
VOC Documentation

Release 0.1.6

Russell Keith-Magee

May 07, 2019

Contents

| | | |
|----------|--------------------------|----------|
| 1 | Table of contents | 3 |
| 1.1 | Tutorial | 3 |
| 1.2 | How-to guides | 3 |
| 1.3 | Background | 3 |
| 1.4 | Reference | 3 |
| 2 | Community | 5 |
| 2.1 | Tutorials | 5 |
| 2.2 | How-to guides | 10 |
| 2.3 | About VOC | 15 |
| 2.4 | Reference | 20 |

VOC is a transpiler that takes Python 3.4+ source code, and compiles it into a Java class file that can then be executed on a JVM, or run through a DEX tool to run on Android. It does this *at the bytecode level*, rather than the source code level.

It honors Python 3.4+ syntax and conventions, but also provides the ability to reference objects and classes defined in Java code, and implement interfaces defined in Java code.

1.1 Tutorial

Get started with a hands-on introduction for beginners

1.2 How-to guides

Guides and recipes for common problems and tasks, including how to contribute

1.3 Background

Explanation and discussion of key topics and concepts

1.4 Reference

Technical reference - commands, modules, classes, methods

VOC is part of the [BeeWare suite](#). You can talk to the community through:

- [@pybeeware](#) on Twitter
- [pybee/general](#) on Gitter

2.1 Tutorials

These tutorials are step-by step guides for using VOC. They all assume that you've set up your development environment as described in [Installation](#).

2.1.1 Tutorial 0 - Hello, world!

In this tutorial, you'll take a really simple “Hello, world!” program written in Python, convert it into a classfile, and run it on the Java Virtual Machine.

Setup

This tutorial assumes you've read and followed the instructions in [Installation](#). If you've done this, you should have:

- Java 7 (or higher) installed and available on your path,
- An `env` directory for your virtualenv
- A `tutorial` directory with a VOC checkout,
- An activated Python 3.4+ virtual environment,
- VOC installed in that virtual environment,
- A compiled VOC support library.

Start a new project

Let's start by creating a `tutorial0` directory in the tutorial directory alongside the `voc` directory you just cloned into:

```
$ mkdir tutorial0
$ cd tutorial0
```

So that your directory structure looks like:

```
tutorial
├── env
├── tutorial0
└── voc
```

Then create a file called `example.py` in this `tutorial0` directory. Add the following Python code to `example.py`:

```
print("Hello World!")
```

Save the file. Run VOC over this file, compiling the Python code into a Java class file:

```
$ voc -v example.py
```

This runs the VOC compiler over the `example.py` source file. The `-v` flag asks VOC to use verbose output so you can see what is going on. You will see output like the following:

```
Compiling example.py ...
Writing python/example.class ...
```

This will produce an `example.class` in the `python` namespace. This classfile can run on any Java 6 (or higher) VM. To run the project, type:

- On Linux / OS X

```
$ java -classpath ../voc/dist/python-java-support.jar:. python.example
Hello World!
```

- On Windows

```
> java -classpath ../voc/dist/python-java-support.jar;. python.example
Hello World!
```

Congratulations! You've just run your first Python program under Java using VOC! Now you're ready to get a little more adventurous.

2.1.2 Tutorial 1 - Interoperating with Java

In this tutorial you'll learn how to use VOC to write Python code interoperating with Java, namely, how to write Python code that calls Java and how to make your code callable by Java.

Setup

This tutorial assumes you've read and followed the instructions in [Installation](#) and that you've successfully followed [Tutorial 0 - Hello, world!](#) – if you haven't done that yet, please do it before this.

Once you've followed that, it will be helpful if you set an environment variable `VOC_DIR` to your local VOC repository. If you're using Linux or Mac, you can do that by writing in your current terminal:

```
VOC_DIR=/path/to/voc
```

In case you don't really want to do that for some reason, you'll need to remember to replace `$VOC_DIR` by the full VOC directory any time it appears in the commands below.

Calling a Java method

Many Java APIs can be called by simply using the imports and calling with the equivalent Python types. Here is a simple example that calls the `System.getProperty()` Java method to show some information about your Java runtime environment:

```
from java.lang import System

print("Your JRE version is", System.getProperty('java.version'))
print("You're using Java from vendor", System.getProperty('java.vendor'))
print("The current class path is", System.getProperty('java.class.path'))
```

Try putting the above into a file named `javainfo.py`, and compile it with `voc -v javainfo.py`.

