

---

# **fatoptimizer Documentation**

*Release 0.2*

**Victor Stinner**

January 23, 2016



**1 Table Of Contents**

**3**



`fatoptimizer` is a static optimizer for Python 3.6 using function specialization with guards. It is implemented as an AST optimizer.

Optimized code requires the *fat module* at runtime if at least one function was specialized.

Links:

- [fatoptimizer documentation](#) (this documentation)
- [fatoptimizer project at GitHub](#) (code, bug tracker)
- [fatoptimizer project at the Python Cheeseshop \(PyPI\)](#) (download releases)
- [FAT Python](#)

`fatoptimizer` requires a Python 3.6 patched with the PEP 511 (`ast.Constant`, `sys.ast_transformers`).



---

## Table Of Contents

---

### 1.1 fatoptimizer module

#### 1.1.1 fatoptimizer API

**Warning:** The API is not stable yet.

**optimize** (*tree*, *filename*, *config*)

Optimize an AST tree. Return the optimized AST tree.

**pretty\_dump** (*node*, *annotate\_fields=True*, *include\_attributes=False*, *lineno=False*, *indent=' '*)

Return a formatted dump of the tree in *node*. This is mainly useful for debugging purposes. The returned string will show the names and the values for fields. This makes the code impossible to evaluate, so if evaluation is wanted *annotate\_fields* must be set to False. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include\_attributes* can be set to True.

**class Config**

Configuration of the optimizer.

See *fatoptimizer configuration*.

**class FATOptimizer** (*config*)

Code transformers for `sys.set_code_transformers()`.

**class OptimizerError**

Exception raised on bugs in the optimizer.

#### 1.1.2 Installation

fatoptimizer requires Python 3.6 patched with the PEP 511 (`ast.Constant`, `sys.ast_transformers`).

Type:

```
pip install fatoptimizer
```

Manual installation:

```
python3.6 setup.py install
```

### 1.1.3 Configuration

It is possible to configure the AST optimizer per module by setting the `__fattoptimizer__` variable. Configuration keys:

- `enabled` (bool): set to `False` to disable all optimization (default: `true`)
- `constant_propagation` (bool): enable *constant propagation* optimization? (default: `true`)
- `constant_folding` (bool): enable *constant folding* optimization? (default: `true`)
- `copy_builtin_to_constant` (bool): enable *copy builtin functions to constants* optimization? (default: `false`)
- `remove_dead_code` (bool): enable *dead code elimination* optimization? (default: `true`)
- maximum size of constants:
  - `max_bytes_len`: Maximum number of bytes of a text string (default: 128)
  - `max_int_bits`: Maximum number of bits of an integer (default: 256)
  - `max_str_len`: Maximum number of characters of a text string (default: 128)
  - `max_seq_len`: Maximum length in number of items of a sequence like tuples (default: 32). It is only a preliminary check: `max_constant_size` still applies for sequences.
  - `max_constant_size`: Maximum size in bytes of other constants (default: 128 bytes), the size is computed with `len(marshal.dumps(obj))`
- `replace_builtin_constant` (bool): enable *replace builtin constants* optimization? (default: `true`)
- `simplify_iterable` (bool): enable *simplify iterable optimization*? (default: `true`)
- `unroll_loops`: Maximum number of loop iteration for loop unrolling (default: 16). Set it to 0 to disable loop unrolling. See *loop unrolling* optimization.

Example to disable all optimizations in a module:

```
__fattoptimizer__ = {'enabled': False}
```

Example to disable the constant folding optimization:

```
__fattoptimizer__ = {'constant_folding': False}
```

See the *Config* class.

### 1.1.4 Run tests

Type:

```
tox
```

You may need to install or update tox:

```
pip3 install -U tox
```

Run manually tests:

```
python3 test_fattoptimizer.py
```

There are also integration tests which requires a Python 3.6 with patches PEP 509, PEP 510 and PEP 511. Run integration tests:



```
python3.6 -X fat test_fat_config.py
python3.6 -X fat test_fat_size.py
```

## 1.2 fat module

The `fat` module is Python extension module (written in C) implementing fast guards. The *fatoptimizer optimizer* uses `fat` guards to specialize functions and only the specialize bytecode under some conditions.

The `fat` module is required to run code optimized by `fatoptimizer` if at least one function is specialized.

- [fat project at GitHub](#)
- [fat project at the Python Cheeseshop \(PyPI\)](#)

The `fat` module requires a Python 3.6 patched with PEP 510 patch.

### 1.2.1 fat module API

**Warning:** The API is not stable yet.

`fat.__version__` is the version string.

#### Functions

**replace\_consts** (*code*, *mapping*)

Create a copy of the code object with replaced constants:

```
new_consts = tuple(mapping.get(const, const) for const in consts)
```

**specialize** (*func*, *code*, *guards*)

Specialize a Python function: add a specialized code with guards.

*code* must be a callable or code object, *guards* must be a non-empty sequence of guards.

**get\_specialized** (*func*)

Get the list of specialized codes with guards as (*code*, *guards*) tuples.

See the PEP 510 for the API of `specialize()` and `get_specialized()`.

#### Guard types

**class GuardArgType** (*arg\_index*, *arg\_types*)

Check the type of the *n*th argument, *arg\_types* must be a sequence of types.

Attributes:

**arg\_index**

Index of the argument (`int`). Read-only attribute.

**arg\_types**

List of accepted types for the argument: list of types. Read-only property.

Keep a strong reference to *arg\_types* types.

**class** **GuardBuiltins** (*names*)

Subtype of *GuardDict*.

Watch for:

- globals of the current frame (`frame.f_globals`)
- globals() [*name*] for all *names*.
- builtins of the current frame (`frame.f_builtins`)
- builtins.\_\_dict\_\_[*name*] for all *names*

The guard initialization fails if `builtins.__dict__[name]` was replaced after `fat` was imported, or if `globals() [name]` already exists.

Attributes:

**guard\_globals**

The *GuardGlobals* used to watch for the global variables. Read-only attribute.

Keep a strong references to the builtin namespace (`builtins.__dict__` dictionary), to the global namespace (`globals()` dictionary), to *names* and to existing builtin symbols called *names* (`builtins.__dict__[name]` for all *names*).

**class** **GuardDict** (*dict*, *keys*)

Watch for `dict[key]` for all *keys*.

Attributes:

**dict**

Watched dictionary (*dict*). Read-only attribute.

**keys**

List of watched dictionary keys: list of `str`. Read-only property.

Keep a strong references to *dict*, to *keys* and to existing dictionary values (`dict[key]` for all *keys*).

**class** **GuardFunc** (*func*)

Watch for the code object (`func.__code__`) of a Python function.

Attributes:

**code**

Watched code object. Read-only attribute.

**func**

Watched function. Read-only attribute.

Keep a strong references to *func* and to `func.__code__`.

**class** **GuardGlobals** (*names*)

Subtype of *GuardDict*.

