now supports multiple threads and OS X
  - The translation process builds cffi import libraries for some stdlib packages, which should prevent confusion when package.py is not used
  - better support for gdb debugging
  - freebsd should be able to translate PyPy “out of the box” with no patches

- **Numpy:**
  - Better support for record dtypes, including the `align` keyword
  - Implement casting and create output arrays accordingly (still missing some corner cases)
  - Support creation of unicode ndarrays
  - Better support `ndarray.flags`
  - Support `axis` argument in more functions
  - Refactor array indexing to support ellipses
  - Allow the docstrings of built-in numpy objects to be set at run-time
Support the buffered nditer creation keyword

Performance improvements:

- Delay recursive calls to make them non-recursive
- Skip loop unrolling if it compiles too much code
- Tweak the heapcache
- Add a list strategy for lists that store both floats and 32-bit integers. The latter are encoded as nonstandard NaNs. Benchmarks show that the speed of such lists is now very close to the speed of purely-int or purely-float lists.
- Simplify implementation of ffi.gc() to avoid most weakrefs
- Massively improve the performance of map() with more than one sequence argument

Please try it out and let us know what you think. We welcome success stories, experiments, or benchmarks, we know you are using PyPy, please tell us about it!

Cheers
The PyPy Team

**PyPy 2.6.0 - Cameo Charm**

We’re pleased to announce PyPy 2.6.0, only two months after PyPy 2.5.1. We are particulary happy to update cffi to version 1.1, which makes the popular ctypes-alternative even easier to use, and to support the new vmprof statistical profiler.

You can download the PyPy 2.6.0 release here:

[http://pypy.org/download.html](http://pypy.org/download.html)

We would like to thank our donors for the continued support of the PyPy project, and for those who donate to our three sub-projects, as well as our volunteers and contributors.

Thanks also to Yury V. Zaytsev and David Wilson who recently started running nightly builds on Windows and MacOSX buildbots.

We’ve shown quite a bit of progress, but we’re slowly running out of funds. Please consider donating more, or even better convince your employer to donate, so we can finish those projects! The three sub-projects are:

- **Py3k** (supporting Python 3.x): We have released a Python 3.2.5 compatible version we call PyPy3 2.4.0, and are working toward a Python 3.3 compatible version
- **STM** (software transactional memory): We have released a first working version, and continue to try out new promising paths of achieving a fast multithreaded Python
- **NumPy** which requires installation of our fork of upstream numpy, available [on bitbucket](https://bitbucket.org/pypy/numpy)

We would also like to encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and RPython documentation improvements, tweaking popular modules to run on pypy, or general help with making RPython’s JIT even better. Nine new people contributed since the last release, you too could be one of them.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast ([pypy and cpython 2.7.x performance comparison](http://pypy.org/performance.html)) due to its integrated tracing JIT compiler.
This release supports x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows, OpenBSD, freebsd), as well as newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux.

While we support 32 bit python on Windows, work on the native Windows 64 bit python is still stalling, we would welcome a volunteer to handle that. We also welcome developers with other operating systems or dynamic languages to see what RPython can do for them.

**Highlights**

- Python compatibility:
  - Improve support for TLS 1.1 and 1.2
  - Windows downloads now package a pypyw.exe in addition to pypy.exe
  - Support for the PYTHONOPTIMIZE environment variable (impacting builtin’s __debug__ property)
  - Issues reported with our previous release were resolved after reports from users on our issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy.

- New features:
  - Add preliminary support for a new lightweight statistical profiler vmprof, which has been designed to accommodate profiling JITted code

- Numpy:
  - Support for object dtype via a garbage collector hook
  - Support for .can_cast and .min_scalar_type as well as beginning a refactoring of the internal casting rules
  - Better support for subtypes, via the __array_interface__, __array_priority__, and __array_wrap__ methods (still a work-in-progress)
  - Better support for ndarray.flags

- Performance improvements:
  - Slight improvement in frame sizes, improving some benchmarks
  - Internal refactoring and cleanups leading to improved JIT performance
  - Improved IO performance of zlib and bz2 modules
  - We continue to improve the JIT’s optimizations. Our benchmark suite is now over 7 times faster than cpython

Please try it out and let us know what you think. We welcome success stories, experiments, or benchmarks, we know you are using PyPy, please tell us about it!

Cheers

The PyPy Team

**PyPy 2.5.1 - Pineapple Bromeliad**

We’re pleased to announce PyPy 2.5.1, Pineapple Bromeliad following on the heels of 2.5.0

You can download the PyPy 2.5.1 release here:

http://pypy.org/download.html
We would like to thank our donors for the continued support of the PyPy project, and for those who donate to our three sub-projects, as well as our volunteers and contributors. We’ve shown quite a bit of progress, but we’re slowly running out of funds. Please consider donating more, or even better convince your employer to donate, so we can finish those projects! The three sub-projects are:

- **Py3k (supporting Python 3.x):** We have released a Python 3.2.5 compatible version we call PyPy3 2.4.0, and are working toward a Python 3.3 compatible version
- **STM (software transactional memory):** We have released a first working version, and continue to try out new promising paths of achieving a fast multithreaded Python
- **NumPy** which requires installation of our fork of upstream numpy, available on bitbucket

We would also like to encourage new people to join the project. PyPy has many layers and we need help with all of them: PyPy and Rpython documentation improvements, tweaking popular modules to run on pypy, or general help with making Rpython’s JIT even better.

### What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy and cpython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows, and OpenBSD), as well as newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux.

While we support 32 bit python on Windows, work on the native Windows 64 bit python is still stalling, we would welcome a volunteer to handle that.

### Highlights

- The past months have seen pypy mature and grow, as rpython becomes the goto solution for writing fast dynamic language interpreters. Our separation of Rpython and the python interpreter PyPy is now much clearer in the PyPy documentation and we now have separate RPython documentation. Tell us what still isn’t clear, or even better help us improve the documentation.
- We merged version 2.7.9 of python’s stdlib. From the python release notice:
  - The entirety of Python 3.4’s ssl module has been backported. See PEP 466 for justification.
  - HTTPS certificate validation using the system’s certificate store is now enabled by default. See PEP 476 for details.
  - SSLv3 has been disabled by default in httplib and its reverse dependencies due to the POODLE attack.
  - The ensurepip module has been backported, which provides the pip package manager in every Python 2.7 installation. See PEP 477.
- The garbage collector now ignores parts of the stack which did not change since the last collection, another performance boost
- errno and LastError are saved around cffi calls so things like pdb will not overwrite it
- We continue to asymptotically approach a score of 7 times faster than cpython on our benchmark suite, we now rank 6.98 on latest runs
- Issues reported with our previous release were resolved after reports from users on our issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy.
Please try it out and let us know what you think. We welcome success stories, experiments, or benchmarks, we know you are using PyPy, please tell us about it!

Cheers

The PyPy Team

**PyPy 2.5.0 - Pincushion Protea**

We’re pleased to announce PyPy 2.5, which contains significant performance enhancements and bug fixes.

You can download the PyPy 2.5.0 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project, and for those who donate to our three sub-projects, as well as our volunteers and contributors (10 new committers joined PyPy since the last release). We’ve shown quite a bit of progress, but we’re slowly running out of funds. Please consider donating more, or even better convince your employer to donate, so we can finish those projects! The three sub-projects are:

- **Py3k (supporting Python 3.x): We have released a Python 3.2.5 compatible version** we call PyPy3 2.4.0, and are working toward a Python 3.3 compatible version

- **STM (software transactional memory):** We have released a first working version, and continue to try out new promising paths of achieving a fast multithreaded Python

- **NumPy** which requires installation of our fork of upstream numpy, available on bitbucket

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast ([pypy and cpython 2.7.x performance comparison](http://pypy.org/download.html)) due to its integrated tracing JIT compiler.

This release supports x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows, and OpenBSD), as well as newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux.

While we support 32 bit python on Windows, work on the native Windows 64 bit python is still stalling, we would welcome a volunteer to handle that.

**Highlights**

- The past months have seen pypy mature and grow, as rpython becomes the goto solution for writing fast dynamic language interpreters. Our separation of rpython and the python interpreter PyPy is now much clearer in the PyPy documentation and we now have separate RPython documentation.

- We have improved warmup time as well as jitted code performance: more than 10% compared to pypy-2.4.0, due to internal cleanup and gc nursery improvements. We no longer zero-out memory allocated in the gc nursery by default, work that was started during a GSoC.

- Passing objects between C and PyPy has been improved. We are now able to pass raw pointers to C (without copying) using pinning. This improves I/O; benchmarks that use networking intensively improved by about 50%. File() operations still need some refactoring but are already showing a 20% improvement on our benchmarks. Let us know if you see similar improvements.

- Our integrated numpy support gained much of the GenericUfunc api in order to support the lapack/blas linalg module of numpy. This dovetails with work in the pypy/numpy repository to support linalg both through the (slower) cpyext capi interface and also via (the faster) pure python cffi interface, using an extended frompyfunc() api. We will soon post a seperate blog post specifically about linalg and PyPy.
• Dictionaries are now ordered by default, see the blog post

• Our nightly translations use –shared by default, including on OS/X and linux

• We now more carefully handle errno (and GetLastError, WSAGetLastError) tying the handlers as close as possible to the external function call, in non-jitted as well as jitted code.

• Issues reported with our previous release were resolved after reports from users on our issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy.

We have further improvements on the way: rpython file handling, finishing numpy linalg compatibility, numpy object dtypes, a better profiler, as well as support for Python stdlib 2.7.9.

Please try it out and let us know what you think. We especially welcome success stories, we know you are using PyPy, please tell us about it!

Cheers

The PyPy Team

**PyPy 2.4 - Snow White**

We’re pleased to announce PyPy 2.4, which contains significant performance enhancements and bug fixes.

You can download the PyPy 2.4.0 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project, and for those who donate to our three sub-projects. We’ve shown quite a bit of progress, but we’re slowly running out of funds. Please consider donating more, or even better convince your employer to donate, so we can finish those projects! We would like to also point out that in September, the Python Software Foundation will match funds for any donations up to $10k! The three sub-projects are:

• **Py3k (supporting Python 3.x):** We have released a Python 3.2.5 compatible version we call PyPy3 2.3.1, and are working toward a Python 3.3 compatible version

• STM (software transactional memory): We have released a first working version, and continue to try out new promising paths of achieving a fast multithreaded Python

• NumPy which requires installation of our fork of upstream numpy, available on bitbucket

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 2.4 and cpython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows, and OpenBSD), as well as newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux.

While we support 32 bit python on Windows, work on the native Windows 64 bit python is still stalling, we would welcome a volunteer to handle that.

**Highlights**

Benchmarks improved after internal enhancements in string and bytestring handling, and a major rewrite of the GIL handling. This means that external calls are now a lot faster, especially the CFFI ones. It also means better performance in a lot of corner cases with handling strings or bytestrings. The main bugfix is handling of many socket objects in your program which in the long run used to “leak” memory.
PyPy now uses Python 2.7.8 standard library.

We fixed a memory leak in IO in the sandbox code

We welcomed more than 12 new contributors, and conducted two Google Summer of Code projects, as well as other student projects not directly related to Summer of Code.

Issues reported with our previous release were fixed after reports from users on our new issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy. Here is a summary of the user-facing changes; for more information see whats-new:

- Reduced internal copying of bytearray operations
- Tweak the internal structure of StringBuilder to speed up large string handling, which becomes advantageous on large programs at the cost of slightly slower small benchmark type programs.
- Boost performance of thread-local variables in both unjitted and jitted code, this mostly affects errno handling on linux, which makes external calls faster.
- Move to a mixed polling and mutex GIL model that make mutlithreaded jitted code run much faster
- Optimize errno handling in linux (x86 and x86-64 only)
- Remove ctypes pythonapi and ctypes.PyDLL, which never worked on PyPy
- Fix performance regression on ufunc(<scalar>, <scalar>) in numpy
- Classes in the ast module are now distinct from structures used by the compiler, which simplifies and speeds up translation of our source code to the PyPy binary interpreter
- Upgrade stdlib from 2.7.5 to 2.7.8
- Win32 now links statically to zlib, expat, bzip, and openssl-1.0.1i. No more missing DLLs
- Many issues were resolved since the 2.3.1 release on June 8

We have further improvements on the way: rpython file handling, numpy linalg compatibility, as well as improved GC and many smaller improvements.

Please try it out and let us know what you think. We especially welcome success stories, we know you are using PyPy, please tell us about it!

Cheers

The PyPy Team

PyPy 2.3.1 - Terrestrial Arthropod Trap Revisited

We’re pleased to announce PyPy 2.3.1, a feature-and-bugfix improvement over our recent release last month.

This release contains several bugfixes and enhancements.

You can download the PyPy 2.3.1 release here:

  http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project, and for those who donate to our three sub-projects. We’ve shown quite a bit of progress but we’re slowly running out of funds. Please consider donating more, or even better convince your employer to donate, so we can finish those projects! The three sub-projects are:

- STM (software transactional memory): a preview will be released very soon, once we fix a few bugs
- NumPy which requires installation of our fork of upstream numpy, available on bitbucket
What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 2.3 and cpython 2.7.x performance comparison; note that cpython’s speed has not changed since 2.7.2) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64, Windows, and OpenBSD, as well as newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux.

While we support 32 bit python on Windows, work on the native Windows 64 bit python is still stalling, we would welcome a volunteer to handle that.

Highlights

Issues with the 2.3 release were resolved after being reported by users to our new issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy. Here is a summary of the user-facing changes; for more information see whats-new:

• The built-in struct module was renamed to _struct, solving issues with IDLE and other modules.
• Support for compilation with gcc-4.9
• A rewrite of packaging.py which produces our downloadable packages to modernize command line argument handling and to document third-party contributions in our LICENSE file
• A CFFI-based version of the gdbm module is now included in our downloads
• Many issues were resolved since the 2.3 release on May 8

Please try it out and let us know what you think. We especially welcome success stories, we know you are using PyPy, please tell us about it!

Cheers

The PyPy Team

PyPy 2.3 - Terrestrial Arthropod Trap

We’re pleased to announce PyPy 2.3, which targets version 2.7.6 of the Python language. This release updates the stdlib from 2.7.3, jumping directly to 2.7.6.

This release also contains several bugfixes and performance improvements, many generated by real users finding corner cases our TDD methods missed. CFFI has made it easier than ever to use existing C code with both cpython and PyPy, easing the transition for packages like cryptography, Pillow (Python Imaging Library [Fork]), a basic port of pygame-cffi, and others.

PyPy can now be embedded in a hosting application, for instance inside uWSGI

You can download the PyPy 2.3 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project, and for those who donate to our three sub-projects. We showed quite a bit of progress but we’re slowly running out of funds. Please consider donating more, or even better convince your employer to donate, so we can finish those projects! The three sub-projects are:

• Py3k (supporting Python 3.x): the release PyPy3 2.2 is imminent.
• STM (software transactional memory): a preview will be released very soon, once we fix a few bugs
• NumPy which is included in the PyPy 2.3 release. More details below.

4.1. Historical release notes
What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 2.3 and cpython 2.7.x performance comparison; note that cpython’s speed has not changed since 2.7.2) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64, Windows, and OpenBSD, as well as newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux.

While we support 32 bit python on Windows, work on the native Windows 64 bit python is still stalling, we would welcome a volunteer to handle that.

Highlights

Bugfixes

Many issues were cleaned up after being reported by users to https://bugs.pypy.org or on IRC at #pypy. Note that we consider performance slowdowns as bugs. Here is a summary of the user-facing changes; for more information see whats-new:

- The ARM port no longer crashes on unaligned memory access to floats and doubles, and singlefloats are supported in the JIT.
- Generators are faster since they now skip unnecessary cleanup
- A first time contributor simplified JIT traces by adding integer bound propagation in indexing and logical operations.
- Optimize consecutive dictionary lookups of the same key in a chain
- Our extensive pre-translation test suite now runs nightly on more platforms
- Fix issues with reimporting builtin modules
- Fix an RPython bug with loop-unrolling that appeared in the Hippy VM PHP port
- Support for corner cases on objects with __int__ and __float__ methods
- Fix multithreaded support for gethostbyname_ex and gethostbyaddr
- Fix handling of tp_name for type objects

New Platforms and Features

- Support for OpenBSD
- Code cleanup: we continue to prune out old and unused code, and to refactor large parts of the codebase. We have separated RPython from the PyPy python interpreter, and RPython is seeing use in other dynamic language projects.
- Support for precompiled headers in the build process for MSVC
- Tweak support of errno in cpyext (the PyPy implementation of the capi)

NumPy

NumPy support has been split into a builtin _numpy module and a fork of the NumPy code base adapted to PyPy at https://bitbucket.org/pypy/numpy. You need to install NumPy separately with a virtualenv:
pip install git+https://bitbucket.org/pypy/numpy.git;

or directly:

git clone https://bitbucket.org/pypy/numpy.git; cd numpy; pypy setup.py install.

- NumPy support has been improved, many failures in indexing, dtypes, and scalars were corrected. We are slowly approaching our goal of passing the NumPy test suite. We still do not support object or unicode ndarrays.
- Speed of iteration in dot() is now within 1.5x of the NumPy c implementation (without BLAS acceleration). Since the same array iterator is used throughout the _numpy module, speed increases should be apparent in all NumPy functionality.
- Most of the core functionality of nditer has been implemented.
- A cffi-based numpy.random module is available as a branch; it will be merged soon after this release.
- Enhancements to the PyPy JIT were made to support virtualizing the raw_store/raw_load memory operations used in NumPy arrays. Further work remains here in virtualizing the alloc_raw_storage when possible. This will allow scalars to have storages but still be virtualized when possible in loops.

Cheers
The PyPy Team

**PyPy 2.2.1 - Incrementalism.1**

We’re pleased to announce PyPy 2.2.1, which targets version 2.7.3 of the Python language. This is a bugfix release over 2.2.

You can download the PyPy 2.2.1 release here:

[http://pypy.org/download.html](http://pypy.org/download.html)

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast ([pypy 2.2 and cpython 2.7.2 performance comparison](http://pypy.org)) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64, Windows 32, or ARM (ARMv6 or ARMv7, with VFPv3).

Work on the native Windows 64 is still stalling, we would welcome a volunteer to handle that.

**Highlights**

This is a bugfix release. The most important bugs fixed are:

- an issue in sockets’ reference counting emulation, showing up notably when using the ssl module and calling makefile().
- Tkinter support on Windows.
- If sys.maxunicode==65535 (on Windows and maybe OS/X), the json decoder incorrectly decoded surrogate pairs.
- some FreeBSD fixes.

4.1. Historical release notes
Note that CFFI 0.8.1 was released. Both versions 0.8 and 0.8.1 are compatible with both PyPy 2.2 and 2.2.1.

Cheers, Armin Rigo & everybody

PyPy 2.2 - Incrementalism

We’re pleased to announce PyPy 2.2, which targets version 2.7.3 of the Python language. This release main highlight is the introduction of the incremental garbage collector, sponsored by the Raspberry Pi Foundation.

This release also contains several bugfixes and performance improvements.

You can download the PyPy 2.2 release here:

http://pypy.org/download.html

We would like to thank our donors for the continued support of the PyPy project. We showed quite a bit of progress on all three projects (see below) and we’re slowly running out of funds. Please consider donating more so we can finish those projects! The three projects are:

- Py3k (supporting Python 3.x): the release PyPy3 2.2 is imminent.
- STM (software transactional memory): a preview will be released very soon, as soon as we fix a few bugs
- NumPy: the work done is included in the PyPy 2.2 release. More details below.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 2.2 and cpython 2.7.2 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64, Windows 32, or ARM (ARMv6 or ARMv7, with VFPv3).

Work on the native Windows 64 is still stalling, we would welcome a volunteer to handle that.