Then, run it with `java -cp $VOC_DIR/dist/python-java-support.jar:. python.javainfo`. You will see something like this

```
Your JRE version is 1.8.0_151
You're using Java from vendor Oracle Corporation
The current class path is /home/elias/src/voc-dev/voc/dist/python-java-support.jar:.
```

The actual text will vary according to your installed version of Java JDK.

The argument `-cp` is the so-called Java **classpath**, and is used by the JVM to locate packages and classes needed by a program.

How does it work?

Behind the scenes, VOC is generating code that uses proxy objects to access Java code, converts Python types like strings into Java strings before calling methods like Java's `System.getProperty()`.

To know more about how this work, see the section about *The VOC type system*.

Common problems

If you see an error message like:

```
Error: Could not find or load main class python.javainfo
```

This usually happens because the classpath is incorrect, causing the JVM to fail to find the class or something that was imported in a class. Ensure you're inside of the correct directory, and make sure to include the correct path to the `python-java-support.jar` file and the current directory in the classpath, separated by `:`.

Extending a Java class

For extending a Java class, you will need to use some special syntax. Here is an example code which creates three Java threads, by extending the `java.lang.Thread`:

```
from java.lang import Math

class MyThread(extends=java.lang.Thread):
    def __init__(self, id):
        self.id = id
        self.counter = 0

    def run(self) -> void:
        print('Starting thread %d' % self.id)
        for i in range(10):
            self.sleep(Math.random() * 1000)
            self.counter += 1
            print('Thread %d executed %d times' % (self.id, self.counter))

MyThread(1).start()
MyThread(2).start()
MyThread(3).start()
```

There are two important syntax features to notice here:

- 1) Extending a Java class: `MyThread(extends=java.lang.Thread)`

Note: Notice how we provide the full absolute path to the Java class being extended. This is required even if you import the class, because of a limitation of the current way the transpilation is implemented.

- 2) Annotating return type for the `run()` method: `-> void`. This is needed in order for the method to be executable from the Java side. In practice, VOC generates two methods like these: one to be callable from Python code, and the other with the Java types got from the annotations.

Compiling and running this will give you an output like:

```
Starting thread 1
Starting thread 2
Starting thread 3
Thread 3 executed 1 times
Thread 3 executed 2 times
Thread 1 executed 1 times
Thread 3 executed 3 times
Thread 2 executed 1 times
Thread 1 executed 2 times
Thread 3 executed 4 times
Thread 2 executed 2 times
Thread 3 executed 5 times
...
```

Example: extending HashMap

Here is another example, showing how to extend a Java class with slightly more involved type annotations, and calling the parent class method:

```

class SpyingHashMap(extends=java.util.HashMap):
    def __init__(self):
        self.counter = 0

    def put(self, key: java.lang.Object, value: java.lang.Object) -> java.lang.Object:
        print('Putting %s in key %s' % (value, key))
        return super().put(key, value)

m = SpyingHashMap()
m.put("hello", "it's me")
m.put("from where?", "the other side")
print('map entries are:', m.entrySet())

```

Here again it's important to notice how the type annotations need the full absolute path for the types.

Compiling and running the above code will give you:

```

Putting it's me in key hello
Putting the other side in key from where?
map entries are: [hello=it's me, from where?=the other side]

```

Common problems

- 1) Forgetting to declare `self` as argument for the run method, will give you an error like this:

```

Exception in thread "main" java.lang.ClassFormatError:
    Arguments can't fit into locals in class file python/extend_thread/MyThread

```

If you get the above error, double check that you're declaring the `self` as first argument in all methods of the Python classes.

- 2) Trying to extend a Java interface instead of implementing it, will give you this error:

```

Exception in thread "main" java.lang.IncompatibleClassChangeError:
    class python.error_extends.MyThread has interface java.lang.Runnable as super_
    ↪class

```

If you get the above error, make sure the thing you're trying to extend is a class and not an interface. Look below to see how to implement a Java interface.

Implementing a Java interface

Implementing a Java interface is similar to extending a Java class: much like in Java, you simply use `implements` instead of `extends`.