Watch for:

- globals of the current frame (`frame.f_globals`)
- globals() [*name*] for all *names*.

Keep a strong references to the global namespace (`globals()` dictionary), to *names* and to existing global variables called *names* (`globals() [name]` for all *names*).

## Guard helper functions

### `guard_type_dict` (*type*, *attrs*)

Create `GuardDict` (`type.__dict__`, `attrs`) but access the real type dictionary, not `type.__dict` which is a read-only proxy.

Watch for `type.attr` (`type.__dict__[attr]`) for all *attrs*.

## 1.2.2 Installation

A Python 3.6 patched with PEP 510 patch is required.

Type:

```
pip install fat
```

Manual installation:

```
python3.6 setup.py install
```

## 1.2.3 Run tests

Type:

```
./runtests.sh
```

## 1.2.4 Changelog

- 2016-01-22: Version 0.2
- `GuardBuiltins` now also checks the builtins and the globals of the current frame. In practice, the guard fails if it creates in a namespace and checked in a different namespace.
- Add a new `GuardGlobals` type which replaces the previous `guard_globals()` helper function (removed). The guard check checks if the frame globals changed or not.
- Guards are now tracked by the garbage collector to handle correctly a reference cycle with `GuardGlobals` which keeps a reference to the module namespace (`globals()`).
- Fix type of dictionary version for 32-bit platforms: `PY_UINT64_T`, not `size_t`.
- Fix `GuardFunc` traverse method: visit also the `code` attribute.
- Implement a traverse method to `GuardBuiltins` to detect correctly reference cycles.
- 2016-01-18: Version 0.1
  - `GuardBuiltins` check remembers if guard init failed
  - Rename `GuardGlobals` to `guard_globals()`
  - Rename `GuardTypeDict` to `guard_dict_type()`
- 2016-01-13: First public release, version 0.0.

## 1.3 Optimizations

Implemented optimizations:

- *Call pure builtins*
- *Loop unrolling*
- *Constant propagation*
- *Constant folding*
- *Replace builtin constants*
- *Dead code elimination*
- *Copy builtin functions to constants*
- *Simplify iterable*

### 1.3.1 Call pure builtins

Call pure builtin functions at compilation: replace the call with the result in the specialized bytecode, add guards on the called builtin functions.

The optimization is disabled when the builtin function is modified or if a variable with the same name is added to the global namespace of the function.

The optimization on the builtin `NAME` requires two guards:

- `NAME` key in builtin namespace
- `NAME` key in global namespace

Example:

Original	Specialized
<pre>def func():     return len("abc")</pre>	<pre>def func():     return 3</pre>

### 1.3.2 Loop unrolling

`for i in range(3): ...` and `for i in (1, 2, 3): ...` are unrolled. By default, only loops with 16 iterations or less are optimized.

---

**Note:** If `break` and/or `continue` instructions are used in the loop body, the loop is not unrolled.

---

*Configuration option:* `unroll_loops`.

#### tuple example

Example with a tuple.

Original	Loop unrolled
<pre>def func():     for i in ("a", "b"):         print(i)</pre>	<pre>def func():     i = "a"     print(i)      i = "b"     print(i)</pre>

No guard is required. The function has no specialized bytecode, the optimization is done directly on the function.

Original bytecode:

.	0 SETUP_LOOP	14 (to 17)
	3 LOAD_CONST	3 (('hello', 'world'))
	6 GET_ITER	
>>	7 FOR_ITER	6 (to 16)
	10 STORE_FAST	0 (i)
	13 JUMP_ABSOLUTE	7
>>	16 POP_BLOCK	
>>	17 LOAD_CONST	0 (None)
	20 RETURN_VALUE	

fatoptimizer bytecode:

LOAD_CONST	1 ("hello")
STORE_FAST	0 (i)
LOAD_CONST	2 ("world")
STORE_FAST	0 (i)
LOAD_CONST	0 (None)
RETURN_VALUE	

## range example

Example of a loop using `range()`.

Original	Loop unrolled
<pre>def func():     for i in range(2):         print(i)</pre>	<pre>def func():     i = 0     print(i)      i = 1     print(i)</pre>

The specialized bytecode requires two *guards*:

- range builtin variable
- range global variable

Combined with *constant propagation*, the code becomes even more interesting:

```
def func():
    i = 0
    print(0)

    i = 1
    print(1)
```

### 1.3.3 Constant propagation

Propagate constant values of variables.

Original	Constant propagation
<pre>def func()     x = 1     y = x     return y</pre>	<pre>def func()     x = 1     y = 1     return 1</pre>

*Configuration option:* constant\_propagation.

### 1.3.4 Constant folding

Compute simple operations at the compilation:

- arithmetic operations:
  - a+b, a-b, a\*b, a/b: int, float, complex
  - +x, -x, ~x: int, float, complex
  - a//b, a%b, a\*\*b: int, float
  - a<<b, a>>b, a&b, a|b, a^b: int
- comparison, tests:
  - a < b, a <= b, a >= b, a > b
  - a == b, a != b: don't optimize bytes == str
  - obj in seq, obj not in seq: for bytes, str, tuple seq
  - not x: int
- str: str + str, str \* int
- bytes: bytes + bytes, bytes \* int
- tuple: tuple + tuple, tuple \* int
- str, bytes, tuple, list: obj[index], obj[a:b:c]
- dict: obj[index]
- replace x in list with x in tuple if list only contains constants
- replace x in set with x in frozenset if set only contains constants
- simplify tests:

Code	Constant folding
<code>not(x is y)</code>	<code>x is not y</code>
<code>not(x is not y)</code>	<code>x is y</code>
<code>not(obj in seq)</code>	<code>obj not in seq</code>
<code>not(obj not in seq)</code>	<code>obj in seq</code>

Note: `not (x == y)` is not replaced with `x != y` because `not x.__eq__(y)` can be different than `x.__ne__(y)` for deliberate reason Same rationale for not replacing `not (x < y)` with `x >= y`. For example, `math.nan` overrides comparison operators to always return `False`.

Examples of optimizations:

Code	Constant folding
<code>-(5)</code>	<code>-5</code>
<code>+5</code>	<code>5</code>
<code>x in [1, 2, 3]</code>	<code>x in (1, 2, 3)</code>
<code>x in {1, 2, 3}</code>	<code>x in frozenset({1, 2, 3})</code>
<code>'Python' * 2</code>	<code>'PythonPython'</code>
<code>3 * (5,)</code>	<code>(5, 5, 5)</code>
<code>'python2.7'[:-2]</code>	<code>'python2'</code>
<code>'P' in 'Python'</code>	<code>True</code>
<code>9 not in (1, 2, 3)</code>	<code>True</code>
<code>[5, 9, 20][1]</code>	<code>9</code>

*Configuration option:* `constant_folding`.

### 1.3.5 Replace builtin constants

Replace `__debug__` constant with its value.