Highlights

- Our Garbage Collector is now “incremental”. It should avoid almost all pauses due to a major collection taking place. Previously, it would pause the program (rarely) to walk all live objects, which could take arbitrarily long if your process is using a whole lot of RAM. Now the same work is done in steps. This should make PyPy more responsive, e.g. in games. There are still other pauses, from the GC and the JIT, but they should be on the order of 5 milliseconds each.
- The JIT counters for hot code were never reset, which meant that a process running for long enough would eventually JIT-compile more and more rarely executed code. Not only is it useless to compile such code, but as more compiled code means more memory used, this gives the impression of a memory leak. This has been tentatively fixed by decreasing the counters from time to time.
- NumPy has been split: now PyPy only contains the core module, called _numpypy. The numpy module itself has been moved to https://bitbucket.org/pypy/numpy and numpypy disappeared. You need to install NumPy separately with a virtualenv: pip install git+https://bitbucket.org/pypy/numpy.git; or directly: git clone https://bitbucket.org/pypy/numpy.git; cd numpy; pypy setup.py install.
- non-inlined calls have less overhead
- Things that use sys.set_trace are now JITted (like coverage)
- JSON decoding is now very fast (JSON encoding was already very fast)
• various buffer copying methods experience speedups (like list-of-ints to `int[]` buffer from cffi)

• We finally wrote (hopefully) all the missing `os.xxx()` functions, including `os.startfile()` on Windows and a handful of rare ones on Posix.

• `numpy` has a rudimentary C API that cooperates with `cpyext`

Cheers, Armin Rigo and Maciej Fijalkowski

**PyPy 2.1 - Considered ARMful**

We’re pleased to announce PyPy 2.1, which targets version 2.7.3 of the Python language. This is the first release with official support for ARM processors in the JIT. This release also contains several bugfixes and performance improvements.

You can download the PyPy 2.1 release here:

http://pypy.org/download.html

We would like to thank the Raspberry Pi Foundation for supporting the work to finish PyPy’s ARM support.

The first beta of PyPy3 2.1, targeting version 3 of the Python language, was just released, more details can be found here.

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 2.1 and cpython 2.7.2 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. This release also supports ARM machines running Linux 32bit - anything with `ARMv6` (like the Raspberry Pi) or `ARMv7` (like the Beagleboard, Chromebook, Cubieboard, etc.) that supports VFPv3 should work. Both hard-float armhf/gnueabihf and soft-float armel/gnueabi builds are provided. armhf builds for Raspbian are created using the Raspberry Pi custom cross-compilation toolchain based on gcc-arm-linux-gnueabihf and should work on ARMv6 and ARMv7 devices running Debian or Raspbian. armel builds are built using the gcc-arm-linux-gnueabi toolchain provided by Ubuntu and currently target ARMv7.

Windows 64 work is still stalling, we would welcome a volunteer to handle that.

**Highlights**

• JIT support for ARM, architecture versions 6 and 7, hard- and soft-float ABI

• Stacklet support for ARM

• Support for `os.statvfs` and `os.fstatvfs` on unix systems

• Improved logging performance

• Faster sets for objects

• Interpreter improvements

• During packaging, compile the CFFI based TK extension

• Pickling of numpy arrays and dtypes

• Subarrays for numpy

• Bugfixes to numpy
• Bugfixes to cffi and ctypes
• Bugfixes to the x86 stacklet support
• Fixed issue 1533: fix an RPython-level OverflowError for space.float_w(w_big_long_number).
• Fixed issue 1552: GreenletExit should inherit from BaseException.
• Fixed issue 1537: numppy __array_interface__
• Fixed issue 1238: Writing to an SSL socket in PyPy sometimes failed with a “bad write retry” message.

Cheers,
David Schneider for the PyPy team.

PyPy 2.1 beta 2

We’re pleased to announce the second beta of the upcoming 2.1 release of PyPy. This beta adds one new feature to the 2.1 release and contains several bugfixes listed below.

You can download the PyPy 2.1 beta 1 release here:
http://pypy.org/download.html

Highlights

• Support for os.statvfs and os.fstatvfs on unix systems.
• Fixed issue 1533: fix an RPython-level OverflowError for space.float_w(w_big_long_number).
• Fixed issue 1552: GreenletExit should inherit from BaseException.
• Fixed issue 1537: numppy __array_interface__
• Fixed issue 1238: Writing to an SSL socket in pypy sometimes failed with a “bad write retry” message.
• distutils: copy CPython’s implementation of customize_compiler, dont call split on environment variables, honour CFLAGS, CPPFLAGS, LDSHARED and LDFLAGS.
• During packaging, compile the CFFI tk extension.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.3. It’s fast due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. Also this release supports ARM machines running Linux 32bit - anything with ARMv6 (like the Raspberry Pi) or ARMv7 (like Beagleboard, Chromebook, Cubieboard, etc.) that supports VFPv3 should work.

Windows 64 work is still stalling, we would welcome a volunteer to handle that.

How to use PyPy?

We suggest using PyPy from a virtualenv. Once you have a virtualenv installed, you can follow instructions from pypy documentation on how to proceed. This document also covers other installation schemes.

Cheers, The PyPy Team.
PyPy 2.1 beta 1

We're pleased to announce the first beta of the upcoming 2.1 release of PyPy. This beta contains many bugfixes and improvements, numerous improvements to the numpy in pypy effort. The main feature being that the ARM processor support is not longer considered alpha level. We would like to thank the Raspberry Pi Foundation for supporting the work to finish PyPy's ARM support.

You can download the PyPy 2.1 beta 1 release here:

http://pypy.org/download.html

Highlights

- Bugfixes to the ARM JIT backend, so that ARM is now an officially supported processor architecture
- Stacklet support on ARM
- Interpreter improvements
- Various numpy improvements
- Bugfixes to cffi and ctypes
- Bugfixes to the stacklet support
- Improved logging performance
- Faster sets for objects

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.3. It's fast due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. Also this release supports ARM machines running Linux 32bit - anything with ARMv6 (like the Raspberry Pi) or ARMv7 (like Beagleboard, Chromebook, Cubieboard, etc.) that supports VFPv3 should work. Both hard-float armhf/gnueabihf and soft-float armel/gnueabi builds are provided. armhf builds for Raspbian are created using the Raspberry Pi custom cross-compilation toolchain based on gcc-arm-linux-gnueabihf and should work on ARMv6 and ARMv7 devices running Debian or Raspbian. armel builds are built using the gcc-arm-linux-gnueabi toolchain provided by Ubuntu and currently target ARMv7.

Windows 64 work is still stalling, we would welcome a volunteer to handle that.

How to use PyPy?

We suggest using PyPy from a virtualenv. Once you have a virtualenv installed, you can follow instructions from pypy documentation on how to proceed. This document also covers other installation schemes.

Cheers, the PyPy team

PyPy 2.0.2 - Fermi Panini

We're pleased to announce PyPy 2.0.2. This is a stable bugfix release over 2.0 and 2.0.1. You can download it here:

http://pypy.org/download.html

4.1. Historical release notes 255
It fixes a crash in the JIT when calling external C functions (with ctypes/cffi) in a multithreaded context.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 2.0 and cpython 2.7.3 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. Support for ARM is progressing but not bug-free yet.

Highlights

This release contains only the fix described above. A crash (or wrong results) used to occur if all these conditions were true:

- your program is multithreaded;
- it runs on a single-core machine or a heavily-loaded multi-core one;
- it uses ctypes or cffi to issue external calls to C functions.

This was fixed in the branch emit-call-x86 (see the example file bug1.py).

Cheers, arigo et. al. for the PyPy team

PyPy 2.0.1 - Bohr Smørrebrød

We’re pleased to announce PyPy 2.0.1. This is a stable bugfix release over 2.0. You can download it here:

http://pypy.org/download.html

The fixes are mainly about fatal errors or crashes in our stdlib. See below for more details.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 2.0 and cpython 2.7.3 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. Support for ARM is progressing but not bug-free yet.

Highlights

- fix an occasional crash in the JIT that ends in RPython Fatal error: NotImplementedError.
- id(x) is now always a positive number (except on int/float/long/complex). This fixes an issue in _sqlite.py (mostly for 32-bit Linux).
- fix crashes of callback-from-C-functions (with cffi) when used together with Stackless features, on asmgcc (i.e. Linux only). Now gevent should work better.
- work around an eventlet issue with socket._decref_socketios().

Cheers, arigo et. al. for the PyPy team
PyPy 2.0 - Einstein Sandwich

We’re pleased to announce PyPy 2.0. This is a stable release that brings a swath of bugfixes, small performance improvements and compatibility fixes. PyPy 2.0 is a big step for us and we hope in the future we’ll be able to provide stable releases more often.

You can download the PyPy 2.0 release here:

http://pypy.org/download.html

The two biggest changes since PyPy 1.9 are:

- stackless is now supported including greenlets, which means eventlet and gevent should work (but read below about gevent)
- PyPy now contains release 0.6 of cffi as a builtin module, which is preferred way of calling C from Python that works well on PyPy

If you’re using PyPy for anything, it would help us immensely if you fill out the following survey:  http://bit.ly/pypysurvey This is for the developers eyes and we will not make any information public without your agreement.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 2.0 and cpython 2.7.3 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. Windows 64 work is still stalling, we would welcome a volunteer to handle that. ARM support is on the way, as you can see from the recently released alpha for ARM.

Highlights

- Stackless including greenlets should work. For gevent, you need to check out pypycore and use the pypy-hacks branch of gevent.
- cffi is now a module included with PyPy. (cffi also exists for CPython; the two versions should be fully compatible.) It is the preferred way of calling C from Python that works on PyPy.
- Callbacks from C are now JITted, which means XML parsing is much faster.
- A lot of speed improvements in various language corners, most of them small, but speeding up some particular corners a lot.
- The JIT was refactored to emit machine code which manipulates a “frame” that lives on the heap rather than on the stack. This is what makes Stackless work, and it could bring another future speed-up (not done yet).
- A lot of stability issues fixed.
- Refactoring much of the numpypy array classes, which resulted in removal of lazy expression evaluation. On the other hand, we now have more complete dtype support and support more array attributes.

Cheers, fijal, arigo and the PyPy team

PyPy 2.0 beta 2

We’re pleased to announce the 2.0 beta 2 release of PyPy. This is a major release of PyPy and we’re getting very close to 2.0 final, however it includes quite a few new features that require further testing. Please test and report issues, so we can have a rock-solid 2.0 final. It also includes a performance regression of about 5% compared to 2.0 beta 1 that
we hope to fix before 2.0 final. The ARM support is not working yet and we’re working hard to make it happen before
the 2.0 final. The new major features are:

• JIT now supports stackless features, that is greenlets and stacklets. This means that JIT can now optimize the
code that switches the context. It enables running eventlet and gevent on PyPy (although gevent requires some
special support that’s not quite finished, read below).

• This is the first PyPy release that includes cffi as a core library. Version 0.6 comes included in the PyPy library.
cffi has seen a lot of adoption among library authors and we believe it’s the best way to wrap C libraries. You
can see examples of cffi usage in _curses.py and _sqlite3.py in the PyPy source code.

You can download the PyPy 2.0 beta 2 release here:

    http://pypy.org/download.html

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.3. It’s fast (pypy 2.0 beta
2 and cpython 2.7.3 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. It also supports ARM
machines running Linux, however this is disabled for the beta 2 release. Windows 64 work is still stalling, we would
welcome a volunteer to handle that.

How to use PyPy?

We suggest using PyPy from a virtualenv. Once you have a virtualenv installed, you can follow instructions from pypy
documentation on how to proceed. This document also covers other installation schemes.

Highlights

• cffi is officially supported by PyPy. It comes included in the standard library, just use import cffi

• stackless support - eventlet just works and gevent requires pypycore and pypy-hacks branch of gevent (which
mostly disables cython-based modules)

• callbacks from C are now much faster. pyexpat is about 3x faster, cffi callbacks around the same

• __length_hint__ is implemented (PEP 424)

• a lot of numpy improvements

Improvements since 1.9

• JIT hooks are now a powerful tool to introspect the JITting process that PyPy performs

• various performance improvements compared to 1.9 and 2.0 beta 1

• operations on long objects are now as fast as in CPython (from roughly 2x slower)

• we now have special strategies for dict/set/list which contain unicode strings, which means that now such
collections will be both faster and more compact.
PyPy 2.0 beta 1

We’re pleased to announce the 2.0 beta 1 release of PyPy. This release is not a typical beta, in a sense the stability is the same or better than 1.9 and can be used in production. It does however include a few performance regressions documented below that don’t allow us to label is as 2.0 final. (It also contains many performance improvements.)

The main features of this release are support for ARM processor and compatibility with CFFI. It also includes numerous improvements to the numpy in pypy effort, cpyext and performance.

You can download the PyPy 2.0 beta 1 release here:

http://pypy.org/download.html

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.3. It’s fast (pypy 2.0 beta 1 and cpython 2.7.3 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. It also supports ARM machines running Linux. Windows 64 work is still stalling, we would welcome a volunteer to handle that.

How to use PyPy?

We suggest using PyPy from a virtualenv. Once you have a virtualenv installed, you can follow instructions from pypy documentation on how to proceed. This document also covers other installation schemes.

Regressions

Reasons why this is not PyPy 2.0:

• the ctypes fast path is now slower than it used to be. In PyPy 1.9 ctypes was either incredibly faster or slower than CPython depending whether you hit the fast path or not. Right now it’s usually simply slower. We’re probably going to rewrite ctypes using cffi, which will make it universally faster.

• cffi (an alternative to interfacing with C code) is very fast, but it is missing one optimization that will make it as fast as a native call from C.

• numpypy lazy computation was disabled for the sake of simplicity. We should reenable this for the final 2.0 release.

Highlights

• cffi is officially supported by PyPy. You can install it normally by using pip install cffi once you have installed PyPy and pip. The corresponding 0.4 version of cffi has been released.

• ARM is now an officially supported processor architecture. PyPy now work on soft-float ARM/Linux builds. Currently ARM processors supporting the ARMv7 and later ISA that include a floating-point unit are supported.

• This release contains the latest Python standard library 2.7.3 and is fully compatible with Python 2.7.3.

• It does not however contain hash randomization, since the solution present in CPython is not solving the problem anyway. The reason can be found on the CPython issue tracker.

• gc.get_referrers() is now faster.

• Various numpy improvements. The list includes:
PyPy, Release 7.2.0

- axis argument support in many places
- full support for fancy indexing
- complex128 and complex64 dtypes

**JIT hooks** are now a powerful tool to introspect the JITting process that PyPy performs.

**kwds** usage is much faster in the typical scenario

operations on long objects are now as fast as in CPython (from roughly 2x slower)

We now have special strategies for dict/set/list which contain unicode strings, which means that now such collections will be both faster and more compact.

**Things we’re working on**

There are a few things that did not make it to the 2.0 beta 1, which are being actively worked on. Greenlets support in the JIT is one that we would like to have before 2.0 final. Two important items that will not make it to 2.0, but are being actively worked on, are:

- Faster JIT warmup time.
- Software Transactional Memory.

Cheers, Maciej Fijalkowski, Armin Rigo and the PyPy team

**PyPy 1.9 - Yard Wolf**

We’re pleased to announce the 1.9 release of PyPy. This release brings mostly bugfixes, performance improvements, other small improvements and overall progress on the numpypy effort. It also brings an improved situation on Windows and OS X.

You can download the PyPy 1.9 release here:

http://pypy.org/download.html

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 1.9 and cpython 2.7.2 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. Windows 64 work is still stalling, we would welcome a volunteer to handle that.

**Thanks to our donors**

But first of all, we would like to say thank you to all people who donated some money to one of our four calls:

- NumPy in PyPy (got so far $44502 out of $60000, 74%)
- Py3k (Python 3) (got so far $43563 out of $105000, 41%)
- Software Transactional Memory (got so far $21791 of $50400, 43%)
- as well as our general PyPy pot.
Thank you all for proving that it is indeed possible for a small team of programmers to get funded like that, at least for some time. We want to include this thank you in the present release announcement even though most of the work is not finished yet. More precisely, neither Py3k nor STM are ready to make it in an official release yet: people interested in them need to grab and (attempt to) translate PyPy from the corresponding branches (respectively py3k and stm-thread).

**Highlights**

- This release still implements Python 2.7.2.
- Many bugs were corrected for Windows 32 bit. This includes new functionality to test the validity of file descriptors; and correct handling of the calling conventions for ctypes. (Still not much progress on Win64.) A lot of work on this has been done by Matti Picus and Amaury Forgeot d’Arc.
- Improvements in cpyext, our emulator for CPython C extension modules. For example PyOpenSSL should now work. We thank various people for help.
- Sets now have strategies just like dictionaries. This means for example that a set containing only ints will be more compact (and faster).
- A lot of progress on various aspects of numpypy. See the numpy-status page for the automatic report.
- It is now possible to create and manipulate C-like structures using the PyPy-only _ffi module. The advantage over using e.g. ctypes is that _ffi is very JIT-friendly, and getting/setting of fields is translated to few assembler instructions by the JIT. However, this is mostly intended as a low-level backend to be used by more user-friendly FFI packages, and the API might change in the future. Use it at your own risk.
- The non-x86 backends for the JIT are progressing but are still not merged (ARMv7 and PPC64).
- JIT hooks for inspecting the created assembler code have been improved. See JIT hooks documentation for details.
- select.kqueue has been added (BSD).
- Handling of keyword arguments has been drastically improved in the best-case scenario: proxy functions which simply forwards *args and **kwargs to another function now performs much better with the JIT.
- List comprehension has been improved.

**JitViewer**

There will be a corresponding 1.9 release of JitViewer which is guaranteed to work with PyPy 1.9. See the JitViewer docs for details.

Cheers, The PyPy Team

**PyPy 1.8 - business as usual**

We’re pleased to announce the 1.8 release of PyPy. As habitual this release brings a lot of bugfixes, together with performance and memory improvements over the 1.7 release. The main highlight of the release is the introduction of list strategies which makes homogenous lists more efficient both in terms of performance and memory. This release also upgrades us from Python 2.7.1 compatibility to 2.7.2. Otherwise it’s “business as usual” in the sense that performance improved roughly 10% on average since the previous release.

you can download the PyPy 1.8 release here:

http://pypy.org/download.html
What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 1.8 and cpython 2.7.1 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 32/64 or Windows 32. Windows 64 work has been stalled, we would welcome a volunteer to handle that.

Highlights

• List strategies. Now lists that contain only ints or only floats should be as efficient as storing them in a binary-packed array. It also improves the JIT performance in places that use such lists. There are also special strategies for unicode and string lists.
• As usual, numerous performance improvements. There are many examples of python constructs that now should be faster; too many to list them.
• Bugfixes and compatibility fixes with CPython.
• Windows fixes.
• NumPy effort progress; for the exact list of things that have been done, consult the numpy status page. A tentative list of things that has been done:
  – multi dimensional arrays
  – various sizes of dtypes
  – a lot of ufuncs
  – a lot of other minor changes

Right now the numpy module is available under both numpy and numpypy names. However, because it’s incomplete, you have to import numpypy first before doing any imports from numpy.
• New JIT hooks that allow you to hook into the JIT process from your python program. There is a brief overview of what they offer.
• Standard library upgrade from 2.7.1 to 2.7.2.

Ongoing work

As usual, there is quite a bit of ongoing work that either didn’t make it to the release or is not ready yet. Highlights include:

• Non-x86 backends for the JIT: ARMv7 (almost ready) and PPC64 (in progress)
• Specialized type instances - allocate instances as efficient as C structs, including type specialization
• More numpy work
• Since the last release there was a significant breakthrough in PyPy’s fundraising. We now have enough funds to work on first stages of numpypy and py3k. We would like to thank again to everyone who donated.
• It’s also probably worth noting, we’re considering donations for the Software Transactional Memory project. You can read more about our plans

Cheers, The PyPy Team
PyPy 1.7 - widening the sweet spot

We’re pleased to announce the 1.7 release of PyPy. As became a habit, this release brings a lot of bugfixes and performance improvements over the 1.6 release. However, unlike the previous releases, the focus has been on widening the “sweet spot” of PyPy. That is, classes of Python code that PyPy can greatly speed up should be vastly improved with this release. You can download the 1.7 release here:

http://pypy.org/download.html

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7. It’s fast (pypy 1.7 and cpython 2.7.1 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 32/64 or Windows 32. Windows 64 work is ongoing, but not yet natively supported.

The main topic of this release is widening the range of code which PyPy can greatly speed up. On average on our benchmark suite, PyPy 1.7 is around 30% faster than PyPy 1.6 and up to 20 times faster on some benchmarks.