Here is the threads example from earlier, re-written to use a Python class implementing the Java interface `java.lang.Runnable`:

```

from java.lang import Math, Thread

class MyThread(implements=java.lang.Runnable):
    def __init__(self, id):
        self.id = id

```

(continues on next page)

(continued from previous page)

```
self.counter = 0

def run(self) -> void:
    print('Starting thread %d' % self.id)
    for i in range(10):
        Thread.sleep(Math.random() * 1000)
        self.counter += 1
        print('Thread %d executed %d times' % (self.id, self.counter))

Thread(MyThread(1)).start()
Thread(MyThread(2)).start()
Thread(MyThread(3)).start()
```

2.2 How-to guides

How-to guides are recipes that take the user through steps in key subjects. They are more advanced than tutorials and assume a lot more about what the user already knows than tutorials do, and unlike documents in the tutorial they can stand alone.

2.2.1 Contributing to VOC

If you experience problems with VOC, [log them on GitHub](#).

If you want to contribute code, please [fork the code](#) and [submit a pull request](#).

If you're a newcomer to the project looking for how to start contributing, you may find useful the [First Timers Guide](#).

Setting up your development environment

The process of setting up a development environment is very similar to the [Installation](#) process. The biggest difference is that instead of using the official PyBee repository, you'll be using your own Github fork.

As with the getting started guide, these instructions will assume that you have Python 3.4+, a Java ≥ 7 JDK, and Apache ANT installed.

Note: If you are on Linux, you will need to install an extra package to be able to run the test suite.

- **Ubuntu 12.04 and 14.04:** `libpython3.4-testsuite` This can be done by running `apt-get install libpython3.4-testsuite`.
- **Ubuntu 16.04 and 16.10:** `libpython3.5-testsuite` This can be done by running `apt-get install libpython3.5-testsuite`.
- **Ubuntu 18.04:** `libpython3.6-testsuite` This can be done by running `apt-get install libpython3.6-testsuite`.

Start by forking VOC into your own Github repository; then check out your fork to your own computer into a development directory:

```
$ mkdir voc-dev
$ cd voc-dev
$ git clone git@github.com:<your github username>/voc.git
```

Then create a virtual environment and install VOC into it:

```
$ python3 -m venv env
$ . env/bin/activate
$ cd voc
$ pip install -e .
```

For Windows the use of cmd under Administrator permission is suggested instead of PowerShell.

```
> py -3 -m venv env
> env\Scripts\activate.bat
> cd voc
> pip install -e .
```

You're now ready to run the test suite!

Running the test suite

To run the entire test suite, type:

```
$ python setup.py test
```

To capture unexpected successes and new failures in test:

```
$ python setup.py test 2>&1 | grep -E 'success|FAIL'
```

Running the full test suite will take quite a while - it takes 40 minutes on the CI server. You can speed this up by running the tests in parallel via pytest:

```
$ pip install -r requirements/tests.txt
$ py.test -n auto
```

You can specify the number of cores to utilize, or use `auto` as shown above to use all available cores.

If you just want to run a single test, or a single group of tests, you can provide command-line arguments.

To run a single test, provide the full dotted-path to the test:

```
$ python setup.py test -s tests.datatypes.test_str.BinaryStrOperationTests.test_add_
↪bool
```

To run a full test case, do the same, but stop at the test case name:

```
$ python setup.py test -s tests.datatypes.test_str.BinaryStrOperationTests
```

Or, to run all the Str datatype tests:

```
$ python setup.py test -s tests.datatypes.test_str
```

Or, to run all the datatypes tests:

```
$ python setup.py test -s tests.datatypes
```

Or you can use Cricket, a GUI tool for running test suites. To start cricket in the background:

```
$ pip install -r requirements/tests.txt
$ cricket-unittest &
```

This should open a GUI window that lists all the tests. From there you can “Run all” or select specific tests and “Run selected.”

Running the code style checks

Before sending your pull request for review, you may want to run the style checks locally.

These checks also run automatically in Travis, but you will avoid unnecessary waiting time if you do this beforehand and fix your code to follow the style rules.

In order to do that, first you need to install flake8:

```
pip install flake8
```

Then, whenever you want to run the checks, run the following command inside the project’s directory:

```
flake8 && ant checkstyle
```

Contributing tests for checking Standard Library Modules

- The purpose of the Standard Library tests are to ensure that the packages from the Python standard library are working within voc.
- You can check out the status of tests, such as if they exist and if they are passing, with the following commands from within the voc directory:
- `python tools/compile_stdlib.py java --collect-status && python tools/build_stdlib_report.py --html`
- Check out the resultant `voc/report.html` file.