*Configuration option:* `replace_builtin_constant`.

### 1.3.6 Dead code elimination

Remove the dead code.

Examples:

Code	Dead code removed
<pre> if test:     pass else:     else_block </pre>	<pre> if not test:     else_block </pre>
<pre> if 1:     body_block </pre>	<pre> body_block </pre>
<pre> if 0:     body_block </pre>	<pre> pass </pre>
<pre> if False:     body_block else:     else_block </pre>	<pre> else_block </pre>
<pre> while 0:     body_block </pre>	<pre> pass </pre>
<pre> while 0:     body_block else:     else_block </pre>	<pre> else_block </pre>
<pre> ... return ... dead_code_block </pre>	<pre> ... return ... </pre>
<pre> ... raise ... dead_code_block </pre>	<pre> ... raise ... </pre>
<pre> try:     pass except ...:     ... </pre>	<pre> pass </pre>
<pre> try:     pass except ...:     ... else:     else_block </pre>	<pre> else_block </pre>
<pre> try:     pass except ...:     ... else:     else_block </pre>	<pre> try:     else_block finally:     final_block </pre>
<pre> 12 finally:     final_block </pre>	<p><b>Chapter 1. Table Of Contents</b></p>



**Note:** If a code block contains `continue`, `global`, `nonlocal`, `yield` or `yield from`, it is not removed.

*Configuration option:* `remove_dead_code`.

### 1.3.7 Copy builtin functions to constants

Opt-in optimization (disabled by default) to copy builtin functions to constants.

Example with a function simple:

```
def log(message):
    print(message)
```

Bytecode	Specialized bytecode
LOAD_GLOBAL 0 (print)	LOAD_CONST 1 (<built-in function print>)
LOAD_FAST 0 (message)	LOAD_FAST 0 (message)
CALL_FUNCTION 1 (1 positional, 0 keyword pair)	CALL_FUNCTION 1 (1 positional, 0 keyword pair)
POP_TOP	POP_TOP
LOAD_CONST 0 (None)	LOAD_CONST 0 (None)
RETURN_VALUE	RETURN_VALUE

The first `LOAD_GLOBAL` instruction is replaced with `LOAD_CONST`. `LOAD_GLOBAL` requires to lookup in the global namespace and then in the builtin namespaces, two dictionary lookups. `LOAD_CONST` gets the value from a C array, O(1) lookup.

The specialized bytecode requires two *guards*:

- `print` builtin variable
- `print` global variable

The `print()` function is injected in the constants with the `func.patch_constants()` method.

The optimization on the builtin `NAME` requires two guards:

- `NAME` key in builtin namespace
- `NAME` key in global namespace

This optimization is disabled by default because it changes the *Python semantics*: if the copied builtin function is replaced in the middle of the function, the specialized bytecode still uses the old builtin function. To use the optimization on a project, you may have to add the following *configuration* at the top of the file:

```
__fatoptimizer__ = {'copy_builtin_to_constant': False}
```

*Configuration option:* `copy_builtin_to_constant`.

See also:

- `codetransformer`: `@asconstants(len=len)` decorator replaces lookups to the `len` name with the builtin `len()` function
- Thread on python-ideas mailing list: [Specifying constants for functions](#) by Serhiy Storchaka, propose to add `const len=len` (or alternatives) to declare a constant (and indirectly copy a builtin functions to constants)

### 1.3.8 Simplify iterable

Try to replace literals built at runtime with constants. Replace also `range(start, stop, step)` with a tuple if the range fits in the *configuration*.

When `range(n)` is replaced, two guards are required on `range` in builtin and global namespaces and the function is specialized.

This optimization helps *loop unrolling*.

Examples:

Code	Simplified iterable
<code>for x in range(3): ...</code>	<code>for x in (0, 1, 2): ...</code>
<code>for x in {}:</code> ...	<code>for x in (): ...</code>
<code>for x in [4, 5, 6]: ...</code>	<code>for x in (4, 5, 6): ...</code>

*Configuration option:* `simplify_iterable`.

### 1.3.9 Comparison with the peephole optimizer

The CPython peephole optimizer only implements a few optimizations: *constant folding* and *dead code elimination*. fatoptimizer implements more *optimizations*.

The peephole optimizer doesn't support *constant propagation*. Example:

```
def f():
    x = 333
    return x
```

Regular bytecode		FAT mode bytecode	
LOAD_CONST	1 (1)	LOAD_CONST	1 (333)
STORE_FAST	0 (x)	STORE_FAST	0 (x)
LOAD_FAST	0 (x)	LOAD_CONST	1 (333)
RETURN_VALUE		RETURN_VALUE	

The *constant folding optimization* of the peephole optimizer keeps original constants. For example, `"x" + "y"` is replaced with `"xy"` but `"x"` and `"y"` are kept. Example:

```
def f():
    return "x" + "y"
```

Regular constants	FAT mode constants
<code>(None, 'x', 'y', 'xy')</code> : 4 constants	<code>(None, 'xy')</code> : 2 constants

The peephole optimizer has a similar limitation even when building tuple constants. The compiler produces AST nodes of type `ast.Tuple`, the tuple items are kept in code constants.

## 1.4 Python semantics and Limitations

fatoptimizer bets that the Python code is not modified when modules are loaded, but only later, when functions and classes are executed. If this assumption is wrong, fatoptimizer changes the semantics of Python.

### 1.4.1 Python semantics

It is very hard, to not say impossible, to implementation and keep the exact behaviour of regular CPython. CPython implementation is used as the Python “standard”. Since CPython is the most popular implementation, a Python implementation must do its best to mimic CPython behaviour. We will call it the Python semantics.

fattoptimizer should not change the Python semantics with the default configuration. Optimizations modifying the Python semantics must be disabled by default: opt-in options.

As written above, it’s really hard to mimic exactly CPython behaviour. For example, in CPython, it’s technically possible to modify local variables of a function from anywhere, a function can modify its caller, or a thread B can modify a thread A (just for fun). See [Everything in Python is mutable](#) for more information. It’s also hard to support all introspections features like `locals()` (`vars()`, `dir()`), `globals()` and `sys._getframe()`.

### 1.4.2 Builtin functions replaced in the middle of a function

fattoptimizer uses *guards* to disable specialized function when assumptions made to optimize the function are no more true. The problem is that guard are only called at the entry of a function. For example, if a specialized function ensures that the builtin function `chr()` was not modified, but `chr()` is modified during the call of the function, the specialized function will continue to call the old `chr()` function.

The *copy builtin functions to constants* optimization changes the Python semantics. If a builtin function is replaced while the specialized function is optimized, the specialized function will continue to use the old builtin function. For this reason, the optimization is disabled by default.

Example:

```
def func(arg):
    x = chr(arg)

    with unittest.mock.patch('builtins.chr', result='mock'):
        y = chr(arg)

    return (x == y)
```

If the *copy builtin functions to constants* optimization is used on this function, the specialized function returns `True`, whereas the original function returns `False`.