Highlights

- Numerous performance improvements. There are too many examples which python constructs now should behave faster to list them.
- Bugfixes and compatibility fixes with CPython.
- Windows fixes.
- PyPy now comes with stackless features enabled by default. However, any loop using stackless features will interrupt the JIT for now, so no real performance improvement for stackless-based programs. Contact pypy-dev for info how to help on removing this restriction.
- NumPy effort in PyPy was renamed numpypy. In order to try using it, simply write:

```python
import numpypy as numpy
```

at the beginning of your program. There is a huge progress on numpy in PyPy since 1.6, the main feature being implementation of dtypes.
- JSON encoder (but not decoder) has been replaced with a new one. This one is written in pure Python, but is known to outperform CPython’s C extension up to 2 times in some cases. It’s about 20 times faster than the one that we had in 1.6.
- The memory footprint of some of our RPython modules has been drastically improved. This should impact any applications using for example cryptography, like tornado.
- There was some progress in exposing even more CPython C API via cpyext.

Things that didn’t make it, expect in 1.8 soon

There is an ongoing work, which while didn’t make it to the release, is probably worth mentioning here. This is what you should probably expect in 1.8 some time soon:

- Specialized list implementation. There is a branch that implements lists of integers/floats/strings as compactly as array.array. This should drastically improve performance/memory impact of some applications
PyPy, Release 7.2.0

- NumPy effort is progressing forward, with multi-dimensional arrays coming soon.
- There are two brand new JIT assembler backends, notably for the PowerPC and ARM processors.

Fundraising

It’s maybe worth mentioning that we’re running fundraising campaigns for NumPy effort in PyPy and for Python 3 in PyPy. In case you want to see any of those happen faster, we urge you to donate to numpy proposal or py3k proposal. In case you want PyPy to progress, but you trust us with the general direction, you can always donate to the general pot.

PyPy 1.6 - kickass panda

We’re pleased to announce the 1.6 release of PyPy. This release brings a lot of bugfixes and performance improvements over 1.5, and improves support for Windows 32bit and OS X 64bit. This version fully implements Python 2.7.1 and has beta level support for loading CPython C extensions. You can download it here:

http://pypy.org/download.html

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.1. It’s fast (pypy 1.5 and cpython 2.6.2 performance comparison) due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64 or Mac OS X. Windows 32 is beta (it roughly works but a lot of small issues have not been fixed so far). Windows 64 is not yet supported.

The main topics of this release are speed and stability: on average on our benchmark suite, PyPy 1.6 is between 20% and 30% faster than PyPy 1.5, which was already much faster than CPython on our set of benchmarks.

The speed improvements have been made possible by optimizing many of the layers which compose PyPy. In particular, we improved: the Garbage Collector, the JIT warmup time, the optimizations performed by the JIT, the quality of the generated machine code and the implementation of our Python interpreter.

Highlights

- Numerous performance improvements, overall giving considerable speedups:
  - better GC behavior when dealing with very large objects and arrays
  - fast ctypes: now calls to ctypes functions are seen and optimized by the JIT, and they are up to 60 times faster than PyPy 1.5 and 10 times faster than CPython
  - improved generators(1): simple generators now are inlined into the caller loop, making performance up to 3.5 times faster than PyPy 1.5.
  - improved generators(2): thanks to other optimizations, even generators that are not inlined are between 10% and 20% faster than PyPy 1.5.
  - faster warmup time for the JIT
  - JIT support for single floats (e.g., for array('f'))
  - optimized dictionaries: the internal representation of dictionaries is now dynamically selected depending on the type of stored objects, resulting in faster code and smaller memory footprint. For example, dictionaries whose keys are all strings, or all integers. Other dictionaries are also smaller due to bugfixes.
PyPy, Release 7.2.0

- JitViewer: this is the first official release which includes the JitViewer, a web-based tool which helps you to see which parts of your Python code have been compiled by the JIT, down until the assembler. The jitviewer 0.1 has already been release and works well with PyPy 1.6.

- The CPython extension module API has been improved and now supports many more extensions. For information on which one are supported, please refer to our compatibility wiki.

- Multibyte encoding support: this was of of the last areas in which we were still behind CPython, but now we fully support them.

- Preliminary support for NumPy: this release includes a preview of a very fast NumPy module integrated with the PyPy JIT. Unfortunately, this does not mean that you can expect to take an existing NumPy program and run it on PyPy, because the module is still unfinished and supports only some of the numpy API. However, barring some details, what works should be blazingly fast :-)

- Bugfixes: since the 1.5 release we fixed 53 bugs in our bug tracker, not counting the numerous bugs that were found and reported through other channels than the bug tracker.

Cheers,
Hakan Ardo, Carl Friedrich Bolz, Laura Creighton, Antonio Cuni, Maciej Fijalkowski, Amaury Forgeot d’Arc, Alex Gaynor, Armin Rigo and the PyPy team

PyPy 1.5: Catching Up

We’re pleased to announce the 1.5 release of PyPy. This release updates PyPy with the features of CPython 2.7.1, including the standard library. Thus all the features of CPython 2.6 and CPython 2.7 are now supported. It also contains additional performance improvements. You can download it here:

http://pypy.org/download.html

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.1. It’s fast (pypy 1.5 and cpython 2.6.2 performance comparison) due to its integrated tracing JIT compiler.

This release includes the features of CPython 2.6 and 2.7. It also includes a large number of small improvements to the tracing JIT compiler. It supports Intel machines running Linux 32/64 or Mac OS X. Windows is beta (it roughly works but a lot of small issues have not been fixed so far). Windows 64 is not yet supported.

Numerous speed achievements are described on our blog. Normalized speed charts comparing pypy 1.5 and pypy 1.4 as well as pypy 1.5 and cpython 2.6.2 are available on our benchmark website. The speed improvement over 1.4 seems to be around 25% on average.

More highlights

- The largest change in PyPy’s tracing JIT is adding support for loop invariant code motion, which was mostly done by Håkan Ardö. This feature improves the performance of tight loops doing numerical calculations.

- The CPython extension module API has been improved and now supports many more extensions. For information on which one are supported, please refer to our compatibility wiki.

- These changes make it possible to support Tkinter and IDLE.

- The cProfile profiler is now working with the JIT. However, it skews the performance in unstudied ways. Therefore it is not yet usable to analyze subtle performance problems (the same is true for CPython of course).

4.1. Historical release notes

265
• There is an external fork which includes an RPython version of the postgresql. However, there are no prebuilt binaries for this.

• Our developer documentation was moved to Sphinx and cleaned up. (click ‘Dev Site’ on http://pypy.org/.)

• and many small things :-)

Cheers,

Carl Friedrich Bolz, Laura Creighton, Antonio Cuni, Maciej Fijalkowski, Amaury Forgeot d’Arc, Alex Gaynor, Armin Rigo and the PyPy team

PyPy 1.4.1

We’re pleased to announce the 1.4.1 release of PyPy. This release consolidates all the bug fixes that occurred since the previous release. To everyone that took the trouble to report them, we want to say thank you.

http://pypy.org/download.html

What is PyPy

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython. Note that it still only emulates Python 2.5 by default; the fast-forward branch with Python 2.7 support is slowly getting ready but will only be integrated in the next release.

In two words, the advantage of trying out PyPy instead of CPython (the default implementation of Python) is, for now, the performance. Not all programs are faster in PyPy, but we are confident that any CPU-intensive task will be much faster, at least if it runs for long enough (the JIT has a slow warm-up phase, which can take several seconds or even one minute on the largest programs).

Note again that we do support compiling and using C extension modules from CPython (pypy_setup.py install). However, this is still an alpha feature, and the most complex modules typically fail for various reasons; others work (e.g. PIL) but take a serious performance hit. Also, for Mac OS X see below.

Please note also that PyPy’s performance was optimized almost exclusively on Linux. It seems from some reports that on Windows as well as Mac OS X (probably for different reasons) the performance might be lower. We did not investigate much so far.

More highlights

• We migrated to Mercurial (thanks to Ronny Pfannschmidt and Antonio Cuni) for the effort) and moved to bitbucket. The new command to check out a copy of PyPy is:

hg clone http://bitbucket.org/pypy/pypy

• In long-running processes, the assembler generated by old JIT-compilations is now freed. There should be no more leak, however long the process runs.

• Improve a lot the performance of the binascii module, and of hashlib.md5 and hashlib.sha.

• Made sys.setrecursionlimit() a no-op. Instead, we rely purely on the built-in stack overflow detection mechanism, which also gives you a RuntimeError — just not at some exact recursion level.

• Fix argument processing (now e.g. pypy -OScpass works like it does on CPython — if you have a clue what it does there : - ) )

• cpyext on Mac OS X: it still does not seem to work. I get systematically a segfault in dlopen(). Contributions welcome.
• Fix two corner cases in the GC (one in minimark, one in asmgcc+JIT). This notably prevented “pypy translate.py -Ojit” from working on Windows, leading to crashes.
• Fixed a corner case in the JIT’s optimizer, leading to “Fatal RPython error: AssertionError”.
• Added some missing built-in functions into the ‘os’ module.
• Fix ctypes (it was not propagating keepalive information from c_void_p).

Cheers,

Armin Rigo, for the rest of the team

PyPy 1.4beta - towards 1.4.0 release

Hello.

As we head towards 1.4 release, which should be considered the very first PyPy release ready to substitute CPython for at least some of us, we’re pleased to announce 1.4 beta release.

This release contains all major features from upcoming 1.4, the only thing missing being improved memory footprint. However, this is a beta release and might still contain some issues. One of those issues is that, like on nightly builds, pypy might print some debugging output at the end of your program run.

Highlights:
• x86_64 JIT backend
• since PyPy 1.3 we have an experimental support for CPython C extensions. Those have to be recompiled using pypy setup.py build. Extensions usually have to be tweaked for e.g. refcounting bugs that don’t manifest on CPython. There is a list of patches available for some extensions.
• rewritten fast and jitted regular expressions
• improvements all across the board (for example faster map calls)
• virtualenv support (virtualenv 1.5 or later)

Cheers, The PyPy team

PyPy 1.4: Ouroboros in practice

We’re pleased to announce the 1.4 release of PyPy. This is a major breakthrough in our long journey, as PyPy 1.4 is the first PyPy release that can translate itself faster than CPython. Starting today, we are using PyPy more for our every-day development. So may you :) You can download it here:

http://pypy.org/download.html

What is PyPy

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython. It’s fast (pypy 1.4 and cpython 2.6 comparison)

Among its new features, this release includes numerous performance improvements (which made fast self-hosting possible), a 64-bit JIT backend, as well as serious stabilization. As of now, we can consider the 32-bit and 64-bit linux versions of PyPy stable enough to run in production.

Numerous speed achievements are described on our blog. Normalized speed charts comparing pypy 1.4 and pypy 1.3 as well as pypy 1.4 and cpython 2.6 are available on benchmark website. For the impatient: yes, we got a lot faster!
More highlights

- PyPy’s built-in Just-in-Time compiler is fully transparent and automatically generated; it now also has very reasonable memory requirements. The total memory used by a very complex and long-running process (translating PyPy itself) is within 1.5x to at most 2x the memory needed by CPython, for a speed-up of 2x.

- More compact instances. All instances are as compact as if they had `__slots__`. This can give programs a big gain in memory. (In the example of translation above, we already have carefully placed `__slots__`, so there is no extra win.)

- Virtualenv support: now PyPy is fully compatible with virtualenv: note that to use it, you need a recent version of virtualenv (>= 1.5).

- Faster (and JITted) regular expressions - huge boost in speeding up the `re` module.

- Other speed improvements, like JITted calls to functions like `map()`.

Cheers,
Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Amaury Forgeot d’Arc, Armin Rigo and the PyPy team

PyPy 1.3: Stabilization

Hello.

We’re please to announce release of PyPy 1.3. This release has two major improvements. First of all, we stabilized the JIT compiler since 1.2 release, answered user issues, fixed bugs, and generally improved speed.

We’re also pleased to announce alpha support for loading CPython extension modules written in C. While the main purpose of this release is increased stability, this feature is in alpha stage and it is not yet suited for production environments.

Highlights of this release

- We introduced support for CPython extension modules written in C. As of now, this support is in alpha, and it’s very unlikely unaltered C extensions will work out of the box, due to missing functions or refcounting details. The support is disable by default, so you have to do:

  ```python
  import cpyext
  ```

  before trying to import any .so file. Also, libraries are source-compatible and not binary-compatible. That means you need to recompile binaries, using for example:

  ```bash
  python setup.py build
  ```

  Details may vary, depending on your build system. Make sure you include the above line at the beginning of setup.py or put it in your PYTHONSTARTUP.

  This is alpha feature. It’ll likely segfault. You have been warned!

- JIT bugfixes. A lot of bugs reported for the JIT have been fixed, and its stability greatly improved since 1.2 release.

- Various small improvements have been added to the JIT code, as well as a great speedup of compiling time.

Cheers, Maciej Fijalkowski, Armin Rigo, Alex Gaynor, Amaury Forgeot d’Arc and the PyPy team
PyPy 1.2: Just-in-Time Compilation

Welcome to the PyPy 1.2 release. The highlight of this release is to be the first that ships with a Just-in-Time compiler that is known to be faster than CPython (and unladen swallow) on some real-world applications (or the best benchmarks we could get for them). The main theme for the 1.2 release is speed.

Main site:

http://pypy.org/

The JIT is stable and we don’t observe crashes. Nevertheless we would recommend you to treat it as beta software and as a way to try out the JIT to see how it works for you.

Highlights of This Release

• The JIT compiler.
• Various interpreter optimizations that improve performance as well as help save memory.
• Introducing a new PyPy website at http://pypy.org/, made by tav and improved by the PyPy team.
• Introducing http://speed.pypy.org/, a new service that monitors our performance nightly, made by Miquel Torres.
• There will be ubuntu packages on “PyPy’s PPA” made by Bartosz Skowron; however various troubles prevented us from having them as of now.

Known JIT problems (or why you should consider this beta software):

• The only supported platform is 32bit x86 for now, we’re looking for help with other platforms.
• It is still memory-hungry. There is no limit on the amount of RAM that the assembler can consume; it is thus possible (although unlikely) that the assembler ends up using unreasonable amounts of memory.

If you want to try PyPy, go to the “download page” on our excellent new site at http://pypy.org/download.html and find the binary for your platform. If the binary does not work (e.g. on Linux, because of different versions of external .so dependencies), or if your platform is not supported, you can try building from the source.

What is PyPy?

Technically, PyPy is both a Python interpreter implementation and an advanced compiler, or more precisely a framework for implementing dynamic languages and generating virtual machines for them. The focus of this release is the introduction of a new transformation, the JIT Compiler Generator, and its application to the Python interpreter.

Socially, PyPy is a collaborative effort of many individuals working together in a distributed and sprint-driven way since 2003. PyPy would not have gotten as far as it has without the coding, feedback and general support from numerous people.

The PyPy release team, Armin Rigo, Maciej Fijalkowski and Amaury Forgeot d’Arc

Together with Antonio Cuni, Carl Friedrich Bolz, Holger Krekel and Samuele Pedroni

and many others: http://codespeak.net/pypy/dist/pypy/doc/contributor.html

PyPy 1.1: Compatibility & Consolidation

Welcome to the PyPy 1.1 release - the first release after the end of EU funding. This release focuses on making PyPy’s Python interpreter more compatible with CPython (currently CPython 2.5) and on making the interpreter more stable and bug-free.
Download page:

http://codespeak.net/pypy/dist/pypy/doc/download.html

PyPy’s Getting Started lives at:

http://codespeak.net/pypy/dist/pypy/doc/getting-started.html

**Highlights of This Release**

- More of CPython’s standard library extension modules are supported, among them ctypes, sqlite3, csv, and many more. Most of these extension modules are fully supported under Windows as well.


- Through a large number of tweaks, performance has been improved by 10%-50% since the 1.0 release. The Python interpreter is now between 0.8-2x (and in some corner case 3-4x) slower than CPython. A large part of these speed-ups come from our new generational garbage collectors.

  http://codespeak.net/pypy/dist/pypy/doc/garbage_collection.html

- Our Python interpreter now supports distutils as well as easy_install for pure-Python modules.

- We have tested PyPy with a number of third-party libraries. PyPy can run now: Django, Pylons, BitTorrent, Twisted, SymPy, Pyglet, Nevow, Pinax:


- A buildbot was set up to run the various tests that PyPy is using nightly on Windows and Linux machines:

  http://codespeak.net:8099/

- Sandboxing support: It is possible to translate the Python interpreter in a special way so that the result is fully sandboxed.


**Other Changes**

- The clr module was greatly improved. This module is used to interface with .NET libraries when translating the Python interpreter to the CLI.


- Stackless improvements: PyPy’s stackless module is now more complete. We added channel preferences which change details of the scheduling semantics. In addition, the pickling of tasklets has been improved to work in more cases.

- Classic classes are enabled by default now. In addition, they have been greatly optimized and debugged:


- PyPy’s Python interpreter can be translated to Java bytecode now to produce a pypy-jvm. At the moment there is no integration with Java libraries yet, so this is not really useful.
We added cross-compilation machinery to our translation toolchain to make it possible to cross-compile our Python interpreter to Nokia’s Maemo platform:

http://codespeak.net/pypy/dist/pypy/doc/maemo.html

Some effort was spent to make the Python interpreter more memory-efficient. This includes the implementation of a mark-compact GC which uses less memory than other GCs during collection. Additionally there were various optimizations that make Python objects smaller, e.g. class instances are often only 50% of the size of CPython.


The support for the trace hook in the Python interpreter was improved to be able to trace the execution of builtin functions and methods. With this, we implemented the _lsprof module, which is the core of the cProfile module.

A number of rarely used features of PyPy were removed since the previous release because they were unmaintained and/or buggy. Those are: The LLVM and the JS backends, the aspect-oriented programming features, the logic object space, the extension compiler and the first incarnation of the JIT generator. The new JIT generator is in active development, but not included in the release.


What is PyPy?

Technically, PyPy is both a Python interpreter implementation and an advanced compiler, or more precisely a framework for implementing dynamic languages and generating virtual machines for them.

The framework allows for alternative frontends and for alternative backends, currently C, Java and .NET. For our main target “C”, we can “mix in” different garbage collectors and threading models, including micro-threads aka “Stackless”. The inherent complexity that arises from this ambitious approach is mostly kept away from the Python interpreter implementation, our main frontend.

Socially, PyPy is a collaborative effort of many individuals working together in a distributed and sprint-driven way since 2003. PyPy would not have gotten as far as it has without the coding, feedback and general support from numerous people.

Have fun,

the PyPy release team, [in alphabetical order]

Amaury Forgeot d’Arc, Anders Hammerquist, Antonio Cuni, Armin Rigo, Carl Friedrich Bolz, Christian Tismer, Holger Krekel, Maciek Fijalkowski, Samuele Pedroni

and many others: http://codespeak.net/pypy/dist/pypy/doc/contributor.html

PyPy 1.0: JIT compilers for free and more

Welcome to the PyPy 1.0 release - a milestone integrating the results of four years of research, engineering, management and sprinting efforts, concluding the 28 months phase of EU co-funding!

Although still not mature enough for general use, PyPy 1.0 materializes for the first time the full extent of our original vision:

- A flexible Python interpreter, written in “RPython”:
  - Mostly unaware of threading, memory and lower-level target platform aspects.
  - Showcasing advanced interpreter features and prototypes.
– Passing core CPython regression tests, translatable to C, LLVM and .NET.

• An advanced framework to translate such interpreters and programs:
  – That performs whole type-inference on RPython programs.
  – Can weave in threading, memory and target platform aspects.
  – Has low level (C, LLVM) and high level (CLI, Java, JavaScript) backends.

• A Just-In-Time Compiler generator able to automatically enhance the low level versions of our Python interpreter, leading to run-time machine code that runs algorithmic examples at speeds typical of JITs!

Previous releases, particularly the 0.99.0 release from February, already highlighted features of our Python implementation and the abilities of our translation approach but the new JIT generator clearly marks a major research result and gives weight to our vision that one can generate efficient interpreter implementations, starting from a description in a high level language.