How to create a new test

- Create a new python file in the `voc/stdlib_tests` directory with the name `test_LibraryName`. This test name must match the name of the python standard library module you are testing.
- Import the module that needs testing into the `test_LibraryName.py` file.
- Try to instantiate the module as an object and call multiple methods for it.
- Make sure you have followed the guide at *Installation*
- Compile the test `voc test_YourTestName`
- Run the code with `java -cp /YourPath/voc/dist/python-java-support.jar:/YourPath/ python.test_YourTestName`

Test Guidelines

- Try to avoid using other libraries.
- If using other libraries, be careful as they may not be implemented yet and this will cause further yak shaving.
- If the feature is not yet implemented, the tests will fail, but we will have some tests for when the feature is implemented and the report will be updated. Thanks for contributing!

Working with code for Java bytecode

If you find yourself needing to work with the parts of VOC that generates Java bytecode, you might find helpful these pointers:

- A [Python interpreter written in Python](#) will get you started on how stack based machines work. While the examples aren't for the JVM, the workings of the machines are similar enough to help you get used to the thinking.
- The *Java bytecode instructions* are represented by classes in `voc.java.opcodes` that inherit from `voc.java.opcodes Opcode`. Most of the code to generate bytecode is in the `voc.python.ast` module, and the bytecode generating code is often a sequence of instances of these opcode classes calling the method `add_opcodes()` for the current context.
- The `add_opcodes()` method also support helpers that work as pseudo-instructions, which allow to generate more complex sequences of instructions, like the `IF()`, `TRY()`, `CATCH()` from the `voc.voc.python.structures` module. It's easier to understand how these work finding an example of usage in VOC itself. Ask in Gitter, if you need help with it.

Troubleshooting generated bytecode

Troubleshooting issues in the generated bytecode can be a bit hard.

There are some tools that can help you to see what's going on. You can use a tool available in the [ASM](#) project to check the bytecode for problems.

Download the ASM binary distribution from the [ASM](#) project, extract the file in some directory and create a script like this:

```
ASM_VERSION=5.2
ASM_HOME=/path/to/asm-${ASM_VERSION}/lib

[ -n "$2" ] || { echo "Usage: $(basename $0) CLASSPATH CLASS_TO_ANALYSE"; exit 1; }

asm_file="$ASM_HOME/asm-${ASM_VERSION}.jar"
[ -f "$asm_file" ] || { echo "Couldn't find file $asm_file"; exit 1; }

classpath=$1
class_to_analyse=$2

java -cp "$ASM_HOME/asm-${ASM_VERSION}.jar:$ASM_HOME/asm-tree-${ASM_VERSION}.jar:$ASM_
↪HOME/asm-analysis-${ASM_VERSION}.jar:$ASM_HOME/asm-util-${ASM_VERSION}.jar:
↪$classpath" org.objectweb.asm.util.CheckClassAdapter $class_to_analyse
```

Then you can call it like:

```
asm.sh /PATH/TO/voc/dist/python-java-support.jar:. path.to.JavaClass
```

This will give you a brief diagnosis of problems found in the bytecode for the given Java class, and if possible will print a friendlier version of the bytecode.

If you just want to see a human friendly version of the Java bytecode to double check the generated code, you can also try the command:

```
javap -c path.to.JavaClass
```

2.2.2 Release Process

Note: For Developers Only!

This guide is provided for the benefit of the VOC team. As an end user, you shouldn't ever need to use these instructions.

So, it's time for a new VOC release! Here's how to publish a new version so others can benefit from the changes that have been made recently.

1. Update local checkout

Make sure your developer checkout of VOC is up to date with a:

```
$ git pull
```

2. Confirm that the trunk currently builds for JDK and Android on each version of Python you're planning to support:

```
$ ant clean
$ ant
```

Fix any problems that are identified

3. Make release related changes

- **Release history** in `docs/background/releases.rst`
- **Build number** in `build.xml`
- **Version number** in `voc/__init__.py`

4. Push to Github to get confirmation of a clean CI build.

5. When CI passes, merge.

6. Update your checkout of the main `pybee/voc` repository

7. Tag the release. There is a version tag for VOC, plus tags for each of the support libraries that will be released:

```
$ git tag v0.1.2
$ git tag 3.4-b3
$ git tag 3.5-b3
$ git tag 3.6-b3
$ git push --tags
```

8. Build the PyPI packages:

```
$ python setup.py sdist bdist_wheel
```

9. Upload the PyPI packages:

```
$ twine upload dist/voc-0.1.2*
```

10. Check that you have AWS credentials in a file named `.env` file in the root directory of your project checkout:

```
AWS_ACCESS_KEY_ID=... AWS_SECRET_ACCESS_KEY=... AWS_REGION=us-west-2
```

11. Upload the support zipfile to S3:

```
$ python tools/upload b3
```

11. Check that Read The Docs has updated.

2.3 About VOC

VOC is a transpiler that takes Python 3.4+ source code, and compiles it into a Java class file that can then be executed on a JVM.