It is possible to work around this limitation by adding the following *configuration* at the top of the file:

```
__fattoptimizer__ = {'copy_builtin_to_constant': False}
```

But the following use cases works as expected in FAT mode:

```
import unittest.mock

def func():
    return chr(65)

def test():
    print(func())
    with unittest.mock.patch('builtins.chr', return_value="mock"):
        print(func())
```

Output:

```
A
mock
```

The `test()` function doesn't use the builtin `chr()` function. The `func()` function checks its guard on the builtin `chr()` function only when it's called, so it doesn't use the specialized function when `chr()` is mocked.

### 1.4.3 Guards on builtin functions

When a function is specialized, the specialization is ignored if a builtin function was replaced after the end of the Python initialization. Typically, the end of the Python initialization occurs just after the execution of the `site` module. It means that if a builtin is replaced during Python initialization, a function will be specialized even if the builtin is not the expected builtin function.

Example:

```
import builtins

builtins.chr = lambda: mock

def func():
    return len("abc")
```

In this example, the `func()` is optimized, but the function is *not* specialize. The internal call to `func.specialize()` is ignored because the `chr()` function was replaced after the end of the Python initialization.

### 1.4.4 Guards on type dictionary and global namespace

For other guards on dictionaries (type dictionary, global namespace), the guard uses the current value of the mapping. It doesn't check if the dictionary value was "modified".

### 1.4.5 Tracing and profiling

Tracing and profiling works in FAT mode, but the exact control flow and traces are different in regular and FAT mode. For example, *loop unrolling* removes the call to `range(n)`.

See `sys.settrace()` and `sys.setprofiling()` functions.

### 1.4.6 Expected limitations

Function inlining optimization makes debugging more complex:

- `sys.getframe()`
- `locals()`
- `pdb`
- etc.
- don't work as expected anymore

Bugs, shit happens:

- Missing guard: specialized function is called even if the "environment" was modified

FAT python! Memory vs CPU, fight!

- Memory footprint: loading two versions of a function is memory uses more memory
- Disk usage: `.pyc` will be more larger

Possible worse performance:

- guards adds an overhead higher than the optimization of the specialized code
- specialized code may be slower than the original bytecode

## 1.5 Benchmarks

fatoptimizer is not ready for macro benchmarks. Important optimizations like function inlining are still missing. See the *fatoptimizer TODO list*.

See *Microbenchmarks*.

### 1.5.1 The Grand Unified Python Benchmark Suite

Project hosted at <https://hg.python.org/benchmarks>

2016-01-22, don't specialized nested functions anymore:

```
$ time python3 ../benchmarks/perf.py ../default/python ../fatpython/python 2>&1|tee log
INFO:root:Automatically selected timer: perf_counter
INFO:root:Running `../fatpython/python ../benchmarks/lib3/2to3/2to3 -f all ../benchmarks/lib/2to3`
INFO:root:Running `../fatpython/python ../benchmarks/lib3/2to3/2to3 -f all ../benchmarks/lib/2to3` 1 t
INFO:root:Running `../default/python ../benchmarks/lib3/2to3/2to3 -f all ../benchmarks/lib/2to3`
INFO:root:Running `../default/python ../benchmarks/lib3/2to3/2to3 -f all ../benchmarks/lib/2to3` 1 t
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_chameleon_v2.py -n 50 --timer perf
INFO:root:Running `../default/python ../benchmarks/performance/bm_chameleon_v2.py -n 50 --timer perf
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_django_v3.py -n 50 --timer perf_c
INFO:root:Running `../default/python ../benchmarks/performance/bm_django_v3.py -n 50 --timer perf_c
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_pickle.py -n 50 --timer perf_cou
INFO:root:Running `../default/python ../benchmarks/performance/bm_pickle.py -n 50 --timer perf_coun
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_pickle.py -n 50 --timer perf_coun
INFO:root:Running `../default/python ../benchmarks/performance/bm_pickle.py -n 50 --timer perf_coun
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_json_v2.py -n 50 --timer perf_cou
INFO:root:Running `../default/python ../benchmarks/performance/bm_json_v2.py -n 50 --timer perf_coun
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_json.py -n 50 --timer perf_coun
INFO:root:Running `../default/python ../benchmarks/performance/bm_json.py -n 50 --timer perf_counter
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_nbody.py -n 50 --timer perf_coun
INFO:root:Running `../default/python ../benchmarks/performance/bm_nbody.py -n 50 --timer perf_counter
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_regex_v8.py -n 50 --timer perf_co
INFO:root:Running `../default/python ../benchmarks/performance/bm_regex_v8.py -n 50 --timer perf_coun
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_tornado_http.py -n 100 --timer pe
INFO:root:Running `../default/python ../benchmarks/performance/bm_tornado_http.py -n 100 --timer per

[ 1/10] 2to3...
[ 2/10] chameleon_v2...
[ 3/10] django_v3...
[ 4/10] fastpickle...
[ 5/10] fastunpickle...
[ 6/10] json_dump_v2...
[ 7/10] json_load...
[ 8/10] nbody...
[ 9/10] regex_v8...
[10/10] tornado_http...
```

```
Report on Linux smithers 4.2.8-300.fc23.x86_64 #1 SMP Tue Dec 15 16:49:06 UTC 2015 x86_64 x86_64
Total CPU cores: 8
```

```

### 2to3 ###
7.232935 -> 7.078553: 1.02x faster

### chameleon_v2 ###
Min: 5.740738 -> 5.642322: 1.02x faster
Avg: 5.805132 -> 5.669008: 1.02x faster
Significant (t=5.61)
Stddev: 0.17073 -> 0.01766: 9.6699x smaller

### fastpickle ###
Min: 0.448408 -> 0.454956: 1.01x slower
Avg: 0.450220 -> 0.469483: 1.04x slower
Significant (t=-8.28)
Stddev: 0.00364 -> 0.01605: 4.4102x larger

### fastunpickle ###
Min: 0.546227 -> 0.582611: 1.07x slower
Avg: 0.554405 -> 0.602790: 1.09x slower
Significant (t=-6.05)
Stddev: 0.01496 -> 0.05453: 3.6461x larger

### regex_v8 ###
Min: 0.042338 -> 0.043736: 1.03x slower
Avg: 0.042663 -> 0.044073: 1.03x slower
Significant (t=-4.09)
Stddev: 0.00173 -> 0.00172: 1.0021x smaller

### tornado_http ###
Min: 0.260895 -> 0.274085: 1.05x slower
Avg: 0.265663 -> 0.277511: 1.04x slower
Significant (t=-11.26)
Stddev: 0.00988 -> 0.00360: 2.7464x smaller

The following not significant results are hidden, use -v to show them:
django_v3, json_dump_v2, json_load, nbody.

real          20m7.994s
user          19m44.016s
sys           0m22.894s