We have prepared several entry points to help you get started:

• The main entry point for JIT documentation and status:
  http://codespeak.net/pypy/dist/pypy/doc/jit.html

• The main documentation and getting-started PyPy entry point:
  http://codespeak.net/pypy/dist/pypy/doc/index.html

• Our online “play1” demos showcasing various Python interpreters, features (and a new way to program AJAX applications):
  http://play1.codespeak.net/

• Our detailed and in-depth Reports about various aspects of the project:
  http://codespeak.net/pypy/dist/pypy/doc/index-report.html

In the next few months we are going to discuss the goals and form of the next stage of development - now more than ever depending on your feedback and contributions - and we hope you appreciate PyPy 1.0 as an interesting basis for greater things to come, as much as we do ourselves!

have fun,

the PyPy release team, Samuele Pedroni, Armin Rigo, Holger Krekel, Michael Hudson, Carl Friedrich Bolz, Antonio Cuni, Anders Chrigstroem, Guido Wesdorp Maciej Fijalkowski, Alexandre Fayolle

and many others: http://codespeak.net/pypy/dist/pypy/doc/contributor.html

What is PyPy?

Technically, PyPy is both a Python interpreter implementation and an advanced compiler, or more precisely a framework for implementing dynamic languages and generating virtual machines for them.

The framework allows for alternative frontends and for alternative backends, currently C, LLVM and .NET. For our main target ‘C’, we can can “mix in” different garbage collectors and threading models, including micro-threads aka “Stackless”. The inherent complexity that arises from this ambitious approach is mostly kept away from the Python interpreter implementation, our main frontend.

PyPy is now also a Just-In-Time compiler generator. The translation framework contains the now-integrated JIT generation technology. This depends only on a few hints added to the interpreter source and should be able to cope with the changes to the interpreter and be generally applicable to other interpreters written using the framework.
Socially, PyPy is a collaborative effort of many individuals working together in a distributed and sprint-driven way since 2003. PyPy would not have gotten as far as it has without the coding, feedback and general support from numerous people.

Formally, many of the current developers were involved in executing an EU contract with the goal of exploring and researching new approaches to language and compiler development and software engineering. This contract’s duration is about to end this month (March 2007) and we are working and preparing the according final review which is scheduled for May 2007.

For the future, we are in the process of setting up structures to help maintain conceptual integrity of the project and to discuss and deal with funding opportunities related to further PyPy sprinting and developments. See here for results of the discussion so far:

http://codespeak.net/pipermail/pypy-dev/2007q1/003577.html

1.0.0 Feature highlights

Here is a summary list of key features included in PyPy 1.0:

- The Just-In-Time compiler generator, now capable of generating the first JIT compiler versions of our Python interpreter:
  http://codespeak.net/pypy/dist/pypy/doc/jit.html
- More Python interpreter optimizations (a CALL_METHOD bytecode, a method cache, rope-based strings), now running benchmarks at around half of CPython’s speed (without the JIT):
  http://codespeak.net/pypy/dist/pypy/doc/interpreter-optimizations.html
- The Python interpreter can be translated to .NET and enables interactions with the CLR libraries:
- Aspect Oriented Programming facilities (based on mutating the Abstract Syntax Tree):
- The JavaScript backend has evolved to a point where it can be used to write AJAX web applications with it. This is still an experimental technique, though. For demo applications which also showcase various generated Python and PROLOG interpreters, see:
  http://play1.codespeak.net/
- Proxying object spaces and features of our Python interpreter:
  - Tainting: a 270-line proxy object space tracking and boxing sensitive information within an application.
  - Transparent proxies: allow the customization of both application and builtin objects from application level code. Now featuring an initial support module (tputil.py) for working with transparent proxies.

For a detailed description and discussion of high level backends and Python interpreter features, please see our extensive “D12” report:


Funding partners and organizations

PyPy development and activities happen as an open source project and with the support of a consortium partially funded by a 28 month European Union IST research grant for the period from December 2004 to March 2007. The
Welcome to the PyPy 0.99.0 release - a major snapshot and milestone of the last 8 months of work and contributions since PyPy-0.9.0 came out in June 2006!

Main entry point for getting-started/download and documentation:

   http://codespeak.net/pypy/dist/pypy/doc/index.html

Further below you’ll find some notes about PyPy, the 0.99.0 highlights and our aims for PyPy 1.0.

have fun,

the PyPy team, Samuele Pedroni, Carl Friedrich Bolz, Armin Rigo, Michael Hudson, Maciej Fijalkowski, Anders Chrigstroem, Holger Krekel, Guido Wesdorp

and many others: http://codespeak.net/pypy/dist/pypy/doc/contributor.html

What is PyPy?

Technically, PyPy is both a Python Interpreter implementation and an advanced Compiler, actually a framework for implementing dynamic languages and generating virtual machines for them. The Framework allows for alternative frontends and for alternative backends, currently C, LLVM and .NET. For our main target “C”, we can can “mix in” different Garbage Collectors and threading models, including micro-threads aka “Stackless”. The inherent complexity that arises from this ambitious approach is mostly kept away from the Python interpreter implementation, our main frontend.

Socially, PyPy is a collaborative effort of many individuals working together in a distributed and sprint-driven way since 2003. PyPy would not have gotten as far without the coding, feedback and general support from numerous people.

Formally, many of the current developers are involved in executing an EU contract with the goal of exploring and researching new approaches to Language/Compiler development and software engineering. This contract’s duration is about to end March 2007 and we are working and preparing the according final review which is scheduled for May 2007.

Key 0.99.0 Features

- new object spaces:
  - Tainting: a 270-line proxy object space tracking and boxing sensitive information within an application. A tainted object is completely barred from crossing an I/O barrier, such as writing to files, databases or sockets. This allows to significantly reduce the effort of e.g. security reviews to the few places where objects are “declassified” in order to send information across I/O barriers.
  - Transparent proxies: allow to customize both application and builtin objects from application level code. Works as an addition to the Standard Object Space (and is translatable). For details see http://codespeak.net/pypy/dist/pypy/doc/proxy.html

- optimizations:
  - Experimental new optimized implementations for various built in Python types (strings, dicts, lists)
– Optimized builtin lookups to not require any dictionary lookups if the builtin is not shadowed by a name in the global dictionary.
– Improved inlining (now also working for higher level backends) and malloc removal.
– twice the speed of the 0.9 release, overall 2-3 slower than CPython

• High level backends:
  – It is now possible to translate the PyPy interpreter to run on the .NET platform, which gives a very compliant (but somewhat slow) Python interpreter.
  – the JavaScript backend has evolved to a point where it can be used to write AJAX web applications with it. This is still an experimental technique, though. For demo applications see: http://play1.codespeak.net/

• new configuration system: There is a new comprehensive configuration system that allows fine-grained configuration of the PyPy standard interpreter and the translation process.

• new and improved modules: Since the last release, the signal, mmap, bz2 and fcntl standard library modules have been implemented for PyPy. The socket, _sre and os modules have been greatly improved. In addition we added a the pypymagic module that contains PyPy-specific functionality.

• improved file implementation: Our file implementation was ported to RPython and is therefore faster (and not based on libc).

• The stability of stackless features was greatly improved. For more details see: http://codespeak.net/pypy/dist/pypy/doc/stackless.html

• RPython library: The release contains our emerging RPython library that tries to make programming in RPython more pleasant. It contains an experimental parser generator framework. For more details see: http://codespeak.net/pypy/dist/pypy/doc/rlib.html

• improved documentation:
  – extended documentation about stackless features: http://codespeak.net/pypy/dist/pypy/doc/stackless.html
  – PyPy video documentation: eight hours of talks, interviews and features: http://codespeak.net/pypy/dist/pypy/doc/video-index.html
  – technical reports about various aspects of PyPy: http://codespeak.net/pypy/dist/pypy/doc/index-report.html

The entry point to all our documentation is: http://codespeak.net/pypy/dist/pypy/doc/index.html

What about 1.0?

In the last week leading up to the release, we decided to go for tagging the release as 0.99.0, mainly because we have some efforts pending to integrate and complete research and coding work:

• the JIT Compiler Generator is ready, but not fully integrated with the PyPy interpreter. As a result, the JIT does not give actual speed improvements yet, so we chose to leave it out of the 0.99 release: the result doesn’t meet yet the speed expectations that we set for ourselves - and which some blogs and people have chosen as the main criterium for looking at PyPy.

• the extension enabling runtime changes of the Python grammar is not yet integrated. This will be used to provide Aspect-Oriented Programming extensions and Design by Contract facilities in PyPy.

• the Logic object space, which provides Logic Variables in PyPy, needs to undergo a bit more testing. A constraint problem solver extension module is ready, and needs to be integrated with the codebase.

PyPy 0.99 is the start for getting to 1.0 end of March 2007, which we intend to become a base for a longer (and more relaxed :) time to come.
Funding partners and organizations

PyPy development and activities happen as an open source project and with the support of a consortium partially funded by a 28 months European Union IST research grant. The full partners of that consortium are:

- Heinrich-Heine University (Germany)
- Open End (Sweden)
- merlinux GmbH (Germany)
- tismerysoft GmbH (Germany)
- Logilab Paris (France)
- DFKI GmbH (Germany)
- ChangeMaker (Sweden)
- Impara (Germany)

pypy-0.9.0: stackless, new extension compiler

The PyPy development team has been busy working and we’ve now packaged our latest improvements, completed work and new experiments as version 0.9.0, our fourth public release.

The highlights of this fourth release of PyPy are:

- **implementation of “stackless” features** We now support the larger part of the interface of the original Stackless Python – see [http://www.stackless.com](http://www.stackless.com) for more. A significant part of this is the pickling and unpickling of a running tasklet.

  These features, especially the pickling, can be considered to be a “technology preview” – they work, but for example the error handling is a little patchy in places.

- **ext-compiler** The “extension compiler” is a new way of writing a C extension for CPython and PyPy at the same time. For more information, see its documentation: [http://codespeak.net/pypy/dist/pypy/doc/extcompiler.html](http://codespeak.net/pypy/dist/pypy/doc/extcompiler.html)

- **rctypes** Most useful in combination with the ext-compiler is the fact that our translation framework can translate code that uses the standard-in-Python-2.5 ctypes module. See its documentation for more: [http://codespeak.net/pypy/dist/pypy/doc/rctypes.html](http://codespeak.net/pypy/dist/pypy/doc/rctypes.html)

- **framework GCs** PyPy’s interpreter can now be compiled to use a garbage collector written in RPython. This added control over PyPy’s execution makes the implementation of new and interesting features possible, apart from being a significant achievement in its own right.

- **__del__/weakref/__subclasses__** The PyPy interpreter’s compatibility with CPython continues improves: now we support __del__ methods, the __subclasses__ method on types and weak references. We now pass around 95% of CPython’s core tests.

- **logic space preview** This release contains the first version of the logic object space, which will add logical variables to Python. See its docs for more: [http://codespeak.net/pypy/dist/pypy/doc/howto-logicobjspace-0.9.html](http://codespeak.net/pypy/dist/pypy/doc/howto-logicobjspace-0.9.html)

- **high level backends preview** This release contains the first versions of new backends targeting high level languages such as Squeak and .NET/CLI and updated versions of the JavaScript and Common Lisp backends. They can’t compile the PyPy interpreter yet, but they’re getting there...

- **bugfixes, better performance** As you would expect, performance continues to improve and bugs continue to be fixed. The performance of the translated PyPy interpreter is 2.5-3x times faster than 0.8 (on richards and pystone), and is now stable enough to be able to run CPython’s test suite to the end.

- **testing refinements** py.test, our testing tool, now has preliminary support for doctests. We now run all our tests every night, and you can see the summary at: [http://buildbot.pypy.org/summary](http://buildbot.pypy.org/summary)

What is PyPy (about)?

PyPy is a MIT-licensed research-oriented reimplementation of Python written in Python itself, flexible and easy to experiment with. It translates itself to lower level languages. Our goals are to target a large variety of platforms, small and large, by providing a compilation toolsuite that can produce custom Python versions. Platform, memory and threading models are to become aspects of the translation process - as opposed to encoding low level details into
the language implementation itself. Eventually, dynamic optimization techniques - implemented as another translation aspect - should become robust against language changes.

Note that PyPy is mainly a research and development project and does not by itself focus on getting a production-ready Python implementation although we do hope and expect it to become a viable contender in that area sometime next year.

PyPy is partially funded as a research project under the European Union’s IST programme.

Where to start?

Getting started: http://codespeak.net/pypy/dist/pypy/doc/getting-started.html

PyPy Documentation: http://codespeak.net/pypy/dist/pypy/doc/

PyPy Homepage: http://codespeak.net/pypy/

The interpreter and object model implementations shipped with the 0.9 version can run on their own and implement the core language features of Python as of CPython 2.4. However, we still do not recommend using PyPy for anything else than for education, playing or research purposes.

Ongoing work and near term goals

The Just-in-Time compiler and other performance improvements will be one of the main topics of the next few months’ work, along with finishing the logic object space.

Project Details

PyPy has been developed during approximately 20 coding sprints across Europe and the US. It continues to be a very dynamically and incrementally evolving project with many of these one-week workshops to follow.

PyPy has been a community effort from the start and it would not have got that far without the coding and feedback support from numerous people. Please feel free to give feedback and raise questions.

contact points: http://codespeak.net/pypy/dist/pypy/doc/contact.html

have fun,

the pypy team, (Armin Rigo, Samuele Pedroni, Holger Krekel, Christian Tismer, Carl Friedrich Bolz, Michael Hudson, and many others: http://codespeak.net/pypy/dist/pypy/doc/contributor.html)

PyPy development and activities happen as an open source project and with the support of a consortium partially funded by a two year European Union IST research grant. The full partners of that consortium are:

Heinrich-Heine University (Germany), AB Strakt (Sweden) merlinux GmbH (Germany), tismerysoft GmbH (Germany) Logilab Paris (France), DFKI GmbH (Germany) ChangeMaker (Sweden), Impara (Germany)

pypy-0.8.0: Translatable compiler/parser and some more speed

The PyPy development team has been busy working and we’ve now packaged our latest improvements, completed work and new experiments as version 0.8.0, our third public release.

The highlights of this third release of PyPy are:

4.1. Historical release notes
PyPy, Release 7.2.0

- Translatable parser and AST compiler. PyPy now integrates its own compiler based on Python own ‘compiler’ package but with a number of fixes and code simplifications in order to get it translated with the rest of PyPy. This makes using the translated pypy interactively much more pleasant, as compilation is considerably faster than in 0.7.0.

- Some Speed enhancements. Translated PyPy is now about 10 times faster than 0.7 but still 10-20 times slower than CPython on pystones and other benchmarks. At the same time, language compliance has been slightly increased compared to 0.7 which had already reached major CPython compliance goals.

- Some experimental features are now translatable. Since 0.6.0, PyPy shipped with an experimental Object Space (the part of PyPy implementing Python object operations and manipulation) implementing lazily computed objects, the “Thunk” object space. With 0.8.0 this object space can also be translated preserving its feature additions.

What is PyPy (about)?

PyPy is a MIT-licensed research-oriented reimplementation of Python written in Python itself, flexible and easy to experiment with. It translates itself to lower level languages. Our goals are to target a large variety of platforms, small and large, by providing a compilation toolsuite that can produce custom Python versions. Platform, Memory and Threading models are to become aspects of the translation process - as opposed to encoding low level details into a language implementation itself. Eventually, dynamic optimization techniques - implemented as another translation aspect - should become robust against language changes.

Note that PyPy is mainly a research and development project and does not by itself focus on getting a production-ready Python implementation although we do hope and expect it to become a viable contender in that area sometime next year.

PyPy is partially funded as a research project under the European Union’s IST programme.

Where to start?

Getting started: http://codespeak.net/pypy/dist/pypy/doc/getting-started.html

PyPy Documentation: http://codespeak.net/pypy/dist/pypy/doc/

PyPy Homepage: http://codespeak.net/pypy/

The interpreter and object model implementations shipped with the 0.8 version can run on their own and implement the core language features of Python as of CPython 2.4. However, we still do not recommend using PyPy for anything else than for education, playing or research purposes.

Ongoing work and near term goals

At the last sprint in Paris we started exploring the new directions of our work, in terms of extending and optimizing PyPy further. We started to scratch the surface of Just-In-Time compiler related work, which we still expect will be the major source of our future speed improvements and some successful amount of work has been done on the support needed for stackless-like features.

This release also includes the snapshots in preliminary or embryonic form of the following interesting but yet not completed sub projects:

- The OOtyper, a RTyper variation for higher-level backends (Squeak, …)
- A JavaScript backend
- A limited (PPC) assembler backend (this related to the JIT)
• some bits for a socket module

PyPy has been developed during approximately 16 coding sprints across Europe and the US. It continues to be a very
dynamically and incrementally evolving project with many of these one-week workshops to follow.

PyPy has been a community effort from the start and it would not have got that far without the coding and feedback
support from numerous people. Please feel free to give feedback and raise questions.

  contact points: http://codespeak.net/pypy/dist/pypy/doc/contact.html

have fun,

  the pypy team, (Armin Rigo, Samuele Pedroni, Holger Krekel, Christian Tismer, Carl Friedrich Bolz,
  Michael Hudson, and many others: http://codespeak.net/pypy/dist/pypy/doc/contributor.html)

PyPy development and activities happen as an open source project and with the support of a consortium partially
funded by a two year European Union IST research grant. The full partners of that consortium are:

  Heinrich-Heine University (Germany), AB Strakt (Sweden) merlinux GmbH (Germany), tismerysoft
  GmbH (Germany) Logilab Paris (France), DFKI GmbH (Germany) ChangeMaker (Sweden), Impara
  (Germany)

**pypy-0.7.0: first PyPy-generated Python Implementations**

What was once just an idea between a few people discussing on some nested mailing list thread and in a pub became
reality . . . the PyPy development team is happy to announce its first public release of a fully translatable self contained
Python implementation. The 0.7 release showcases the results of our efforts in the last few months since the 0.6 preview
release which have been partially funded by the European Union:

  • whole program type inference on our Python Interpreter implementation with full translation to two different
    machine-level targets: C and LLVM
  • a translation choice of using a refcounting or Boehm garbage collectors
  • the ability to translate with or without thread support
  • very complete language-level compliance with CPython 2.4.1

**What is PyPy (about)?**

PyPy is a MIT-licensed research-oriented reimplementaton of Python written in Python itself, flexible and easy to
experiment with. It translates itself to lower level languages. Our goals are to target a large variety of platforms,
small and large, by providing a compilation toolsuite that can produce custom Python versions. Platform, Memory
and Threading models are to become aspects of the translation process - as opposed to encoding low level details into
a language implementation itself. Eventually, dynamic optimization techniques - implemented as another translation
aspect - should become robust against language changes.

Note that PyPy is mainly a research and development project and does not by itself focus on getting a production-ready
Python implementation although we do hope and expect it to become a viable contender in that area sometime next
year.

**Where to start?**

  Getting started: http://codespeak.net/pypy/dist/pypy/doc/getting-started.html

PyPy Documentation: http://codespeak.net/pypy/dist/pypy/doc/

PyPy Homepage: http://codespeak.net/pypy/
The interpreter and object model implementations shipped with the 0.7 version can run on their own and implement the core language features of Python as of CPython 2.4. However, we still do not recommend using PyPy for anything else than for education, playing or research purposes.

Ongoing work and near term goals

PyPy has been developed during approximately 15 coding sprints across Europe and the US. It continues to be a very dynamically and incrementally evolving project with many one-week meetings to follow. You are invited to consider coming to the next such meeting in Paris mid October 2005 where we intend to plan and head for an even more intense phase of the project involving building a JIT-Compiler and enabling unique features not found in other Python language implementations.

PyPy has been a community effort from the start and it would not have got that far without the coding and feedback support from numerous people. Please feel free to give feedback and raise questions.

contact points: http://codespeak.net/pypy/dist/pypy/doc/contact.html
contributor list: http://codespeak.net/pypy/dist/pypy/doc/contributor.html

have fun,

the pypy team, of which here is a partial snapshot of mainly involved persons:

Armin Rigo, Samuele Pedroni, Holger Krekel, Christian Tismer, Carl Friedrich Bolz, Michael Hudson, Eric van Riet Paap, Richard Emslie, Anders Chrigstroem, Anders Lehmann, Ludovic Aubry, Adrien Di Mascio, Niklaus Haldimann, Jacob Hallen, Bea During, Laura Creighton, and many contributors ...