It honors Python 3.4+ syntax and conventions, but also provides the ability to reference objects and classes defined in Java code, and implement interfaces defined in Java code.

2.3.1 The VOC Developer and User community

Note: If you're a newcomer to the project looking for how to start contributing, check out the [First Timers Guide](#).

VOC is part of the [BeeWare suite](#). You can talk to the community through:

- [@pybeeware on Twitter](#)
- [Our Gitter channel](#) for discussion about development and general help around this project or anything under the Beeware suite of projects.

Code of Conduct

The BeeWare community has a strict [Code of Conduct](#). All users and developers are expected to adhere to this code.

If you have any concerns about this code of conduct, or you wish to report a violation of this code, please contact the project founder *Russell Keith-Magee*.

Contributing

If you experience problems with VOC, [log them on GitHub](#). If you want to contribute code, please [fork the code](#) and submit a [pull request](#).

2.3.2 Frequently Asked Questions

Is VOC a source code converter?

No. VOC operates *at the bytecode level*, rather than the source code level. In the initial versions, VOC would take CPython bytecode format (the *.pyc* files generated at runtime by CPython when the code is imported for the first time), and convert that bytecode directly into Java bytecode in a *.class* file.

Currently, VOC uses the Python `ast` module to parse the Python code, and generates the Java bytecode from that.

No intermediate Java source file is generated.

Isn't this the same thing as Jython?

No. Jython is an implementation of a Python interpreter in Java. This means it provides a REPL (an interactive prompt), and you *run* your Python code through Jython. VOC converts Python directly to a Java classfile; The VOC executable isn't needed at runtime (although there is a runtime support library that *is* needed).

The *clamped* extension to Jython enable you to use Jython as a generator of class files - this is a closer analogy to what VOC does.

The easiest way to demonstrate the difference between Jython and VOC is to look at the *eval()* and *exec()* methods. In Jython, these are key to how the process works, because they're just hooks into the runtime process of parsing and evaluating Python code. In VOC, these methods would be difficult to implement because VOC compiles all the class files up front. To implement *eval()* and *exec()*, you'd need to run VOC through VOC, and then expose an API that could be used at runtime to generate new *.class* files.

How fast is VOC?

Faster than a slow thing; slower than a fast thing :-)

Programming language performance is always nebulous to quantify. As a rough guide, it's about an order of magnitude slower than CPython on the same machine.

This means it probably isn't fast enough for an application that is CPU bound. However, if this is the case, you can always write your CPU bound parts in *pure* Java, and call those directly from Python, same as you would for a CPython extension.

It should also be noted that VOC is a very young project, and very little time has been spent on performance optimization. There are many obvious low hanging performance optimizations that could be explored as the project matures.

What can I use VOC for?

You can use VOC anywhere that provides a Java runtime environment, but you want to write your logic in Python. For example:

- Writing Android applicaitons
- Writing Lucene/ElasticSearch custom functions
- Writing Minecraft plugins
- Writing web applications to deploy in a J2EE container

In each of these cases, the project provides a Java (or Java-like, in the case of Android) environment. While some bridging might be possible with JNI, or by writing a thin Java shim that calls out to another language environment, these approaches mean you're developing a plugin at arms length.

The VOC approach allows you to develop your Python application *as if it were native*. The class files even have references to the Python source code, so when a stack trace is generated, it will tell you the line of Python source that caused the problem.

What version of Python does VOC require?

VOC runs under Python 3.4+.

What version of Java does VOC require?

VOC runs on:

- Java 6 without any special handling;
- Java 7 by enabling the `-XX:-UseSplitVerifier` flag at runtime;
- Java 8 by enabling the `-noverify` flag at runtime.

The complication with Java 7 and Java 8 is due to a feature of class files (called a Stack Map Frame) that was introduced as an optional feature in Java 6, and has been decreasingly optional in each subsequent release.

It would be entirely possible to generate Stack Map Frames for the generated class files from the information in a Python class file, but the developers haven't had a chance to get around to that yet.

Why “VOC”?