```

2016-01-21:

```

$ time python3 ../benchmarks/perf.py ../default/python ../fatpython/python
INFO:root:Automatically selected timer: perf_counter
[ 1/10] 2to3...
INFO:root:Running `../fatpython/python ../benchmarks/lib3/2to3/2to3 -f all ../benchmarks/lib/2to3`
INFO:root:Running `../fatpython/python ../benchmarks/lib3/2to3/2to3 -f all ../benchmarks/lib/2to3` 1 t
INFO:root:Running `../default/python ../benchmarks/lib3/2to3/2to3 -f all ../benchmarks/lib/2to3`
INFO:root:Running `../default/python ../benchmarks/lib3/2to3/2to3 -f all ../benchmarks/lib/2to3` 1 t
[ 2/10] chameleon_v2...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_chameleon_v2.py -n 50 --timer perf_
INFO:root:Running `../default/python ../benchmarks/performance/bm_chameleon_v2.py -n 50 --timer perf_
[ 3/10] django_v3...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_django_v3.py -n 50 --timer perf_
INFO:root:Running `../default/python ../benchmarks/performance/bm_django_v3.py -n 50 --timer perf_
[ 4/10] fastpickle...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_pickle.py -n 50 --timer perf_cour
INFO:root:Running `../default/python ../benchmarks/performance/bm_pickle.py -n 50 --timer perf_count

```

```

[ 5/10] fastunpickle...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_pickle.py -n 50 --timer perf_count
INFO:root:Running `../default/python ../benchmarks/performance/bm_pickle.py -n 50 --timer perf_count
[ 6/10] json_dump_v2...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_json_v2.py -n 50 --timer perf_count
INFO:root:Running `../default/python ../benchmarks/performance/bm_json_v2.py -n 50 --timer perf_count
[ 7/10] json_load...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_json.py -n 50 --timer perf_count
INFO:root:Running `../default/python ../benchmarks/performance/bm_json.py -n 50 --timer perf_counter
[ 8/10] nbody...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_nbody.py -n 50 --timer perf_count
INFO:root:Running `../default/python ../benchmarks/performance/bm_nbody.py -n 50 --timer perf_counter
[ 9/10] regex_v8...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_regex_v8.py -n 50 --timer perf_count
INFO:root:Running `../default/python ../benchmarks/performance/bm_regex_v8.py -n 50 --timer perf_count
[10/10] tornado_http...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_tornado_http.py -n 100 --timer perf_count
INFO:root:Running `../default/python ../benchmarks/performance/bm_tornado_http.py -n 100 --timer perf_count

Report on Linux smithers 4.2.8-300.fc23.x86_64 #1 SMP Tue Dec 15 16:49:06 UTC 2015 x86_64 x86_64
Total CPU cores: 8

### 2to3 ###
6.969972 -> 7.362033: 1.06x slower

### chameleon_v2 ###
Min: 5.686547 -> 5.945011: 1.05x slower
Avg: 5.731851 -> 5.976754: 1.04x slower
Significant (t=-21.46)
Stddev: 0.06645 -> 0.04580: 1.4511x smaller

### fastpickle ###
Min: 0.489443 -> 0.448850: 1.09x faster
Avg: 0.518914 -> 0.458638: 1.13x faster
Significant (t=6.48)
Stddev: 0.05688 -> 0.03304: 1.7218x smaller

### fastunpickle ###
Min: 0.598339 -> 0.559612: 1.07x faster
Avg: 0.604129 -> 0.564821: 1.07x faster
Significant (t=13.55)
Stddev: 0.01493 -> 0.01408: 1.0601x smaller

### json_dump_v2 ###
Min: 2.794058 -> 4.456882: 1.60x slower
Avg: 2.806195 -> 4.467750: 1.59x slower
Significant (t=-801.42)
Stddev: 0.00722 -> 0.01276: 1.7678x larger

### regex_v8 ###
Min: 0.041685 -> 0.050890: 1.22x slower
Avg: 0.042082 -> 0.051579: 1.23x slower
Significant (t=-26.94)
Stddev: 0.00177 -> 0.00175: 1.0105x smaller

### tornado_http ###
Min: 0.258212 -> 0.272552: 1.06x slower
Avg: 0.263689 -> 0.280610: 1.06x slower

```

```
Significant (t=-8.59)
Stddev: 0.01614 -> 0.01130: 1.4282x smaller

The following not significant results are hidden, use -v to show them:
django_v3, json_load, nbody.

real      21m53.511s
user      21m29.279s
sys       0m23.055s
```

## 1.6 Microbenchmarks

REMINDER: on a microbenchmark, even a significant speedup doesn't mean that you will get a significant speedup on your application.

The benchmarks/ directory contains microbenchmarks used to test fat and fatoptimizer performances.

### 1.6.1 Function inlining and specialization using the parameter type

Optimize:

```
def _get_sep(path):
    if isinstance(path, bytes):
        return b'/'
    else:
        return '/'

def isabs(s):
    """Test whether a path is absolute"""
    sep = _get_sep(s)
    return s.startswith(sep)
```

to:

```
def isabs(s):
    return s.startswith('/')
```

but only if s parameter is a string.

2016-01-21:

```
original isabs() bytecode: 488 ns
_get_sep() inlined in isabs(): 268 ns (-220 ns, -45.0%, 1.8x faster :-))
```

2015-10-21:

```
$ ./python -m timeit 'import posixpath; isabs = posixpath.isabs' 'isabs("/root")'
1000000 loops, best of 3: 0.939 usec per loop
$ ./python -F -m timeit 'import posixpath; isabs = posixpath.isabs' 'isabs("/root")'
1000000 loops, best of 3: 0.755 usec per loop
```

Script: benchmarks/bench\_posixpath.py.



## 1.6.2 Move invariant out of loop (list.append)

Optimize:

```
def func(obj, data):
    for item in data:
        obj.append(item)
```

to:

```
def func(obj, data):
    append = obj.append
    for item in data:
        append(item)
```

2016-01-21:

```
range(10 ** 0)
- original bytecode: 297 ns
- append=obj.append with guards: 310 ns (+13 ns, +4.4%, 1.0x slower :-())
- append=obj.append: 306 ns (+9 ns, +3.1%, 1.0x slower :-())
range(10 ** 1)
- original bytecode: 972 ns
- append=obj.append with guards: 703 ns (-268 ns, -27.6%, 1.4x faster :-))
- append=obj.append: 701 ns (-271 ns, -27.9%, 1.4x faster :-))
range(10 ** 3)
- original bytecode: 72.2 us
- append=obj.append with guards: 43.8 us (-28.4 us, -39.4%, 1.6x faster :-))
- append=obj.append: 43.7 us (-28.6 us, -39.5%, 1.7x faster :-))
range(10 ** 5)
- original bytecode: 8131.5 us
- append=obj.append with guards: 5289.5 us (-2842.0 us, -35.0%, 1.5x faster :-))
- append=obj.append: 5294.2 us (-2837.4 us, -34.9%, 1.5x faster :-))
```

2015-10-21:

```
$ ./python bench.py
regular python: range(1)-> 502 ns
regular python: range(10)-> 1.7 us
regular python: range(10**3)-> 122.0 us
regular python: range(10**5)-> 8.5 ms

$ ./python -F bench.py
fat python: range(1)-> 479 ns (-5%)
fat python: range(10)-> 1.1 us (-35%)
fat python: range(10**3)-> 65.2 us (-47%)
fat python: range(10**5)-> 5.3 ms (-38%)
```

Script: benchmarks/bench\_list\_append.py.