PyPy development and activities happen as an open source project and with the support of a consortium partially funded by a two year European Union IST research grant. Here is a list of the full partners of that consortium:

Heinrich-Heine University (Germany), AB Strakt (Sweden) merlinux GmbH (Germany), tismerysoft GmbH(Germany) Logilab Paris (France), DFKI GmbH (Germany) ChangeMaker (Sweden), Impara (Germany)

The PyPy 0.6 release

The PyPy Development Team is happy to announce the first public release of PyPy after two years of spare-time and half a year of EU funded development. The 0.6 release is eminently a preview release.

What it is and where to start

Getting started: getting-started.html

PyPy Documentation: index.html

PyPy Homepage: http://pypy.org

PyPy is a MIT-licensed reimplementaton of Python written in Python itself. The long term goals are an implementation that is flexible and easy to experiment with and retarget to different platforms (also non-C ones) and such that high performance can be achieved through high-level implementations of dynamic optimization techniques.

The interpreter and object model implementations shipped with 0.6 can be run on top of CPython and implement the core language features of Python as of CPython 2.3. PyPy passes around 90% of the Python language regression tests that do not depend deeply on C-extensions. Some of that functionality is still made available by PyPy piggy-backing on the host CPython interpreter. Double interpretation and abstractions in the code-base make it so that PyPy running on CPython is quite slow (around 2000x slower than CPython ), this is expected.
This release is intended for people that want to look and get a feel into what we are doing, playing with interpreter and perusing the codebase. Possibly to join in the fun and efforts.

Interesting bits and highlights

The release is also a snap-shot of our ongoing efforts towards low-level translation and experimenting with unique features.

- By default, PyPy is a Python version that works completely with new-style-classes semantics. However, support for old-style classes is still available. Implementations, mostly as user-level code, of their metaclass and instance object are included and can be re-made the default with the --oldstyle option.

- In PyPy, bytecode interpretation and object manipulations are well separated between a bytecode interpreter and an object space which implements operations on objects. PyPy comes with experimental object spaces augmenting the standard one through delegation:
  - an experimental object space that does extensive tracing of bytecode and object operations;
  - the ‘thunk’ object space that implements lazy values and a ‘become’ operation that can exchange object identities.

These spaces already give a glimpse in the flexibility potential of PyPy. See demo/fibonacci.py and demo/sharedref.py for examples about the ‘thunk’ object space.

- The 0.6 release also contains a snapshot of our translation-efforts to lower level languages. For that we have developed an annotator which is capable of inferring type information across our code base. The annotator right now is already capable of successfully type annotating basically all of PyPy code-base, and is included with 0.6.

- From type annotated code, low-level code needs to be generated. Backends for various targets (C, LLVM,…) are included; they are all somehow incomplete and have been and are quite in flux. What is shipped with 0.6 is able to deal with more or less small/medium examples.

Ongoing work and near term goals

Generating low-level code is the main area we are hammering on in the next months; our plan is to produce a PyPy version in August/September that does not need to be interpreted by CPython anymore and will thus run considerably faster than the 0.6 preview release.

PyPy has been a community effort from the start and it would not have got that far without the coding and feedback support from numerous people. Please feel free to give feedback and raise questions.

contact points: http://pypy.org/contact.html

contributor list: contributor.html

have fun,

Armin Rigo, Samuele Pedroni,
Holger Krekel, Christian Tismer,
Carl Friedrich Bolz

PyPy development and activities happen as an open source project and with the support of a consortium funded by a two year EU IST research grant. Here is a list of partners of the EU project:

Heinrich-Heine University (Germany), AB Strakt (Sweden)
merlinux GmbH (Germany), tismerysoft GmbH(Germany)
Logilab Paris (France), DFKI GmbH (Germany)
4.1.3 CPython 3.3 compatible versions

PyPy3 v5.5.0

We’re pleased to announce the release of PyPy3 v5.5.0. Coming four months after PyPy3.3 v5.2, it improves compatibility with Python 3.3 (3.3.5). We strongly recommend updating from previous PyPy3 versions.

We would like to thank all of the people who donated to the py3k proposal for supporting the work that went into this release.

You can download the PyPy3.3 v5.5.0 release here:

http://pypy.org/download.html#python-3-3-5-compatible-pypy3-3-v5-5

Highlights

• Improved Python 3.3.5 support.
  – os.get_terminal_size(), time.monotonic(), str.casefold()
  – faulthandler module
  – There are still some missing features such as a PEP 393-like space efficient string representation and known issues including performance regressions (e.g. issue #2305). The focus for this release has been updating to 3.3 compatibility. Windows is also not yet supported.
• ensurepip is also included (it’s only included in CPython 3 >= 3.4).

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.10 and 3.3.5. It’s fast due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports:

• x86 machines on most common operating systems except Windows (Linux 32/64, Mac OS X 64, OpenBSD, FreeBSD),
• newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
• big- and little-endian variants of PPC64 running Linux,
• s390x running Linux

Please try it out and let us know what you think. We welcome feedback, we know you are using PyPy, please tell us about it!

Cheers

The PyPy Team
PyPy3 v5.2 alpha 1

We're pleased to announce the first alpha release of PyPy3 v5.2. This is the first release of PyPy which targets Python 3.3 (3.3.5) compatibility.

We would like to thank all of the people who donated to the py3k proposal for supporting the work that went into this and future releases.

You can download the PyPy3 v5.2 alpha 1 release here:

http://pypy.org/download.html#python-3-3-5-compatible-pypy3-3-v5-2

Highlights

• Python 3.3.5 support!
  – Being an early alpha release, there are some missing features such as a PEP 393-like space efficient string representation and known issues including performance regressions (e.g. issue #2305). The focus for this release has been updating to 3.3 compatibility. Windows is also not yet supported.
  • ensurepip is also included (it’s only included in CPython 3 >= 3.4).

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.10 and one day 3.3.5. It’s fast due to its integrated tracing JIT compiler.

We also welcome developers of other dynamic languages to see what RPython can do for them.

This release supports:

• x86 machines on most common operating systems except Windows (Linux 32/64, Mac OS X 64, OpenBSD, FreeBSD),
• newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux,
• big- and little-endian variants of PPC64 running Linux,
• s390x running Linux

Please try it out and let us know what you think. We welcome feedback, we know you are using PyPy, please tell us about it!

We’d especially like to thank these people for their contributions to this release:

Manuel Jacob, Ronan Lamy, Mark Young, Amaury Forgeot d’Arc, Philip Jenvey, Martin Matusiak, Vasily Kuznetsov, Matti Picus, Armin Rigo and many others.

Cheers

The PyPy Team

4.1.4 CPython 3.2 compatible versions

PyPy3 2.4 - Snow White

We’re pleased to announce PyPy3 2.4, which contains significant performance enhancements and bug fixes.

You can download the PyPy3 2.4.0 release here:
We would like to thank our donors for the continued support of the PyPy project, and for those who donate to our three sub-projects. We’ve shown quite a bit of progress, but we’re slowly running out of funds. Please consider donating more, or even better convince your employer to donate, so we can finish those projects! The three sub-projects are:

- **Py3k** (supporting Python 3.x): *This is a Python 3.2.5 compatible* version we call PyPy3 2.4, and we are working toward a Python 3.3 compatible version
- **STM** (software transactional memory): We have released a first working version, and continue to try out new promising paths of achieving a fast multithreaded Python
- **NumPy** which requires installation of our fork of upstream numpy, available on bitbucket

**What is PyPy?**

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7 or 3.2.5. It’s fast (pypy 2.4 and cpython 2.7.x performance comparison) due to its integrated tracing JIT compiler.

This release supports **x86** machines on most common operating systems (Linux 32/64, Mac OS X 64, Windows, and OpenBSD), as well as newer **ARM** hardware (ARMv6 or ARMv7, with VFPv3) running Linux.

While we support 32 bit python on Windows, work on the native Windows 64 bit python is still stalling, we would welcome a volunteer to handle that.

**PyPy3 Highlights**

Issues reported with our previous release were fixed after reports from users on our new issue tracker at https://bitbucket.org/pypy/pypy/issues or on IRC at #pypy. Here is a summary of the user-facing PyPy3 specific changes:

- Better Windows compatibility, e.g. the nt module functions _getfinalpathname & _getfileinformation are now supported (the former is required for the popular pathlib library for example)
- Various fsencode PEP 383 related fixes to the posix module (readlink, uname, ttyname and ctermid) and improved locale handling
- Switched default binary name os POSIX distributions to ‘pypy3’ (which symlinks to to ‘pypy3.2’)
- Fixed a couple different crashes related to parsing Python 3 source code

**Further Highlights (shared w/ PyPy2)**

Benchmarks improved after internal enhancements in string and bytearray handling, and a major rewrite of the GIL handling. This means that external calls are now a lot faster, especially the CFFI ones. It also means better performance in a lot of corner cases with handling strings or bytearrays. The main bugfix is handling of many socket objects in your program which in the long run used to “leak” memory.

We fixed a memory leak in IO in the sandbox code

We welcomed more than 12 new contributors, and conducted two Google Summer of Code projects, as well as other student projects not directly related to Summer of Code.

- Reduced internal copying of bytearray operations
- Tweak the internal structure of StringBuilder to speed up large string handling, which becomes advantageous on large programs at the cost of slightly slower small benchmark type programs.
• Boost performance of thread-local variables in both unjitted and jitted code, this mostly affects errno handling on linux, which makes external calls faster.
• Move to a mixed polling and mutex GIL model that make mutlithreaded jitted code run much faster
• Optimize errno handling in linux (x86 and x86-64 only)
• Remove ctypes pythonapi and ctypes.PyDLL, which never worked on PyPy
• Classes in the ast module are now distinct from structures used by the compiler, which simplifies and speeds up translation of our source code to the PyPy binary interpreter
• Win32 now links statically to zlib, expat, bzip, and openssl-1.0.1i. No more missing DLLs
• Many issues were resolved since the 2.3.1 release in June

We have further improvements on the way: rpython file handling, numpy linalg compatibility, as well as improved GC and many smaller improvements.

Please try it out and let us know what you think. We especially welcome success stories, we know you are using PyPy, please tell us about it!

Cheers
The PyPy Team

PyPy3 2.3.1 - Fulcrum

We’re pleased to announce the first stable release of PyPy3. PyPy3 targets Python 3 (3.2.5) compatibility.

We would like to thank all of the people who donated to the py3k proposal for supporting the work that went into this.

You can download the PyPy3 2.3.1 release here:

http://pypy.org/download.html#pypy3-2-3-1

Highlights

• The first stable release of PyPy3: support for Python 3!
• The stdlib has been updated to Python 3.2.5
• Additional support for the u’unicode’ syntax (PEP 414) from Python 3.3
• Updates from the default branch, such as incremental GC and various JIT improvements
• Resolved some notable JIT performance regressions from PyPy2:
• Re-enabled the previously disabled collection (list/dict/set) strategies
• Resolved performance of iteration over range objects
• Resolved handling of Python 3’s exception __context__ unnecessarily forcing frame object overhead

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.6 or 3.2.5. It’s fast due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64, Windows, and OpenBSD, as well as newer ARM hardware (ARMv6 or ARMv7, with VFPv3) running Linux.
PyPy, Release 7.2.0

While we support 32 bit python on Windows, work on the native Windows 64 bit python is still stalling, we would welcome a volunteer to handle that.

How to use PyPy?

We suggest using PyPy from a virtualenv. Once you have a virtualenv installed, you can follow instructions from pypy documentation on how to proceed. This document also covers other installation schemes.

Cheers, the PyPy team

PyPy3 2.1 beta 1

We're pleased to announce the first beta of the upcoming 2.1 release of PyPy3. This is the first release of PyPy which targets Python 3 (3.2.3) compatibility.

We would like to thank all of the people who donated to the py3k proposal for supporting the work that went into this and future releases.

You can download the PyPy3 2.1 beta 1 release here:

http://pypy.org/download.html#pypy3-2-1-beta-1

Highlights

- The first release of PyPy3: support for Python 3, targetting CPython 3.2.3!
  - There are some known issues including performance regressions (issues #1540 & #1541) slated to be resolved before the final release.

What is PyPy?

PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.3 or 3.2.3. It’s fast due to its integrated tracing JIT compiler.

This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. Also this release supports ARM machines running Linux 32bit - anything with ARMv6 (like the Raspberry Pi) or ARMv7 (like Beagleboard, Chromebook, Cubieboard, etc.) that supports VFPv3 should work.

Windows 64 work is still stalling and we would welcome a volunteer to handle that.

How to use PyPy?

We suggest using PyPy from a virtualenv. Once you have a virtualenv installed, you can follow instructions from pypy documentation on how to proceed. This document also covers other installation schemes.

Cheers, the PyPy team

4.2 Changelogs

4.2.1 CPython 2.7 compatible versions
What's new in PyPy2.7 7.1+

Fix typo
Remove rpython.jit.metainterpreter.typesystem and clean up related code in rpython/jit/
Add DateTime_FromTimestamp and Date_FromTimestamp
Test and reduce the probability of a deadlock when acquiring a semaphore by moving global state changes closer to the actual acquire.
Make the shadowstack size more dynamic
Move _ssl and _hashlib from rpython to a cffi-based module, like on python3. Reduces the number of problematic linked-in libraries (libssl, libcrypto)
Fix a bug that prevent memory-tracking in vmprof working on PyPy.
Cleanup optimizeopt

What's new in PyPy2.7 7.0+

Make sure zlib decompressobj have their streams deallocated immediately on flush.
Fix calling copy on already-flushed compressobj.
The zlib module’s compressobj and decompressobj now expose copy methods as they do on CPython.
Improve performance of long operations where one of the operands fits into an int.
Improve register allocation in the JIT.
Implement rlib.jit.promote_unicode to complement promote_string
Use utf8 internally to represent unicode, with the goal of never using rpython-level unicode
Since _ctypes is implemented in pure python over libffi, add interfaces and methods to support the buffer interface from python. Specifically, add a __pypy__.newmemoryview function to create a memoryview and extend the use of the PyPy-specific __buffer__ class method.

What's new in PyPy2.7 6.0+

Main items: vastly better template resolution and improved performance. In detail: upgrade to backend 1.4, improved handling of templated methods and functions (in particular automatic deduction of types), improved pythonization interface, range of compatibility fixes for Python3, free functions now take fast libffi path when possible, moves for strings (incl. from Python str), easier/faster handling of std::vector by numpy, improved and faster object identity preservation
Make sure ‘blocking-ness’ of socket is set along with default timeout
Include crypt.h for crypt() on Linux
Log additional gc-minor and gc-collect-step info in the PYPYLOG
The reverse-debugger branch has been merged. For more information, see https://bitbucket.org/pypy/revdb
Small refactoring in the Python parser.
Backport CPython fix for possible shell injection issue in distutils.spawn, https://bugs.python.org/issue34540
Enable use of unicode file names in dlopen
Backport CPython fix for thread.RLock

4.2. Changelogs
PyPy, Release 7.2.0

Make GC hooks measure time in seconds (as opposed to an opaque unit).
Update most test_lib_pypy/ tests and move them to extra_tests/.
Make it possible to manually manage the GC by using a combination of gc.disable() and gc.collect_step(). Make sure to write a proper release announcement in which we explain that existing programs could leak memory if they run for too much time between a gc.disable()/gc.enable()

What’s new in PyPy2.7 5.10+

Big refactoring of some cpyext code, which avoids a lot of nonsense when calling C from Python and vice-versa: the result is a big speedup in function/method calls, up to 6 times faster.
Support tzinfo field on C-API datetime objects, fixes latest pandas HEAD
Fix a corner case of mapdict: When an instance is used like a dict (using setattr and getattr, or .__dict__) and a lot of attributes are added, then the performance using mapdict is linear in the number of attributes. This is now fixed (by switching to a regular dict after 80 attributes).
When using cpyext, improve the speed of passing certain objects from PyPy to C code, most notably None, True, False, types, all instances of C-defined types. Before, a dict lookup was needed every time such an object crossed over, now it is just a field read.
Improve datetime + timedelta performance.
Improve way to describe memory
Allow compilaiton with Visual Studio 2017 compiler suite on windows
Refactor cpyext slots.
Speed up branchy code that does a lot of function inlining by saving one call to read the TLS in most bridges.
Refactor in rpython signatures
Store error state thread-locally in executioncontext, fixes issue #2764
Optimize Py*_Check for Bool, Float, Set. Also refactor and simplify W_PyCWrapperObject which is used to call slots from the C-API, greatly improving microbenchmarks in https://github.com/antocuni/cpyext-benchmarks
Fix two (unrelated) JIT bugs manifesting in the re module:
  • green fields are broken and were thus disabled, plus their usage removed from the _sre implementation
  • in rare “trace is too long” situations, the JIT could break behaviour arbitrarily.
Be more efficient about JIT hooks. Make it possible for the frontend to declare that jit hooks are currently not enabled at all. in that case, the list of ops does not have to be created in the case of the on_abort hook (which is expensive).
Improve speed of Python parser, improve ParseError messages slightly.
Work around possible bugs in upstream ioctl users, like CPython allocate at least 1024 bytes for the arg in calls to ioctl(fd, request, arg). Fixes issue #2776
Fix for python-level classes that inherit from C-API types, previously the w_obj was not necessarily preserved throughout the lifetime of the pyobj which led to cases where instance attributes were lost. Fixes issue #2793
Improve line offsets that are reported by SyntaxError. Improve error messages for a few situations, including mismatched parenthesis.
Fix a rare GC bug that was introduced more than one year ago, but was not diagnosed before issue #2752.
Introduce GC hooks, as documented in doc/gc_info.rst
Improve GC hooks
Update backend to 0.6.0 and support exceptions through wrappers

**What's new in PyPy2.7 5.10**

Cleanup and improve cppyy packaging

Add a smartstrip tool, which can optionally keep the debug symbols in a separate file, instead of just stripping them away. Use it in packaging

Fix failures on FreeBSD, contributed by David Naylor as patches on the issue tracker (issues 2694, 2695, 2696, 2697)

Run extra_tests/ in buildbot

Upgrade the _vmprof backend to vmprof 0.4.10

Fix a vmprof+continulets (i.e. greenelts, eventlet, gevent, ...)

Make it possible to declare that the hash function of an r_dict is fast in RPython.

**What's new in PyPy2.7 5.9**

In previous versions of PyPy, instance.method would return always the same bound method object, when gotten out of the same instance (as far as is and id() can tell). CPython doesn’t do that. Now PyPy, like CPython, returns a different bound method object every time. For type.method, PyPy2 still returns always the same unbound method object; CPython does it for built-in types but not for user-defined types.

The two cffi-* branches are part of the upgrade to cffi 1.11.

Indexing into char* behaves differently than CPython

Improve and fix issues with vmprof

CPyext PyListObject.pop must return the value

If tp_hash is PyObject_HashNotImplemented, set obj.__dict__["__hash__"] to None

Renaming of cppyy to _cppyy. The former is now an external package installable with pip install cppyy.

At the end of translation, run attr -q -s pax.flags -V m on PAX-enabled systems on the produced binary. This seems necessary because PyPy uses a JIT.

Improve bytearray performance (backported from py3.5)

Fix the bounds in the GC when allocating a lot of objects with finalizers, fixes issue #2590

Small improvement to optimize list accesses with constant indexes better by throwing away information about them less eagerly.

More information is retained into a bridge: knowledge about the content of arrays (at fixed indices) is stored in guards (and thus available at the beginning of bridges). Also, some better feeding of information about known fields of constant objects into bridges.

Add support for leakfinder in cpyext tests (disabled for now, due to too many failures).

Added _swappedbytes_ support for ctypes.Structure

Convert many Py*_Check cpyext functions into macros, like CPython.

Explicitly use Py_ssize_t as the Signed type in pypy c-api

Differentiate the code to call METH_NOARGS, METH_O and METH_VARARGS in cpyext: this allows to write specialized code which is much faster than previous completely generic version. Moreover, let the JIT to look inside the cpyext module: the net result is that cpyext calls are up to 7x faster. However, this is true only for very simple situations: in all real life code, we are still much slower than CPython (more optimizations to come)

4.2. Changelogs
What's new in PyPy2.7 5.8+

Add cpyext interfaces for PyModule_New

Correctly handle `dict.pop` where the `pop` key is not the same type as the `dict`'s and `pop` is called with a default (will be part of release 5.7.1)

Fix missing tp_new on `w_object` called through multiple inheritance (will be part of release 5.7.1)

PyPy support to profile native frames in vmprof.