## 1.6.3 Call builtin

Optimize:

```
def func():
    return len("abc")
```

to:

```
def func():
    return 3
```

2015-01-21 (best timing of 5 runs):

Test	Perf
Original bytecode (call len)	116 ns
return 3 with guard on builtins	90 ns
return 3	79 ns

GuardBuiltins has a cost of 11 ns.

Script: benchmarks/bench\_len\_abc.py.

## 1.6.4 Copy builtin function to constant

Optimize:

```
def func(obj):
    return len(obj)
```

to:

```
def func(obj):
    return 'LEN'(obj)
func.__code__ = fat.replace_consts(func.__code__, {'LEN': len})
```

2015-01-21 (best timing of 5 runs):

Test	Perf
Original bytecode (LOAD_GLOBAL)	121 ns
LOAD_CONST with guard on builtins	116 ns
LOAD_CONST	105 ns

GuardBuiltins has a cost of 11 ns.

Script: benchmarks/bench\_copy\_builtin\_to\_cst.py.

## 1.6.5 Copy global function to constant

Optimize:

```
mylen = len

def func(obj):
    return len(obj)
```

to:

```
mylen = len

def func(obj):
    return 'MYLEN'(obj)
func.__code__ = fat.replace_consts(func.__code__, {'MYLEN': len})
```

2015-01-21 (best timing of 5 runs):

Test	Perf
Original bytecode (LOAD_GLOBAL)	115 ns
LOAD_CONST with guard on globals	112 ns
LOAD_CONST	105 ns

GuardGlobals has a cost of 7 ns.

Script: benchmarks/bench\_copy\_global\_to\_cst.py.

## 1.6.6 Cost of guards

Cost of GuardDict guard.

2016-01-21:

```
no guard: 81 ns
with 1000 guards on globals: 3749 ns
cost of 1000 guards: 3667 ns (4503.4%)
average cost of 1 guard: 4 ns (4.5%)

no guard: 82 ns
with 100 guards on globals: 419 ns
cost of 100 guards: 338 ns (414.6%)
average cost of 1 guard: 3 ns (4.1%)

no guard: 81 ns
with 10 guards on globals: 117 ns
cost of 10 guards: 36 ns (43.9%)
average cost of 1 guard: 4 ns (4.4%)

no guard: 82 ns
with 1 guards on globals: 87 ns
cost of 1 guards: 5 ns (6.5%)
average cost of 1 guard: 5 ns (6.5%)
```

2016-01-06:

```
no guard: 431 ns
with 1000 guards on globals: 7974 ns
cost of 1000 guards: 7542 ns (1748.1%)
average cost of 1 guard: 8 ns (1.7%)

no guard: 429 ns
with 100 guards on globals: 1197 ns
cost of 100 guards: 768 ns (179.0%)
average cost of 1 guard: 8 ns (1.8%)

no guard: 426 ns
with 10 guards on globals: 515 ns
cost of 10 guards: 89 ns (20.8%)
average cost of 1 guard: 9 ns (2.1%)

no guard: 430 ns
with 1 guards on globals: 449 ns
cost of 1 guards: 19 ns (4.5%)
average cost of 1 guard: 19 ns (4.5%)
```

Script: benchmarks/bench\_guards.py.

## 1.7 Misc

### 1.7.1 Implementation

#### Steps and stages

The optimizer is splitted into multiple steps. Each optimization has its own step: `fatoptimizer.const_fold.ConstantFolding` implements for example constant folding.

The function optimizer is splitted into two stages:

- stage 1: run steps which don't require function specialization
- stage 2: run steps which can add guard and specialize the function

Main classes:

- `ModuleOptimizer`: Optimizer for `ast.Module` nodes. It starts by looking for `__fatoptimizer__ configuration`.
- `FunctionOptimizer`: Optimizer for `ast.FunctionDef` nodes. It starts by running `FunctionOptimizerStage1`.
- `Optimizer`: Optimizer for other AST nodes.

Steps used by `ModuleOptimizer`, `Optimizer` and `FunctionOptimizerStage1`:

- `NamespaceStep`: populate a `Namespace` object which tracks the local variables, used by `ConstantPropagation`
- `ReplaceBuiltinConstant`: replace builtin optimization
- `ConstantPropagation`: constant propagation optimization
- `ConstantFolding`: constant folding optimization
- `RemoveDeadCode`: dead code elimination optimization

Steps used by `FunctionOptimizer`:

- `NamespaceStep`: populate a `Namespace` object which tracks the local variables
- `UnrollStep`: loop unrolling optimization
- `CallPureBuiltin`: call builtin optimization
- `CopyBuiltinToConstantStep`: copy builtins to constants optimization

Some optimizations produce a new AST tree which must be optimized again. For example, loop unrolling produces new nodes like `"i = 0"` and duplicates the loop body which uses `"i"`. We need to rerun the optimizer on this new AST tree to run optimizations like constant propagation or constant folding.

### 1.7.2 Possible optimizations

Short term:

- Function `func2()` calls `func1()` if `func1()` is pure: inline `func1()` into `func2()`
- Call builtin pure functions during compilation. Example: replace `len("abc")` with `3` or `range(3)` with `(0, 1, 2)`.
- Constant folding: replace a variable with its value. We may do that for optimal parameters with default value if these parameters are not set. Example: replace `app.DEBUG` with `False`.

Using types:

- Detect the exact type of parameters and function local variables

- Specialized code relying on the types. For example, move invariant out of loops (ex: `obj.append` for list).
- `x + 0` gives a `TypeError` for str, but can be replaced with `x` for int and float. Same optimization for `x*0`.
- See `astoptimizer` for more ideas.

Longer term:

- Compile to machine code using Cython, Numba, PyPy, etc. Maybe only for numeric types at the beginning? Release the GIL if possible, but check “sometimes” if we got UNIX signals.

### 1.7.3 Pure functions

A “pure” function is a function with no side effect.

Example of pure operators:

- `x+y`, `x-y`, `x*y`, `x/y`, `x//y`, `x**y` for types int, float, complex, bytes, str, and also tuple and list for `x+y`

Example of instructions with side effect:

- “global var”

Example of pure function:

```
def mysum(x, y):
    return x + y
```

Example of function with side effect:

```
global _last_sum

def mysum(x, y):
    global _last_sum
    s = x + y
    _last_sum = s
    return s
```

### 1.7.4 Changelog

- 2016-01-23: Version 0.2
  - Fix the function optimizer: don’t specialized nested function. The specialization is more expensive than the speedup of optimizations.
  - Fix `Config.replace()`: copy logger attribute
  - `get_literal()` now also returns tuple literals when items are not constants
  - Adjust usage of `get_literal()`
  - `SimplifyIterable` also replaces empty dict (created a runtime) with an empty tuple (constant)
- 2016-01-18: Version 0.1
  - Add `fattoptimizer.pretty_dump()`
  - Add Sphinx documentation: `doc/` directory
  - Add benchmark scripts: `benchmarks/` directory
  - Update `fattoptimizer._register()` for the new version of the PEP 511 (`sys.set_code_transformers()`)

- 2016-01-14: First public release, version 0.0.

### 1.7.5 Constants

FAT Python introduced a new AST type: `ast.Constant`. The optimizer starts by converting `ast.NameConstant`, `ast.Num`, `ast.Str`, `ast.Bytes` and `ast.Tuple` to `ast.Constant`. Later, it can create constant of other types. For example, `frozenset('abc')` creates a `frozenset` constant.

Supported constants:

- None singleton
- bool: `True` and `False`
- numbers: `int`, `float`, `complex`
- strings: `bytes`, `str`
- containers: `tuple`, `frozenset`

### 1.7.6 Literals

Literals are a superset of constants.

Supported literal types:

- (all constant types)
- containers: `list`, `dict`, `set`

### 1.7.7 FunctionOptimizer

`FunctionOptimizer` handles `ast.FunctionDef` and emits a specialized function if a call to a builtin function can be replaced with its result.

For example, this simple function:

```
def func():
    return chr(65)
```

is optimized to:

```
def func():
    return chr(65)

_ast_optimized = func

def func():
    return "A"
_ast_optimized.specialize(func,
                          [{'guard_type': 'builtins', 'names': ('chr',)}])

func = _ast_optimized
del _ast_optimized
```

### 1.7.8 Detection of free variables

`VariableVisitor` detects local and global variables of an `ast.FunctionDef` node. It is used by the `FunctionOptimizer` to detect free variables.

### 1.7.9 Corner cases

Calling the `super()` function requires a cell variables.

## 1.8 fatoptimizer TODO list

### 1.8.1 Goal

To get visible performance gain, the following optimizations must be implemented:

- Function inlining
- Detect and call pure functions
- Elimination of unused variables (set but never read): the constant propagation and loop unrolling create many of them. For example, replace “`def f(): x=1; return x`” with “`def f(): return 1`”
- Copy constant global variable to function globals
- Specialization for argument types: move invariant out of loops. Ex: create a bounded method “`obj_append = obj.append`” out of the loop.

Other ideas of optimizations:

- Call pure methods of constants, ex: “`hello`”.`strip()`

Even if many optimizations can be implemented with a static optimizers, it’s still not a JIT compiler. A JIT compiler is required to implement even more optimizations.

### 1.8.2 Known Bugs

- `import *` is ignored

### 1.8.3 Search ideas of new optimizations

- [python.org wiki: PythonSpeed/PerformanceTips](http://python.org/wiki/PythonSpeed/PerformanceTips)
- Open issues of type Performance
- Closed issues of type Performance
- Unladen Swallow ProjectPlan
- Ideas from PyPy, Pyston, Numba, etc.

## 1.8.4 Needed to merge FAT mode into CPython

### Short term

- implement keywords in function calls

### Later

- implement `func.__sizeof__` and update `test_sys`
- Cleanup `Lib/test/test_fat.py`, especially `Lib/test/fattester.py`. Remove duplicated tests.
- handle python modules and python imports
  - checksum of the `.py` content?
  - how to handle C extensions? checksum of the `.so` file?
  - how to handle `.pyc` files?
- efficient optimizations on objects, not only simple functions
- need feedback from `pythran`, `numba` and others to check if the FAT mode has required features to be usable (useful) by these projects
- write unit tests for `sys.asthook`
- Usage of `locals()` or `vars()` must disable optimization. Maybe only when the optimizer produces new variables?

## 1.8.5 More optimizations

### MUST HAVE

More complex to implement (without breaking Python semantics).

- Remove useless temporary variables. Example:

Code:

```
def func():
    res = 1
    return res
```

Constant propagation:

```
def func():
    res = 1
    return 1
```

Remove `res` local variable:

```
def func():
    return 1
```

Maybe only for simple types (`int`, `str`). It changes object lifetime: <https://bugs.python.org/issue2181#msg63090>

- Compute if a function is pure. See `pythran.analysis.PureFunctions` of `pythran` project, depend on `ArgumentEffects` and `GlobalEffects` analysis
- Inline calls to short pure functions: see [Issue #10399](#), AST Optimization: inlining of function calls



- Inline calls to all functions, short or not? Need guards on these functions and the global namespace. Example: `posixpath._get_sep()`.
- Call pure functions of `math`, `struct` and `string` modules. Example: replace `math.log(32) / math.log(2)` with `5.0`.

## Random

Easy to implement.

- [Python-ideas] (FAT Python) Convert keyword arguments to positional? <https://mail.python.org/pipermail/python-ideas/2016-January/037874.html>
- Loop unrolling: support multiple targets:

```
for x, y in ((1, 2), (3, 4)):
    print(x, y)
```

- Tests:
  - `if a: if b: code => if a and b: code`
- Optimize `str%args` and `bytes%args`
- Constant folding:
  - replace `get_constant()` with `get_literal()`?
    - \* `list + list`
    - \* `frozenset | frozenset`
    - \* `set | set`
  - `2.0j ** 3.0`
  - `1 < 2 < 3`
  - `if x and True: pass => if x: pass` <http://bugs.python.org/issue7682>
  - replace `'(a and b) and c'` (2 op) with `'a and b and c'` (1 op), same for “or” operator
- Call methods of builtin types if the object and arguments are constants. Example: `"h\xe9ho".encode("utf-8")` replaced with `b"h\xc3\xa9ho"`.
- Optimize `fullvisit_AsyncFunctionDef`

## Can be done later

Unknown speedup, easy to medium to implement.

- Replace `dict(...)` with `{...}` (dict literal): <https://doughellmann.com/blog/2012/11/12/the-performance-impact-of-using-dict-instead-of-in-cpython-2-7-2/>
- Use `SimplifyIterable` for `dict/frozenset` argument
- `print()`: convert arguments to strings
- Remove dead code: remove “`pass; pass`”
- Simplify iterable:
  - `for x in set("abc"): ... => for x in frozenset("abc"): ...` Need a guard on `set` builtin
  - `for x in "abc": ... => for x in ("a", "b", "c"): ...` Is it faster? Does it use less memory?