Performance tweaks in the x86 JIT-generated machine code: rarely taken blocks are moved off-line. Also, the temporary register used to contain large constants is reused across instructions.

Refactor rpython.rtyper.controllerentry.

Internal refactoring of buffers and memoryviews. Memoryviews will now be accepted in a few more places, e.g. in `compile()`.

Redo much of the refcount semantics in `PyList_{SG}etItem` to closer match CPython and ensure the same `PyObject` stored in the list can be later retrieved

Implement `Py_EnterRecursiveCall` and associated functions

Remove faulty fastpath from `ctypes`

Passing a buffersize of 0 to `socket.getsockopt`

Improve the performance of `struct.pack` and `struct.pack_into` by using `raw_store` or `gc_store_indexed` whenever possible. Moreover, enable the existing `struct.unpack` fast path to all the existing buffer types, whereas previously it was enabled only for strings

Add profile-based optimization option `profopt`, and specify training data via `profoptpath`

What's new in PyPy2.7 5.7

Since a while now, PyPy preserves the order of dictionaries and sets. However, the set literal syntax `{x, y, z}` would by mistake build a set with the opposite order: `set([z, y, x])`. This has been fixed. Note that CPython is inconsistent too: in 2.7.12, `{5, 5.0}` would be `set([5])`, but in 2.7.trunk it is `set([5.0])`. PyPy's behavior changed in exactly the same way because of this fix.

Any uncaught RPython exception (from a PyPy bug) is turned into an app-level SystemError. This should improve the lot of users hitting an uncaught RPython error.

Try to improve the consistency of RPython annotation unions.

Clean-ups in the `jit.optimizeopt`

Add `jit.conditional_call_elidable()`, a way to tell the JIT “conditionally call this function” returning a result.

Refactor `FunctionDesc specialize()` and related code (RPython annotator).

Assign `tp_doc` to the new `TypeObject`'s type dictionary `__doc__` key so it will be picked up by app-level objects of that type

Module cppyy now uses cling as its backend (Reflex has been removed). The user-facing interface and main developer tools (genreflex, selection files, class loader, etc.) remain the same. A `libc++_backend.so` library is still needed but is now available through PyPI with pip: `PyPy-cppyy-backend`.

The Cling-backend brings support for modern C++ (11, 14, etc.), dynamic template instantiations, and improved integration with CFFI for better performance. It also provides interactive C++ (and bindings to that).

Improve the performance of `PyDict_Next`. When trying `PyDict_Next` on a typedef dict, the test exposed a problem converting a `GetSetProperty` to a `PyGetSetDescrObject`. The other direction seem to be fully
implemented. This branch made a minimal effort to convert the basic fields to avoid segfaults, but trying to use the PyGetSetDescrObject will probably fail.

Updated the implementation to match CPython 2.7.13 instead of 2.7.13.

Fix PyObject_GetBuffer and PyMemoryView_GET_BUFFER, which leaked memory and held references. Add a finalizer to CPyBuffer, add a PyMemoryViewObject with a PyBuffer attached so that the call to PyMemoryView_GET_BUFFER does not leak a PyBuffer-sized piece of memory. Properly call bf_releasebuffer when not NULL.

Support translations of cpyext with the Boehm GC (for special cases like revdb).

Implement StringBuffer.get_raw_address (missing feature for the buffer protocol). More generally it is now possible to obtain the address of any object (if it is readonly) without pinning it.

Refactor cpyext initialisation.

Fix a test failure introduced by strbuf-as-buffer

Do not recreate the object in PyMemoryView_FromBuffer, rather pass it to the returned PyMemoryViewObject, to take ownership of it. Fixes a ref leak.

Give (almost!) all GetSetProperties a valid __objclass__.

Improve mixing app-level classes in c-extensions, especially if the app-level class has a tp_new or tp_dealloc. The issue is that c-extensions expect all the method slots to be filled with a function pointer, where app-level will search up the mro for an appropriate function at runtime. With this branch we now fill many more slots in the c-extension type objects. Also fix for c-extension type that calls tp_hash during initialization (str, unicode types), and fix instantiating c-extension types from built-in classes by enforcing an order of instantiation.

rffi structures in cpyext can now be created by parsing simple C headers. Additionally, the cts object that holds the parsed information can act like cff’s ffi objects, with the methods cts.cast() and cts.gettype().

Don’t freeze hashes in the translated pypy. In practice, that means that we can now translate PyPy with the option --hash=siphash24 and get the same hashes as CPython 3.5, which can be randomized (in a cryptographically good way). It is the default in PyPy3. The default of PyPy2 remains unchanged: there are user programs out there that depend on constant hashes (or even sometimes on specific hash results).

Our dicts, which are always ordered, now have an extra “method” for Python 3.x which moves an item to first or last position. In PyPy 3.5 it is the standard OrderedDict.move_to_end() method, but the behavior is also available on Python 2.x or for the dict type by calling __pypy__.move_to_end(dict, key, last=True).

Improve the optimization of branchy Python code by retaining more information across failing guards.

Internal refactoring of space.wrap(), which is now replaced with explicitly-typed methods. Notably, there are now space.newbytes() and space.newtext(): these two methods are identical on PyPy 2.7 but not on PyPy 3.x. The latter is used to get an app-level unicode string by decoding the RPython string, assumed to be utf-8.

Fix for ctypes.c_bool-returning ctypes functions

Improve handling of the Py3-style buffer slots in cpyext: fix memoryviews keeping objects alive forever (missing decref), and make sure that bf_releasebuffer is called when it should, e.g. from PyBuffer_Release.

Fix bug (bad reported info) when asked to translate SyntaxWarning to SyntaxError.

Improve the optimization of branchy Python code by retaining more information across failing guards. This is done by appending some carefully encoded extra information into the resume code.

Two changes that together bring the performance of shadowstack close to asmgcc—close enough that we can now make shadowstack the default even on Linux. This should remove a whole class of rare bugs introduced by asmgcc.
**What’s new in PyPy2.7 5.6**

Backport rpython changes made directly on the py3k and py3.5 branches.

Implement PyObject_GetBuffer, PyMemoryView_GET_BUFFER, and handles memoryviews in numpppy

Improve merging of virtual states in the JIT in order to avoid jumping to the preamble. Achieved by allocating virtual objects where non-virtuals are expected.

JIT residual calls: if the called function starts with a fast-path like “if x.foo != 0: return x.foo”, then inline the check before doing the CALL. For now, string hashing is about the only case.

The compiled pypy now looks for its lib-python/lib_pypy path starting from the location of the libpypy-c instead of the executable. This is arguably more consistent, and also it is what occurs anyway if you’re embedding pypy. Linux distribution packagers, take note! At a minimum, the libpypy-c.so must really be inside the path containing lib-python and lib_pypy. Of course, you can put a symlink to it from somewhere else. You no longer have to do the same with the pypy executable, as long as it finds its libpypy-c.so library.

CPython allows warning.warn(('something', 1), Warning), on PyPy this produced a “expected a readable buffer object” error. Test and fix.

CPython rejects ‘a’.strip(buffer(‘ ’)); only None, str or unicode are allowed as arguments. Test and fix for str and unicode

Port the ‘faulthandler’ module to PyPy default. This module is standard in Python 3.3 but can also be installed from CPython >= 2.6 from PyPI.

Refactor cpyext testing to be more pypy3-friendly.

Improve the error message when the user forgot the “self” argument of a method.

Change the timeit module: it now prints the average time and the standard deviation over 7 runs by default, instead of the minimum. The minimum is often misleading.

Make optimiseopt iterative instead of recursive so it can be reasoned about more easily and debugging is faster.

Update stdlib to version 2.7.12

Improve support for new buffer interface in cpyext, bf_getbuffer on built-in types still missing

Improve compatibility with CPython in the struct module. In particular, struct.unpack now returns an int whenever the returned value fits, while previously it always returned a long for certain format codes such as Q (and also I, L and q on 32 bit)

s390x implementation for vector operations used in VecOpt

PowerPC implementation for vector operations used in VecOpt

Match CPython’s stricter handling of __new__/__init__ arguments

Support for OpenSSL version 1.1 (in addition to version 1.0). Tested on Linux (1.1, 1.0), on Win32, and Mac (1.0 only)

**What’s new in PyPy2.7 5.4**

Improve CPython compatibility for is. Now code like if x is (): works the same way as it does on CPython. See http://pypy.readthedocs.io/en/latest/cpython_differences.html#object-identity-of-primitive-values-is-and-id.

Add sys.{get,set}dlopenflags, for cpyext extensions.

Resolves an issue with the generator script to build the dfa for Python syntax.
Fixes a critical issue in the register allocator and extends support on s390x. PyPy runs and translates on the s390x revisions z10 (released February 2008, experimental) and z196 (released August 2010) in addition to zEC12 and z13. To target e.g. z196 on a zEC12 machine supply `CFLAGS="-march=z196"` to your shell environment.

Implement the backend related changes for s390x.

Simplify handling of interp-level tests and make it more forward-compatible.

Sync `w_file` with the c-level `FILE*` before returning `FILE*` in `PyFile_AsFile`.

Allow `rw` access to the `char*` returned from `PyString_AS_STRING`, also refactor `PyStringObject` to look like cpython’s and allow subclassing `PyString_Type` and `PyUnicode_Type`.

Bug fix: if `socket.socket()` failed, the `socket.error` did not show the `errno` of the failing system call, but instead some random previous `errno`.

Refactor `PyTupleObject` to look like cpython’s and allow subclassing `PyTuple_Type`.

Use offsets from `PyTypeObject` to find actual c function to call rather than fixed functions, allows function override after `PyType_Ready` is called.

Avoid exhausting the stack in the JIT due to successive guard failures in the same Python function ending up as successive levels of `RPython` functions, while at app-level the traceback is very short.

Try harder to memory to the OS. See e.g. issue #2336. Note that it does not show up as a reduction of the `VIRT` column in `top`, and the `RES` column might also not show the reduction, particularly on Linux >= 4.5 or on OS/X: it uses `MADV_FREE`, which only marks the pages as returnable to the OS if the memory is low.

Fill in more slots when creating a `PyTypeObject` from a `W_TypeObject` More slots are still TBD, like `tp_print` and `richcmp`.

Align json module decode with the cpython’s impl, fixes issue 2345.

Copy CPython’s logic more closely for handling of `__instancecheck__()` and `__subclasscheck__()`.

Fixes issue 2343.

Rewrite the Win32 dependencies of ‘subprocess’ to use cffi instead of ctypes. This avoids importing ctypes in many small programs and scripts, which in turn avoids enabling threads (because ctypes creates callbacks at import time, and callbacks need threads).

The new logging facility that integrates with and adds features to vmprof.com.

Resolve issues to use the new logging facility on a 32bit system.

Trying harder to make `hash(-1)` return -2, like it does on CPython.

Log exact line positions in debug merge points.

Allocate all RPython strings with one extra byte, normally unused. It is used to hold a final zero in case we need some `char *` representation of the string, together with checks like `not can_move()` or object pinning. Main new thing that this allows: `ffi.from_buffer(string)` in CFFI. Additionally, and most importantly, CFFI calls that take directly a string as argument don’t copy the string any more—this is like CFFI on CPython.

Add a new command line option `-X track-resources` which will produce ResourceWarnings when the GC closes unclosed files and sockets.

Implement `PyObject_Realloc`.

Improve a little bit the readability of the generated C code.

Improved vmprof support: now tries hard to not miss any Python-level frame in the captured stacks, even if there is the metainterp or blackhole interp involved. Also fix the stacklet (greenlet) support.

`type.__dict__` now returns a `dict_proxy` object, like on CPython. Previously it returned what looked like a regular dict object (but it was already read-only).
Reduce the size of the generated C code by constant-folding `we_are_jitted` in non-jitcode.

Support for memoryview attributes (format, itemsize, ...). Extends the cpyext emulation layer.

Log more information to properly rebuild the redirected traces in jitviewer.

Copy `Py_TPFLAGS_CHECKTYPES`, `Py_TPFLAGS_HAVE_INPLACEOPS` when inheriting.

### What's new in PyPy2.7 5.3.1

A bug-fix release, merging these changes:

- Add include guards to `pymem.h`, fixes issue #2321
- Make `vmprof` build on OpenBSD, from pull request #456
- Fix `bytearray('').replace('a', 'ab')`, issue #2324

### What's new in PyPy2.7 5.3

Reduce the size of generated C sources.

Remove a number of options from the build process that were never tested and never set. Fix a performance bug in the method cache.

JIT: use bitstrings to compress the lists of read or written descrs that we attach to `EffectInfo`. Fixes a problem we had in `remove-objspace-options`.

Update cpyext C-API support After this branch, we are almost able to support upstream `numpyp` via cpyext, so we created (yet another) fork of `numpyp` at github.com/pypy/numpy with the needed changes. Among the significant changes to cpyext:

- allow c-snippet tests to be run with -A so we can verify we are compatible
- fix many edge cases exposed by fixing tests to run with -A
- `issequence()` logic matches cpython
- make `PyStringObject` and `PyUnicodeObject` field names compatible with cpython
- add preliminary support for `PyDateTime_*`
- support `PyComplexObject`, `PyFloatObject`, `PyDict_Merge`, `PyDictProxy`, `PyMemoryView_*`, `_Py_HashDouble`, `PyFile_AsFile`, `PyFile_FromFile`, `PyAnySet_CheckExact`, `PyUnicode_Concat`
- improve support for `PyGILState_Ensure`, `PyGILState_Release`, and thread primitives, also find a case where CPython will allow thread creation before `PyEval_InitThreads` is run, dissallow on PyPy
- create a `PyObject-specific list strategy`
- rewrite slot assignment for typeobjects
- improve tracking of `PyObject` to rpython object mapping
- support `tp_as_{number, sequence, mapping, buffer}` slots

(makes the pypy-c bigger; this was fixed subsequently by the share-cpyext-cpython-api branch)

Reduce generated code for subclasses by using the same function objects in all generated subclasses.

CPyExt tweak: instead of “GIL not held when a CPython C extension module calls PyXxx”, we now silently acquire/release the GIL. Helps with CPython C extension modules that call some PyXxx() functions without holding the GIL (arguably, they are theoretically buggy).
Get the cpyext tests to pass with “-A” (i.e. when tested directly with CPython).

Compile c snippets with -Werror in cpyext

Add gc.FinalizerQueue, documented in pypy/doc/discussion/finalizer-order.rst. It is a more flexible way to make RPython finalizers.

Use the new gc.FinalizerQueue mechanism to clean up the handling of __del__ methods. Fixes notably issue #2287. (All RPython subclasses of W_Root need to use FinalizerQueue now.)

Implement ufunc.outer on numppyp

Support pypy -v: verbose imports. It does not log as much as cpython, but it should be enough to help when debugging package layout problems.

Fix some warnings when compiling CPython C extension modules

Remove most of the _ovf, _zer and _val operations from RPython. Kills quite some code internally, and allows the JIT to do better optimizations: for example, app-level code like \( x / 2 \) or \( x \mod 2 \) can now be turned into \( x \gg 1 \) or \( x \& 1 \), even if \( x \) is possibly negative.

Generalize cpyext old-style buffers to more than just str/buffer, add support for mmap

Move _numppypy headers into a directory so they are not picked up by upstream numpy, scipy. This allows building upstream numpy and scipy in pypy via cpyext

Teach RPython JIT’s off-line traceviewer the most common debug_merge_point formats.

Enable pickling of W_PyCFunctionObject by monkeypatching pickle.Pickler.dispatch at cpyext import time

Add a way to ask “give me a raw pointer to this list’s items”. Only for resizable lists of primitives. Turns the GcArray nonmovable, possibly making a copy of it first.

Finish the work already partially merged in cpyext-for-merge. Adds support for ByteArrayObject using the nonmovable-list, which also enables buffer(bytearray(<some-list>))

*What’s new in PyPy 5.1*

The jit compiler backend implementation for the s390x architecture. The backend manages 64-bit values in the literal pool of the assembly instead of loading them as immediates. It includes a simplification for the operation ‘zero_array’. Start and length parameters are bytes instead of size.

Replace py.log with something simpler, which should speed up logging

Implemented numpy.where for 1 argument (thanks sergem)

Implement yet another strange numpy indexing compatibility; indexing by a scalar returns a scalar

Allow arguments to transpose to be sequences

Improve the tracing speed in the frontend as well as heapcache by using a more compact representation of traces

Wrap more POSIX functions in rpython.rlib.rposix.

A local clean-up in the JIT front-end.

Remove the forced minor collection that occurs when rewriting the assembler at the start of the JIT backend. This is done by emitting the ConstPtrs in a separate table, and loading from the table. It gives improved warm-up time and memory usage, and also removes annoying special-purpose code for pinned pointers.

Remove old unneeded numpy headers, what is left is only for testing. Also generate pypy_numpy.h which exposes functions to directly use micronumpy ndarray and ufuncs

Reuse rposix definition of TIMESPEC in rposix_stat, add wrapper for fstatat(). This updates the underlying rpython functions with the ones needed for the py3k branch

*4.2. Changelogs*
Add broadcast to micronumpy

**What's new in PyPy 5.0**

Fixed `_PyLong_FromByteArray()`, which was buggy.

Fixed a crash with stacklets (or greenlets) on non-Linux machines which showed up if you forget stacklets without resuming them.

Fix tests to run cleanly with -A and start to fix micronumpy for upstream numpy which is now 1.10.2

Fix the cpyext tests on OSX by linking with -flat_namespace

Refactor and improve exception analysis in the annotator.

Fix issue #2193. `isinstance(..., int) => isinstance(..., numbers.Integral)` to allow for alternate int-like implementations (e.g., `future.types.newint`)

Improve the performance of struct.unpack, which now directly reads inside the string buffer and directly casts the bytes to the appropriate type, when allowed. Unpacking of floats and doubles is about 15 times faster now, while for integer types it’s up to ~50% faster for 64bit integers.

Remove unnecessary special handling of space.wrap().

Improve the memory signature of numbering instances in the JIT. This should massively decrease the amount of memory consumed by the JIT, which is significant for most programs.

Improve the heuristic when disable trace-too-long

Make rlist’s ll_listsetslice() able to resize the target list to help simplify objspace/std/listobject.py. Was issue #2196.

A somewhat random bunch of changes and fixes following up on branch ‘anntype’. Highlights:

- Implement `@doubledispatch` decorator and use it for intersection() and difference().
- Turn isintance into a SpaceOperation
- Create a few direct tests of the fundamental annotation invariant in test_model.py
- Remove bookkeeper attribute from DictDef and ListDef.

Simplification. Backends implement too many loading instructions, only having a slightly different interface. Four new operations (gc_load/gc_load_indexed, gc_store/gc_store_indexed) replace all the commonly known loading operations

Move wrappers for OS functions from `rpython/rtyper` to `rpython/rlib` and turn them into regular RPython functions. Most RPython-compatible `os.*` functions are now directly accessible as `rpython.rposix.*`.

Simplify a bit the GIL handling in non-jitted code. Fixes issue #2205.

Trivial cleanups in flowspace.operation : fix comment & duplicated method

Add a test for pre-existing AF_NETLINK support. Was part of issue #1942.

Trivial misc cleanups: typo, whitespace, obsolete comments

Fix the cryptic exception message when attempting to use extended slicing in rpython. Was issue #2211.

Optimize the case where, in a new C-created thread, we keep invoking short-running Python callbacks. (CFFI on CPython has a hack to achieve the same result.) This can also be seen as a bug fix: previously, thread-local objects would be reset between two such calls.

Optimize global lookups.

Fix SSL tests by importing cpython’s patch

Remove pure variants of `getfield_gc_*` operations from the JIT. Relevant optimizations instead consult the field descriptor to determine the purity of the operation. Additionally, pure `getfield` operations are now handled entirely by `rpython/jit/metainterop/optimizeopt/heap.py` rather than `rpython/jit/metainterop/optimizeopt/pure.py`, which can result in better codegen for traces containing a large number of pure `getfield` operations.

Try to ensure that no new functions get annotated during the ‘source_c’ phase. Refactor sandboxing to operate at a higher level.

Refactor vmprof to work cross-operating-system.