## Can be done later and are complex

Unknown speedup, complex to implement.

- Implement CALL\_METHOD bytecode, but execute the following code correctly (output must be 1, 2 and not 1, 1):

```
class C(object):
    def foo(self):
        return 1
c = C()
print c.foo()
c.foo = lambda: 2
print c.foo()
```

Need a guard on C.foo?

See <https://bugs.python.org/issue6033#msg95707>

Is it really possible? FAT Python doesn't support guards on the instance dict, it's more designed to use guards on the type dict.

- Optimize 'lambda: chr(65)'. Lambda are functions, but defined as expressions. It's not easy to inject the func.specialize() call, func.\_\_code\_\_.replace\_consts() call, etc. Maybe only optimize in some specific cases?
- Enable copy builtins to constants when we know that builtins and globals are not modified. Need to ensure that the function is pure and only calls pure functions.
- Move invariant out of loops using guards on argument types:
  - Merge duplicate LOAD\_ATTR, when we can make sure that the attribute will not be modified
  - list.append: only for list type
- Loop unrolling:
  - support break and continue
  - support raise used outside try/except
- Constant propagation, copy accross namespaces:
  - list-comprehension has its own separated namespace:

```
n = 100
seq = [randrange(n) for i in range(n)]
```

- copy globals to locals: need a guard on globals
- Convert naive loop to list/dict/set comprehension. Replace “x=[]; for item in data: x.append(item.upper())” with “x=[item.upper() for item in data]”. Same for x=set() and x={ }.
- Call more builtin functions:
  - all(), any()
  - enumerate(iterable), zip()
  - format()
  - filter(pred, iterable), map(pred, iterable), reversed()
- operator module:
  - need to add an import, need to ensure that operator name is not used
  - lambda x: x[1] => operator.itemgetter(1)

- lambda x: x.a => operator.attrgetter('a')
- lambda x: x.f('a', b=1) => operator.methodcaller('f', 'a', b=1)
- map, itertools.map, filter:
  - [f(x) for x in a] => map(f, a) / list(map(f, a))
  - (f(x) for x in a) => itertools.map(f, a) / map(f, a) ? scope ?
  - (x for x in a if f(x)) => filter(f, a)
  - (x for x in a if not f(x)) => \_\_builtin\_\_filternot\_\_(f, a) ?
  - (2 \* x for x in a) => map((2).\_\_mul\_\_, a)
  - (x for x in a if x in 'abc') => filter('abc'.\_\_contains\_\_, a)

## 1.8.6 Profiling

- implement code to detect the exact type of function parameters and function locals and save it into an annotation file
- implement profiling directed optimization: benchmark guards at runtime to decide if it's worth to use a specialized function. Measure maybe also the memory footprint using tracemalloc?
- implement basic strategy to decide if specialized function must be emitted or not using raw estimation, like the size of the bytecode in bytes

## 1.8.7 Later

- find an efficient way to specialize nested functions
- configuration to manually help the optimizer:
  - give a whitelist of “constants”: app.DEBUG, app.enum.BLUE, ...
  - type hint with strict types: x is Python int in range [3; 10]
  - expect platform values to be constant: sys.version\_info, sys.maxunicode, os.name, sys.platform, os.linesep, etc.
  - declare pure functions
  - see fatoptimizer for more ideas
- Restrict the number of guards, number of specialized bytecode, number of arg\_type types with fatoptimizer.Config
- fatoptimizer.VariableVisitor: support complex assignments like 'type(mock).\_\_mock\_check\_sig = checksig'
- Support specialized CFunction\_Type, not only specialized bytecode?
- Add an opt-in option to skip some guards if the user knows that the application will never modify function \_\_code\_\_, override builtin methods, modify a constant, etc.
- Optimize real objects, not only simple functions. For example, inline a method.
- Function parameter: support more complex guard to complex types like list of integers?
- handle default argument values for argument type guards?
- Guards: give up on checking the value after N fails?
- How to distribute .pyc optimized code? Modify distutils?

- Really fix unit tests in FAT mode, don't skip them
  - Rewrite test\_metaclass without doctest to get assertEquals() to fix the test on type.\_\_prepare\_\_() result type
  - Fix test\_extcall in FAT mode: rewrite doctest to unittest
- Support locals()[key], vars()[key], globals()[key]?
- Support decorators
- Copy super() builtin to constants doesn't work. Calling the builtin super() function creates a free variable, whereas calling the constant doesn't create a free variable.
- test\_generators: rewrite test without doctest to check the type of g (see the diff in mercurial on Lib/test/test\_generators.py)

### 1.8.8 Support decorator

weakref.py:

```
@property
def atexit(self):
    """Whether finalizer should be called at exit"""
    info = self._registry.get(self)
    return bool(info) and info.atexit

@atexit.setter
def atexit(self, value):
    info = self._registry.get(self)
    if info:
        info.atexit = bool(value)
```

It's not possible to replace it with:

```
def atexit(self):
    """Whether finalizer should be called at exit"""
    info = self._registry.get(self)
    return bool(info) and info.atexit
atexit = property(atexit)

def atexit(self, value):
    info = self._registry.get(self)
    if info:
        info.atexit = bool(value)
atexit = atexit.setter(atexit)
```

The last line 'atexit = atexit.setter(atexit)' because 'atexit' is now the second function, not more the first decorated function (the property).

Define the second atexit under a different name? No! It changes the code name, which is wrong.

Maybe we can replace it with:

```
def atexit(self):
    """Whether finalizer should be called at exit"""
    info = self._registry.get(self)
    return bool(info) and info.atexit
atexit = property(atexit)

_old_atexit = atexit
def atexit(self, value):
```

```
info = self._registry.get(self)
if info:
    info.atexit = bool(value)
atexit = _old_atexit.setter(atexit)
```

But for this, we need to track the namespace during the optimization. The `VariableVisitor` in run *before* the optimizer, it doesn't track the namespace at the same time.



## A

arg\_index (GuardArgType attribute), 5  
arg\_types (GuardArgType attribute), 5

## C

code (GuardFunc attribute), 6  
Config (built-in class), 3

## D

dict (GuardDict attribute), 6

## F

FATOptimizer (built-in class), 3  
func (GuardFunc attribute), 6

## G

get\_specialized() (built-in function), 5  
guard\_globals (GuardBuiltins attribute), 6  
guard\_type\_dict() (built-in function), 7  
GuardArgType (built-in class), 5  
GuardBuiltins (built-in class), 5  
GuardDict (built-in class), 6  
GuardFunc (built-in class), 6  
GuardGlobals (built-in class), 6

## K

keys (GuardDict attribute), 6

## O

optimize() (built-in function), 3  
OptimizerError (built-in class), 3

## P

pretty\_dump() (built-in function), 3

## R

replace\_consts() (built-in function), 5

## S

specialize() (built-in function), 5