Separate structmember.h from Python.h Also enhance creating api functions to specify which header file they appear in (previously only pypy_decl.h)

Refactor register_external(), remove running_on_llinterp mechanism and apply sandbox transform on externals at the end of annotation.

vmprof should work on Windows.

When creating instances and adding attributes in several different orders depending on some condition, the JIT would create too much code. This is now fixed.

Improve CPython C API support, which means lxml now runs unmodified (after removing pypy hacks, pending pull request)

Look inside tuple hash, improving mdp benchmark

Compress resume data, saving 10-20% of memory consumed by the JIT

Fix boolean-array indexing in micronumpy

Support ndarray.partition() as an app-level function numpy.core._partition_use, provided as a cffi wrapper to upstream’s implementation in the pypy/numpy repo

What’s new in PyPy 4.0.1

Use pkg-config to find ssl headers on OS-X

The PPC machines now support the _continuation module (stackless, greenlets)

Document that libgdbm-dev is required for translation/packaging

Ensure that ndarray conversion from int16->float16->float32->float16->int16 preserves all int16 values, even across nan conversions. Also fix argmax, argmin for nan comparisons

Support common use-cases for __array_interface__, passes upstream tests

Some refactoring of class handling in the annotator. Remove class specialisation and _settled_ flag.

What’s new in PyPy 4.0.0

Improve the performance of `dict.update()` and a bunch of methods from sets, by reusing the hash value stored in one dict when inspecting or changing another dict with that key.

A major refactoring of the `ResOperations` that kills Box. Also rewrote unrolling to enable future enhancements. Should improve warmup time by 20% or so.

Optimize common sequences of operations like `int_lt/cond_call` in the JIT backends
Fix for missing headers in OpenBSD, already applied in downstream ports

Remove a source of non-incremental-ness in the GC: now external_malloc() no longer runs gc_step_until() any more. If there is a currently-running major collection, we do only so many steps before returning. This number of steps depends on the size of the allocated object. It is controlled by tracking the general progress of these major collection steps and the size of old objects that keep adding up between them.

Reenable jithooks

Issue #2141: fix a crash on Windows and OS/X and ARM when running at least 20 threads.

Add support for ndarray.ctypes property.

Share guard resume data between consecutive guards that have only pure operations and guards in between.

Fix performance regression on operations mixing numpy scalars and Python floats, cf. issue #2148.

Win32: support __stdcall in CFFI.

Refactorings of annotation and rtyping of function calls.

Allow creation of fortran-ordered ndarrays

Remove some remnants of the old ootypesystem vs lltypesystem dichotomy.

ffi.new_handle() returns handles that work more like CPython’s: they remain valid as long as the target exists (unlike the previous version, where handles become invalid before the __del__ is called).

allow automatic casting in ufuncs (and frompypython) to cast the arguments to the allowed function type declarations, fixes various failures in linalg CFFI functions

A new optimization pass to use emit vectorized loops

The PowerPC JIT backend is merged.

Improve the performance of simple trace functions by lazily calling fast2locals and locals2fast only if f_locals is actually accessed.

What’s new in PyPy 2.6.1

Correctly resolve the output dtype of ufunc(array, scalar) calls.

Update stdlib to version 2.7.10

The JIT no longer performs loop unrolling if the loop compiles to too much code.

Build cffi import libraries as part of translation by monkey-patching an additional task into translation

Use a compact strategy for Python lists that mix integers and floats, at least if the integers fit inside 32 bits. These lists are now stored as an array of floats, like lists that contain only floats; the difference is that integers are stored as tagged NaNs. (This should have no visible effect! After lst = [42, 42.5], the value of lst[0] is still not the float 42.0 but the integer 42.)

Part of cffi 1.2.

Part of cffi 1.2.

Partial implementation of unicode dtype and unicode scalars.

Improve compatibility with numpy dtypes; handle offsets to create unions, fix str() and repr(), allow specifying item-size, metadata and titles, add flags, allow subclassing dtype

Refactor array indexing to support ellipses.

Allow the docstrings of built-in numpy objects to be set at run-time.
Implement nditer ‘buffered’ flag and fix some edge cases
Allow multiple axes in ufunc.reduce()
Update tinylang goals to match current rpython
Clean up of vmprof, notably to handle correctly multiple threads
Remove extra link library from Boehm GC

What’s new in PyPy 2.6.0

issue2005: ignore errors on closing random file handles while importing a module (cpython compatibility)
issue2013: added constants to _ssl for TLS 1.1 and 1.2
issue2014: Add PyLong_FromUnicode to cpyext.
issue2017: On non-Linux-x86 platforms, reduced the memory impact of creating a lot of greenlets/tasklets. Particularly useful on Win32 and on ARM, where you used to get a MemoryError after only 2500-5000 greenlets (the 32-bit address space is exhausted).

Update gdb_pypy for python3 (gdb comatability)
Merged rstrategies into rpython which provides a library for Storage Strategies
Support unicode strings in numpy.dtype creation i.e. np.dtype(u’int64’)
Various rpython cleanups for vmprof support
issue2019: Fix isspace as called by rpython unicode.strip()
issue2023: In the cpyext ‘Concrete Object Layer’ API, don’t call methods on the object (which can be overriden), but directly on the concrete base type.
issue2029: Hide the default_factroy attribute in a dict
issue2027: Better document pyinteractive and add –withmod-time
branch gc-incminimark-pinning-improve: Object Pinning is now used in bz2 and rzlib (therefore also affects Python’s zlib). In case the data to compress/decompress is inside the nursery (incminimark) it no longer needs to create a non-moving copy of it. This saves one malloc and copying the data. Additionally a new GC environment variable is introduced (PYPY_GC_MAX_PINNED) primarily for debugging purposes.
branch refactor-pycall: Make *-unpacking in RPython function calls completely equivalent to passing the tuple’s elements as arguments. In other words, f(*(a, b)) now behaves exactly like f(a, b).
branch issue2018: Allow prebuilt rpython dict with function values
branch object-dtype2: Extend numpy dtypes to allow using objects with associated garbage collection hook
branch vmprof2: Add backend support for vmprof - a lightweight statistical profiler - to linux64, see client at https://vmprof.readthedocs.org
branch jit_hint_docs: Add more detail to @jit.elidabile and @jit.promote in rpython/rlib/jit.py
branch remove_frame-debug-atrs: Remove the debug attributes from frames only used for tracing and replace them with a debug object that is created on-demand
branch can_cast: Implement np.can_cast, np.min_scalar_type and missing dtype comparison operations.
branch numpy-fixes: Fix some error related to object dtype, non-contiguous arrays, implement parts of __array_interface__, __array_priority__, __array_wrap__
branch cells-local-stack: Unify the PyFrame.cells and Pyframe.locals_stack_w lists, making frame objects 1 or 3 words smaller.

4.2. Changelogs 299
branch pythonoptimize-env Implement PYTHONOPTIMIZE environment variable, fixing issue #2044
branch numpy-flags Finish implementation of ndarray.flags, including str() and repr()
branch cffi-1.0 PyPy now includes CFFI 1.0.
branch pypyw PyPy on windows provides a non-console pypyw.exe as well as pypy.exe. Similar to pythonw.exe, any use of stdout, stderr without redirection will crash.
branch fold-arith-ops remove multiple adds on add chains (“1 + 1 + 1 + . . . ”)
branch fix-result-types: * Refactor dtype casting and promotion rules for consistency and compatibility with CNumPy.
* Refactor ufunc creation. * Implement np.promote_types().

What’s new in PyPy 2.5.1

Non-blocking file reads sometimes raised EAGAIN even though they had buffered data waiting, fixed in b1c4fcb04a42
Fix a bug in cpyext in multithreaded programs acquiring/releasing the GIL.
Avoid tracing all stack roots during repeated minor collections, by ignoring the part of the stack that didn’t change
Update stdlib to version 2.7.9
Fix exception being raised by kqueue.control (CPython compatibility)
Refactor rpython.flowspace.framestate.FrameState.
Add an alternative location to save LastError, errno around ctypes, cffi external calls so things like pdb will not overwrite it
Speed up the warmup times of the JIT by removing a quadratic algorithm in the heapcache.
Simplify flow graphs on the fly during annotation phase.

What’s new in PyPy 2.5.0

Fix c code generation for msvc so empty “[ ]” are avoided in unions, Avoid re-opening files created with NamedTemporaryFile, Allocate by 4-byte chunks in rffi_platform, Skip testing objdump if it does not exist, and other small adjustments in own tests
Small internal refactorings in the rtyper.
Store annotations on the Variable objects, rather than in a big dict. Introduce a new framework for double-dispatched annotation implementations.
Refactor ClassRepr and make normalizecalls independent of the rtyper.
Remove all remaining multimethods.
Split RPython documentation from PyPy documentation and clean up. There now is a clearer separation between documentation for users, developers and people interested in background information.
Kill multimethod machinery, all multimethods were removed earlier.
Implement external_loop argument to numpy’s nditer
Rename pypy/module/rctime to pypy/module/time, since it contains the implementation of the ‘time’ module.
Use SSA form for flow graphs inside build_flow() and part of simplify_graph()
Implement most of the GenericUfunc api to support numpy linalg. The strategy is to encourage use of pure python or cffi ufuncs by extending frompyfunc(). See the docstring of frompyfunc for more details. This dovetails with a branch
of pypy/numpy - cffi-linalg which is a rewrite of the _umath_linalg module in python, calling lapack from cffi. The branch also support traditional use of cpyext GenericUfunc definitions in c.

This makes ordered dicts the default dictionary implementation in RPython and in PyPy. It polishes the basic idea of ordereddict.py and then fixes various things, up to simplifying collections.OrderedDict.

Note: Python programs can rely on the guaranteed dict order in PyPy now, but for compatibility with other Python implementations they should still use collections.OrderedDict where that really matters. Also, support for reversed() was not added to the ‘dict’ class; use OrderedDict.

Benchmark results: in the noise. A few benchmarks see good speed improvements but the average is very close to parity.

This branch adds an interface rgc.pin which would (very temporarily) make object non-movable. That’s used by rffi.alloc_buffer and rffi.get_nonmovable_buffer and improves performance considerably for IO operations.

A branch started by Wenzhu Man (SoC’14) and then done by fijal. It removes the clearing of the nursery. The drawback is that new objects are not automatically filled with zeros any longer, which needs some care, mostly for GC references (which the GC tries to follow, so they must not contain garbage). The benefit is a quite large speed-up.

Fix intern() to return mortal strings, like in CPython.

--shared support on OS/X (thanks wouter)

Changes how errno, GetLastError, and WSAGetLastError are handled. The idea is to tie reading the error status as close as possible to the external function call. This fixes some bugs, both of the very rare kind (e.g. errno on Linux might in theory be overwriten by mmap(), called rarely during major GCs, if such a major GC occurs at exactly the wrong time), and some of the less rare kind (particularly on Windows tests).

Improve performance of integer globals and class attributes.

What’s new in PyPy 2.4+

Bytearray operations no longer copy the bytearray unnecessarily

Added support for __getitem__, __setitem__, __getslice__, __setslice__, and __len__ to RPython

Give the StringBuilder a more flexible internal structure, with a chained list of strings instead of just one string. This make it more efficient when building large strings, e.g. with cStringIO().

Also, use systematically jit.conditional_call() instead of regular branches. This lets the JIT make more linear code, at the cost of forcing a bit more data (to be passed as arguments to conditional_calls). I would expect the net result to be a slight slow-down on some simple benchmarks and a speed-up on bigger programs.

Change the executioncontext’s lookup to be done by reading a thread- local variable (which is implemented in C using ‘_thread’ if possible, and pthread_getspecific() otherwise). On Linux x86 and x86-64, the JIT backend has a special optimization that lets it emit directly a single MOV from a %gs- or %fs-based address. It seems actually to give a good boost in performance.

A faster way to handle the GIL, particularly in JIT code. The GIL is now a composite of two concepts: a global number (it’s just set from 1 to 0 and back around CALL_RELEASE_GIL), and a real mutex. If there are threads waiting to acquire the GIL, one of them is actively checking the global number every 0.1 ms to 1 ms. Overall, JIT loops full of external function calls now run a bit faster (if no thread was started yet), or a lot faster (if threads were started already).

Optimize the errno handling in the JIT, notably around external function calls. Linux-only.

Remove non-functioning ctypes.pythonapi and ctypes.PyDLL, document this incompatibility with cpython. Recast sys.dlhandle to an int.

Fix performance regression on ufunc(<scalar>, <scalar>) in numpy.
PyPy, Release 7.2.0

Update our copies of py.test and pylib to versions 2.5.2 and 1.4.20, respectively.
Classes in the ast module are now distinct from structures used by the compiler.
Upgrades from 2.7.6 to 2.7.8
Fix issue #1861 - cpython compatibility madness

What's new in PyPy 2.3

Move built-in struct module to _struct to allow pypy "-m idlelib.idle"
Support compilation with gcc-4.9
Added support for the stdlib gdbm module via cffi
Annotator cleanups
Use argparse for packaging.py, and add third-party components to LICENSE file. Also mention that gdbm is GPL. Do not crash the packaging process on failure in CFFI or license-building, rather complete the build step and return -1.

What's new in PyPy 2.2.1

Clean up numpy types, add newbyteorder functionality
Package tk/tcl runtime with win32
JIT support for singlefloats on ARM using the hardfloat ABI
Better support for record numpy arrays
OSX: Ensure frameworks end up in Makefile when specified in External compilation info
Use subclasses of SpaceOperation instead of SpaceOperator objects. Random cleanups in flowspace and annotator.
adds support for the buffer= argument to the ndarray ctor
On OpenBSD do not pull in libcompat.a as it is about to be removed. And more generally, if you have gettimeofday(2) you will not need ftime(3).
Remove dependency upon <sys/timeb.h> on OpenBSD. This will be disappearing along with libcompat.a.
Fix 3 broken links on PyPy published papers in docs.
Remove multimethods on str/unicode/bytearray and make the implementations share code.
Speed up generators that don’t yield inside try or wait blocks by skipping unnecessary cleanup.
Remove FlowObjSpace. Improve cohesion between rpython.flowspace and rpython.annotator.
mapdicts keep track of whether or not an attribute is every assigned to multiple times. If it’s only assigned once then an elidable lookup is used when possible.
Create a Makefile using precompiled headers for MSVC platforms. The downside is a messy nmake-compatible Makefile. Since gcc shows minimal speedup, it was not implemented.
With a properly configured 256-color terminal (TERM=...-256color), the Mandelbrot set shown during translation now uses a range of 50 colours. Essential!
Simplify implementation of NonConstant.
Kill some guards and operations in JIT traces by adding integer bounds propagation for getfield_(rawlge) and getarrayitem_(rawlge).
Optimize away INT_AND with constant mask of 1s that fully cover the bitrange of other operand.
Propagate appropriate bounds through INT_(OR|XOR|AND) operations if the operands are positive to kill some guards

cleans int/long/smalllong/bool multimethods

Cleanup micronumpy module

In a lot of places CPython allows objects with __int__ and __float__ instead of actual ints and floats, while until now pypy disallowed them. We fix it by making space.{int_w,float_w,etc.} accepting those objects by default, and disallowing conversions only when explicitly needed.

Fix for getarrayitem.gc_pure optimization

Implements SimpleRangeListStrategy for case range(n) where n is a positive number. Makes some traces nicer by getting rid of multiplication for calculating loop counter and propagates that n > 0 further to get rid of guards.

Provide an exit status for popen’ed RFiles via pclose

Update stdlib to v2.7.6

Support for virtualizing raw_store/raw_load operations

Separate the interp-level buffer API from the buffer type exposed to app-level. The Buffer class is now used by W_MemoryView and W_Buffer, which is not present in Python 3. Previously W_Buffer was an alias to Buffer, which was wrappable itself.

Improve the situation when dict lookups of the same key are performed in a chain

Add support for PyErr_SetFromErrnoWithFilenameObject to cpyext

fix more tests for win32

Fix broken links in documentation

fix ast classes __dict__ are always empty problem and fix the ast deepcopy issue when there is missing field

Fix issues with reimporting builtin modules

Implement the core of nditer, without many of the fancy flags (external_loop, buffered)

Separate iterator from its state so jit can optimize better

Implement searchsorted without sorter kwarg

add ‘lib’ prefix to link libraries on OpenBSD

Improve optimization of small allocation-heavy loops in the JIT

Properly implement old/new buffer API for objects and start work on replacing bufferstr usage

Add a lock for unsafe calls to gethostbyname and gethostbyaddr

Changes hacks surrounding W_TypeObject.name to match CPython’s tp_name

OS/X specific header path

What’s new in PyPy 2.2

Make PyPy’s bytecode dispatcher easy to read, and less reliant on RPython magic. There is no functional change, though the removal of dead code leads to many fewer tests to execute.

Fast json decoder written in RPython, about 3-4x faster than the pure Python decoder which comes with the stdlib

Improve the performance of I/O writing up to 15% by using memcpy instead of copying char-by-char in str2charp and get_nonmovingbuffer

Simplify rpython/flowspace/ code by using more metaprogramming. Create SpaceOperator class to gather static information about flow graph operations.
Adapt package.py script to compile CFFI tk extension. Add a –without-tk switch to optionally skip it.

Copy CPython’s implementation of customize_compiler, dont call split on environment variables, honour CFLAGS, CPPFLAGS, LDSHARE and LDFLAGS on Unices.

When an RPython class is instantiated via an indirect call (that is, which class is being instantiated isn’t known precisely) allow the optimizer to have more precise information about which functions can be called. Needed for Topaz.

Make PyPy respect PYTHONINSPECT variable set via os.putenv in the same process to start interactive prompt when the script execution finishes. This adds new __pypy__._os.real_getenv call that bypasses Python cache and looks up env in the underlying OS. Translatorshell now works on PyPy.

Added os.statvfs and os.fstatvfs

Added some addition tests for statvfs.

Allow subclassing ndarray, i.e. matrix

Implement ndarray in-place sorting (for numeric types, no non-native byte order)

Implement much of numpy’s c api in cpyext, allows (slow) access to ndarray from c

Added an abstraction for functions with a fast and slow path in the JIT. This speeds up list.append() and list.pop().

Constant-fold reading out of constant tuples in PyPy.

No longer delegate numpy string methods to space.StringObject, in numpy this works by kind of by accident. Support for merging the refactor-str-types branch

Remove the “type system” abstraction, now that there is only ever one kind of type system used.

Kills gen_store_back_in_virtualizable - should improve non-inlined calls by a bit

Work on improving UnionError messages and stack trace displays.

More improvements and refactorings of error messages.

Unbreak tests in rlib.

Use subclasses of SpaceOperation instead of SpaceOperator objects. Random cleanups in flowspace.

make open() and friends rpython

Added the new incminimark GC which performs GC in incremental steps

fastpath for cffi.new("long[]")

remove a pointless abstraction

Allow the jit to continue running when sys.settrace() is active, necessary to make coverage.py fast

Remove lib_pypy/numpypy in favor of external numpy fork

Tweak the jit counters: decay them at minor collection (actually only every 32 minor collection is enough). Should avoid the “memory leaks” observed in long-running processes, actually created by the jit compiling more and more rarely executed paths.

Fixed the usage of sys.settrace() with the JIT. Also made it so using sys.settrace() doesn’t cause the GIL to be released on every single iteration.

Implement OrderedDict in RPython
What’s new in PyPy 2.1

* put and array.put
* Pickling of numpy arrays and dtypes (including record dtypes)
* Remove multimethods in the arraymodule
* Fixed bug when switching stacklets from a C callback
* Remove multi-methods on sets
* Implement subarrays for numpy
* Remove multi-methods on dict
* Remove remaining multi-methods on list
* Stacklet support for ARM, enables _continuation support
* Remove multi-methods on tuple
* Remove multi-methods on iterators
* Added list of resops to the pypyjit on_abort hook.
* Speeds up the stdlib logging module
* Adds a couple convenient format specifiers to operationerrfmt
* Skip and fix some non-translated (own) tests for win32 builds
* Add the ‘*_obj’ attribute on ctypes pointer() and byref() objects
* Fix a segfault in argsort when sorting by chunks on multidim numppypy arrays (mikefc)
* Improve performance of str(long).
* Add view to ndarray and zeroD arrays, not on dtype scalars yet
* fix segfault caused by iterating over empty ndarrays
* Faster sets for objects
* Inline the fast path of id() and hash()
* Adapt package.py script to compile CFFI tk extension. Add a –without-tk switch to optionally skip it.

What’s new in PyPy 2.0

* Split rpython and pypy into separate directories
* Callbacks from C are now better JITted
* Implement __length_hint__ according to PEP 424
* Long double support for numppypy
* Since r_longdouble support is missing, disable all longdouble and derivative dtypes using ENABLED_LONG_DOUBLE = False
* Convert real, imag from ufuncs to views. This involves the beginning of view() functionality
* Adds indexing by scalar, adds int conversion from scalar and single element array, fixes compress, indexing by an array with a smaller shape and the indexed object.
* Allow concatenation of str and numeric arrays
Improved RPython typing

Rudimentary support for bytearray in RPython

Fix a bug which caused cffi to return the wrong result when calling a C function which calls a Python callback which forces the frames

JIT optimizations which make cffi calls even faster, by removing the need to allocate a temporary buffer where to store the arguments.

Improve documents and straighten out links

Inline the fast path of newarray in the assembler. Disabled on ARM until we fix issues.

Allow dynamic loading of a (Reflex) backend that implements the C-API needed to provide reflection information

Fixed the interaction between two internal tools for controlling the JIT.

Better optimized certain types of frame accesses in the JIT, particularly around exceptions that escape the function they were raised in.

Some missing attributes from ndarrays

Consolidated the lib_pypy/pypy_test and pypy/module/test_lib_pypy tests into one directory for reduced confusion and so they all run nightly.

Add “__pypy__thread.signals_enabled”, a context manager. Can be used in a non-main thread to enable the processing of signal handlers in that thread.

Support enumerate() over rstr types.

Cleanup _numpypy and numpypy namespaces to more closely resemble numpy.

Random cleanups to hide FlowObjSpace from public view.

Moves optimized JIT frames from stack to heap. As a side effect it enables stackless to work well with the JIT on PyPy. Also removes a bunch of code from the GC which fixes cannot find gc roots.

Documentation fixes after going through the docs at PyCon 2013 sprint.

cffi implementation of _curses

cffi implementation of sqlite3

**What’s new in PyPy xxx**

Fixed the performance of gc.get_referrers()

Provides cppyy module (disabled by default) for access to C++ through Reflex. See doc/cppyy.rst for full details and functionality.

Check that axis arg is valid in _numpypy

Remove numpy lazy evaluation and simplify everything

Support for array[array-of-ints] in numpypy

Implement better JIT hooks

Improve handling of **kwds greatly, making them virtual sometimes.

Introduce __int128 on systems where it’s supported and improve the speed of rlib/rbigint.py greatly.

Start to clean up a bit the flow object space.

Support CFFI. [http://morepypy.blogspot.ch/2012/08/cffi-release-03.html](http://morepypy.blogspot.ch/2012/08/cffi-release-03.html)
The stdlib was updated to version 2.7.3
Complex dtype support for numpy
Improve dtypes intp, uintp, void, string and record
Add float16 numpy dtype
major cleanups including killing some object support
implement threadstate-related functions in cpyext
add dict/list/set strategies optimized for unicode items
Support for utf-8 encoding in RPython
Support ARM in the JIT.

What’s new in PyPy 1.9

Working hash function for numpy types.
Improved jit hooks
Added some primitives for dealing with floats as raw bytes.
Added more float byte primitives.
Put more debug info into resops.
Kill “geninterp”, an old attempt to statically turn some fixed app-level code to interp-level.
Finished select.kqueue.
Special dictionary strategy for dealing with **kwds. Now having a simple proxy def f(*args, **kwds):
return x(*args, **kwds) should not make any allocations at all.
numpypy can now handle matrix multiplication.
The stdlib was updated to version 2.7.2
cpyext: Better support for PyEval_SaveThread and other PyThreadState_* functions.
flatitier for numpy
reuse more of original numpy
concatenation support for numpy
indexing by bool arrays
record dtypes on numpy has been started
various refactorings regarding numpy
The “out” argument was added to most of the numppypy functions.
CPyext improvements. For example PyOpenSSL should now work
Sets now have strategies just like dictionaries. This means a set containing only ints will be more compact (and faster)
The simplest case of list comprehension is preallocating the correct size of the list. This speeds up select benchmarks
quite significantly.
The directory “lib-python/modified-2.7” has been removed, and its content merged into “lib-python/2.7”.
The common case of a xrange iterator with no step argument specified was somewhat optimized. The tightest loop
involving it, sum(xrange(n)), is now 18% faster on average.
PyPy, Release 7.2.0

PyPy refuses filenames with chr(0) characters. This is implemented in RPython which can enforce no-NUL correctness and propagation, similar to const-correctness in C++.

Many bugs were corrected for windows 32 bit. New functionality was added to test validity of file descriptors, leading to the removal of the global `_invalid_parameter_handler`

Add os.kill to windows even if translating python does not have os.kill

Handle calling conventions for the `_ffi` and ctypes modules

Memory “leaks” associated with zlib are fixed.

The `ffistruct` branch adds a very low level way to express C structures with `_ffi` in a very JIT-friendly way (preliminary work for sepcomp)

4.2.2 CPython 3.5 compatible versions

What’s new in PyPy 3.0+

What’s new in PyPy 3.0+

Fix multiprocessing regression on newer glibcs

Use implementation-specific site directories in sysconfig like in Python2

The reverse-debugger branch has been merged. For more information, see https://bitbucket.org/pypy/revdb

What’s new in PyPy 3.9

What’s new in PyPy 3.7+

Use “<python> -m test” to run the CPython test suite, as documented by CPython, instead of our outdated regrverbose.py script.

Enable the ‘faulthandler’ module on Windows; this unblocks the Python test suite.

Implement posix.posix_fallocate() and posix.posix_fadvise()

Fix for different posix primitives on MacOS

Internal refactoring of memoryviews and buffers, fixing some related performance issues.

Add sched_get min/max to rposix

What’s new in PyPy 3.7.0

Implement some level of compatibility with PEP 393 APIs.

4.2.3 CPython 3.3 compatible versions

What’s new in PyPy 3.5.0

Update fallback code in time to match CPython.

Add str.casefold().

Update Unicode character database to version 6.1.0.
Make win_perf_counter expose the clock info. Add a couple more fallbacks. Make time.monotonic conditionally available depending on platform.

Make hash(-1) return -2, like it does on CPython.

Fix the mappingproxy type to behave as in CPython.

Implement keyword-only arguments for built-in functions. Fix functions in the posix module to have keyword-only arguments wherever CPython has them, instead of regular keyword arguments.

Add os.get_terminal_size().

Implement slicing of memoryview objects and improve their compatibility with CPython.

Set up ImportError attributes properly in _imp.load_dynamic().

Allow __len__ to return any index-like.

Replace stub faulthandler module with a working implementation.

What's new in PyPy3 5.1.1 alpha 1

Python 3.3 compatibility

4.2.4  CPython 3.2 compatible versions

What's new in PyPy3 2.4.0

What's new in PyPy3 2.3.1

#encoding utf-8

4.3  Contributors

Armin Rigo
Maciej Fijalkowski
Carl Friedrich Bolz-Tereick
Antonio Cuni
Amaury Forgeot d'Arc
Matti Picus
Samuele Pedroni
Ronan Lamy
Alex Gaynor
Philip Jenvey
Richard Plangger
Brian Kearns
Michael Hudson-Doyle
Manuel Jacob
David Schneider
Holger Krekel
Christian Tismer
Hakan Ardo
Benjamin Peterson
Anders Chrigstrom
Wim Lavrijsen

(continues on next page)
Eric van Riet Paap
Remi Meier
Richard Emslie
Alexander Schremmer
Dan Villiomi Podlaski Christiansen
Lukas Diekmann
Sven Hager
Anders Lehmann
Aurelien Campeas
Niklaus Haldimann
Camillo Bruni
Laura Creighton
Toon Verwaest
Leonardo Santagada
Seo Sanghyeon
Romain Guillebert
Ronny Pfannschmidt
Justin Peel
Raffael Tfirst
David Edelsohn
Anders Hammarquist
Jakub Gustak
Gregor Wegberg
Guido Wesdorp
Lawrence Oluyede
Bartosz Skowron
Daniel Roberts
Adrien Di Mascio
Niko Matsakis
Alexander Hesse
Ludovic Aubry
stian
Jacob Hallen
Jason Creighton
Mark Young
Alex Martelli
Spenser Bauman
Michal Bendowski
Jan de Mooij
Tyler Wade
Vincent Legoll
Michael Foord
Stephan Diehl
Stefano Rivera
Jean-Paul Calderone
Stefan Schwarzer
Tomek Meka
Valentino Volonghi
Patrick Maupin
Devin Jeanpierre
Bob Ippolito
Bruno Gola
Andrew Lawrence
David Malcolm
Squeaky
Edd Barrett
Timo Paulissen
Laurence Tratt
Marius Gedminas
Nicolas Truesnel
Alexandre Fayolle
Simon Burton
Martin Matusiak
Wenzhu Man
Konstantin Lopuhin
John Witulski
Stefan Beyer
Jeremy Thurgood
Greg Price
Ivan Sichmann Freitas
Dario Bertini
Mark Pearse
Simon Cross
Tobias Pape
Andreas Stührk
Jean-Philippe St. Pierre
Guido van Rossum
Pavel Vinogradov
William Leslie
Paweł Piotr Przeradowski
marky1991
Ilya Osadchyi
Tobias Oberstein
Paul deGrandis
Boris Feigin
Taavi Burns
Adrian Kuhn
tav
Stian Andreassen
Georg Brandl
Joannah Nanjekye
Bert Freudenberg
Wanja Saatkamp
Mike Blume
Gerald Klix
Julian Berman
Oscar Nierstrasz
Rami Chowdhury
Stefan H. Muller
Dodan Mihai
Tim Felgentreff
Eugene Oden
Colin Valliant
Jeff Terrace
Henry Mason
Vasily Kuznetsov
Preston Timmons
David Ripton
Pieter Zieschang
Dusty Phillips
Lukas Renggli
Guenter Jantzen
Jasper Schultz
Ned Batchelder

4.3. Contributors
Amit Regmi
Anton Gulenko
Sergey Matyunin
Andrew Chambers
Łukasz Langa
Nicolas Chauvat
Andrew Durdin
Ben Young
Michael Schneider
Yusuke Tsutsumi
Nicholas Riley
Jason Chu
Igor Trindade Oliveira
Yichao Yu
Michael Twomey
Rocco Moretti
Gintautas Miliauskas
Lucian Branescu Mihaila
Mariano Anaya
anatoly techtonik
Karl Bartel
Gabriel Lavoie
Jared Grubb
Alec Sandru Patrascu
Olivier Dormond
Wouter van Heyst
Sebastian Pawluś
Brian Dorsey
Victor Stinner
Andrews Medina
Aaron Iles
Toby Watson
Daniel Patrick
Stuart Williams
Antoine Pitrou
Christian Hudon
Justas Sadzevicius
Neil Shepperd
Michael Cheng
Mikael Schönenberg
Stanislaw Halik
Mihnea Saracín
Matt Jackson
Berkin Ilbeyi
Gasper Zejn
Faye Zhao
Elmo Müntynen
Anders Qvist
Corbin Simpson
Chirag Jadwani
Pauli Virtanen
Jonathan David Riehl
Beatrice During
Alex Perry
Robert Zaremba
Alan McIntyre
Alexander Sedov
(continued from previous page)

David C Ellis
Vaibhav Sood
Reuben Cummings
Attila Gobi
Floris Bruynooghe
Christopher Pope
Tristan Arthur
Christian Tismer
Dan Stromberg
Carl Meyer
Florin Papa
Arianna Avanzini
Jens-Uwe Mager
Valentina Mukhamedzhanova
Stefano Parmesan
touilleMan
Marc Abramowitz
Arjun Naik
Aaron Gallagher
Alexis Daboville
Karl Ramm
Lukas Vacek
Omer Katz
Jacek Generowicz
Tomasz Dziopa
Lin Cheng
Sylvain Thenault
Jakub Stasiak
Andrew Dalke
Alejandro J. Cura
Vladimir Kryachko
Gabriel
Thomas Hisch
Mark Williams
Kunal Grover
Nathan Taylor
Barry Hart
Travis Francis Athougies
Yasir Suhail
Sergey Kishchenko
Martin Blais
Lutz Paelike
Ian Foote
Philipp Rustemeyer
Logan Chien
Catalin Gabriel Manciu
Jacob Oscarson
Ryan Gonzalez
Antoine Dupre
Kristjan Valur Jonsson
Lucio Torre
Richard Lancaster
Dan Buch
Lene Wagner
Tomo Cocoa
Miro Hrončok
Anthony Sottile

(continues on next page)
David Lievens
Neil Blakey-Milner
Henrik Vendelbo
Lars Wassermann
Ignas Mikalajunas
Christoph Gerum
Miguel de Val Borro
Artur Lisiecki
afteryu
Toni Mattis
Laurens Van Houtven
Bobby Impollonia
Roberto De Ioris
Jeong YunWon
Christopher Armstrong
Aaron Tubbs
Vasantha Ganesh K
Jason Michalski
Radu Ciorba
Markus Holtermann
Andrew Thompson
Yusei Tahara
Ruochen Huang
Fabio Niephaus
Akira Li
Gustavo Niemeyer
Nate Bragg
Lucas Stadler
roberto@goyle
Carl Bordum Hansen
Matt Bogosian
Yury V. Zaytsev
florinpapa
Anders Sigfridsson
Nikolay Zinov
rafalgalcznynski@gmail.com
Joshua Gilbert
Anna Katrina Dominguez
Kim Jin Su
Amber Brown
Andrew Stepanov
Rafał Gałczyński
Ben Darnell
Juan Francisco Cantero Hurtado
Godefroid Chappelle
Stephan Busemann
Dan Colish
timo
Volodymyr Vladomyrov
Daniel Neuhausser
Flavio Percoco
halgari
Jim Baker
Chris Lambacher
John Aldis
coolbutuseless@gmail.com
Mike Bayer
Rodrigo Araújo
Daniil Yarancev
Min RK
Olivier Blanvillain
Jonas Pfannschmidt
Zearin
Johan Forsberg
Andrey Churin
Dan Crosta
reubano@gmail.com
Stanisław Halik
Julien Phalip
Roman Podoliaka
Steve Papanik
Eli Stevens
Boglarka Vezer
gabrielg@ec2-54-146-239-158.compute-1.amazonaws.com
Pavlo Kapyshin
Hervé Beraud
Tomer Chachamu
Christopher Groskopf
Asmo Soinio
Antony Lee
Jim Hunziker
shoma hosaka
Buck Golemon
Iraklis D.
JohnDoe
yrttyr
Michael Chermside
Anna Ravencroft
remarkablerocket
Petre Vijiac
Berker Peksag
Christian Muirhead
soareschen
Matthew Miller
Konrad Delong
Dinu Gherman
pizi
Tomáš Pružina
James Robert
Armin Ronacher
Diana Popa
Mads Kiilerich
Brett Cannon
Caleb Hattingh
aliceinwire
Zooko Wilcox-O Hearn
James Lan
jiaaro
Evgenii Gorinov
Markus Unterwaditzer
Kristoffer Kleine
Graham Markall
Dan Loewenherz
werat

(continues on next page)
4.4 Glossary

application level  applevel code is normal Python code running on top of the PyPy or CPython interpreter (see interpreter level).

CPython  The “default” implementation of Python, written in C and distributed by the PSF on http://www.python.org.

interpreter level  Code running at this level is part of the implementation of the PyPy interpreter and cannot interact normally with application level code; it typically provides implementation for an object space and its builtins.

mixed module  a module that accesses PyPy’s interpreter level. The name comes from the fact that the module’s implementation can be a mixture of application level and interpreter level code.

object space  The Object Space (often abbreviated to “objspace”) creates all objects and knows how to perform operations on the objects. You may think of an object space as being a library offering a fixed API, a set of operations, with implementations that a) correspond to the known semantics of Python objects, b) extend or twist these semantics, or c) serve whole-program analysis purposes.

stackless  Technology that enables various forms of non conventional control flow, such as coroutines, greenlets and tasklets. Inspired by Christian Tismer’s Stackless Python.

standard interpreter  It is the subsystem implementing the Python language, composed of the bytecode interpreter and of the standard objectspace.

ebuilt constant  In RPython module globals are considered constants. Moreover, global (i.e. prebuilt) lists and dictionaries are supposed to be immutable (“prebuilt constant” is sometimes abbreviated to “pbc”).
CHAPTER 5

Contact

#pypy on irc.freenode.net  Many of the core developers are hanging out here. You are welcome to join and ask questions (if they are not already answered in the FAQ). You can find logs of the channel here.

Development mailing list  Development and conceptual discussions

Commit mailing list  Updates to code and documentation

Development bug/feature tracker  Filing bugs and feature requests

Meeting PyPy developers  The PyPy developers are organizing sprints and presenting results at conferences all year round. They will be happy to meet in person with anyone interested in the project. Watch out for sprint announcements on the development mailing list.
CHAPTER 6

Indices and tables

- genindex
- modindex
- search
A
application level, 316
args, 117

B
bigint_w() (built-in function), 122
bytes_w() (pypy.interpreter.baseobjspace.ObjSpace
method), 122

call() (built-in function), 119
call_function() (built-in function), 120
call_method() (built-in function), 120
callable() (built-in function), 120
CPython, 316

delegate(), 117
disable_debug() (built-in function), 36
enable_debug() (built-in function), 35
eq_w() (built-in function), 120

F
finditem() (built-in function), 120
float_w() (built-in function), 122

G
get_stats_snapshot() (built-in function), 36
get_tproxy_controller() (built-in function), 116
getbuiltinmodule() (built-in function), 119
getexecutioncontext() (built-in function), 119
getindex_w() (built-in function), 122

H
hash_w() (built-in function), 120

I
index() (built-in function), 119
int_w() (built-in function), 121
interp_w() (built-in function), 122
interpclass_w() (built-in function), 122
interpreter level, 316
is__() (built-in function), 119
is_true() (built-in function), 121
is_w() (built-in function), 120
isinstance() (built-in function), 120

J
JitInfoSnapshot (built-in class), 36
JitLoopInfo (built-in class), 36

K
kwargs, 117

L
len_w() (built-in function), 120

M
make_proxy() (built-in function), 117
mixed module, 316

N
newbool() (built-in function), 121
newbytes() (built-in function), 121
newdict() (built-in function), 121
newint() (built-in function), 121
newlist() (built-in function), 121
newlong() (built-in function), 121
newslice() (built-in function), 121
newtext() (built-in function), 121
newtuple() (built-in function), 121
newunicode() (built-in function), 121
newutf8() (built-in function), 121
not__() (built-in function), 120
O

obj, 117
object space, 316
ObjSpace.BuiltinModuleTable (built-in variable), 123
ObjSpace.ConstantTable (built-in variable), 123
ObjSpace.ExceptionTable (built-in variable), 123
ObjSpace.IrregularOpTable (built-in variable), 123
ObjSpace.MethodTable (built-in variable), 123
opname, 117

P

proxyobj, 117
Python Enhancement Proposals
  PEP 0393, 69
  PEP 8, 73

S

set_abort_hook (built-in function), 35
set_compile_hook (built-in function), 35
space.builtin (built-in variable), 122
space.sys (built-in variable), 122
space.w_Ellipsis (built-in variable), 123
space.w_False (built-in variable), 122
space.w_float (built-in variable), 123
space.w_instance (built-in variable), 123
space.w_int (built-in variable), 123
space.w_long (built-in variable), 123
space.w_None (built-in variable), 122
space.w_NotImplemented (built-in variable), 123
space.w_slice (built-in variable), 123
space.w_str (built-in variable), 123
space.w_True (built-in variable), 122
space.w_tuple (built-in variable), 123
space.w_type (built-in variable), 123
space.w_unicode (built-in variable), 123
stackless, 316
standard interpreter, 316
str_w (built-in function), 122

T

text_w (pypy.interpreter.baseobjspace.ObjSpace
  method), 122
tproxy (built-in function), 116

U

unicode_w (built-in function), 122
unpackiterable (built-in function), 120
unpacktuple (built-in function), 120
unwrap (built-in function), 121