The *Vereenigde Oostindische Compagnie* (VOC), or Dutch East India Company, is often considered to be the world's first multinational corporation. It was also the first company to issue shares, and facilitate the trading of those shares. It was granted a 21 year monopoly to carry out trade activities in Asia, primarily the Spice Islands - the Dutch East Indies. They established a major trading port at Batavia - now Jakarta, on the island of Java (now part of Indonesia). As a result of their monopoly, the VOC became an incredibly valuable company, issuing an 18% annual dividend for almost 200 years.

VOC was... the world's first Enterprise site in Java. (rimshot!)

Can I make an Android app already?

Yes, but currently you have to use Android Java API (you'll be able to use `toga` once `toga-android` is more mature).

You can see [here an example TicTacToe app that does that](#).

2.3.3 Installation

In this guide we will walk you through setting up your VOC environment for development and testing. We will assume that you have Python 3.4 or 3.5, Java 7 or Java 8 JDK, and Apache ANT installed.

Checking Dependencies

To check if you have Python installed, run `python --version` at the command line

```
$ python --version
Python 3.4.4
```

If you do not have Python 3.4 or newer [install Python](#) and check again.

To check if you have the JDK installed, run `javac -version`

```
$ javac -version
javac 1.7.0_101
```

If you do not have at least Java 7 [install Java](#) and check again.

To check if Apache ANT is installed, run `ant -version`

```
$ ant -version
Apache Ant(TM) version 1.9.7 compiled on April 24 2016
```

If Apache Ant is not installed, look for the binary file from [Apache](#) to download the latest version.

Get a copy of VOC

The first step is to create a project directory, and clone VOC:

```
$ mkdir tutorial
$ cd tutorial
$ git clone https://github.com/pybee/voc.git
```

Then create a virtual environment and install VOC into it:

```
$ python3 -m venv env
$ . env/bin/activate
$ cd voc
$ pip install -e .
```

For Windows the use of cmd under Administrator permission is suggested instead of PowerShell.

```
> py -3 -m venv env
> env\Scripts\activate.bat
> cd voc
> pip install -e .
```

Building the support JAR file

Next, you need to build the Python support file:

```
$ ant java
```

This should create a `dist/python-java-support.jar` file. This JAR file is a support library that implements Python-like behavior and provides the Python standard library for the Java environment. This JAR file must be included on the classpath for any VOC-generated project.

Next Steps

You now have a working VOC environment, so you can *start the first tutorial*.

2.3.4 Release History

0.1.6

- Major speed improvements in generated code
- Improvements to generator handling
- Improvements to exception handling
- Added support for *yield from*

0.1.5

- Added support for if conditions in list comprehensions
- Added support for printf-style formatting
- Several fixes for arguments handling and operations with numbers
- Many bugfixes and improvements to docs, tests and error messages
- Added implementations for many builtin functions and type methods
- VOC jar library now ships with the stdlib modules that it already compiles

0.1.4

- Added support for self-assignment for attributes and subscript
- Added support for multiple comparison
- Improved support for slices
- Many bugfixes and improvements to docs and tests
- Added implementations for many builtin functions and type methods
- VOC is now tested in Python 3.6, and uses BeeKeeper for CI

0.1.3

Progress release. VOC is able to run Toga tutorials on Android.

0.1.2

Progress release. VOC is able to compile the Toga and Colosseum libraries.

0.1.1

Switched approach to parse Python code with `ast` instead of transpiling CPython bytecode.

Many bugs fixed.

0.1.0

Progress release. VOC is currently able to compile a simple working Android application.

0.0.1

Initial public release.

2.3.5 VOC Roadmap

VOC is a new project - we have lots of things that we'd like to do. If you'd like to contribute, providing a patch for one of these features:

- Port a set of basic type operations
- Implement a Python standard library module for
- Implement StackMapFrames for the generated Java class files.
- Work out how to run the CPython test suite with VOC

2.4 Reference

This is the technical reference for public APIs provided by VOC.

2.4.1 Python signatures for Java-defined methods

In essence, a method is not much different from a Java method. One defines a method as follows. Here is a simple example of a 1-argument function.

```
@org.python.Method(  
    __doc__ = "foobar(fizz) -> buzz" +  
        "\n" +  
        "Return the foobarified version of fizz.\n",  
    args = {"fizz"}  
)  
public function org.python.Object foobar(org.python.Object fizz) {  
    return buzz;  
}
```

The `org.python.Method` creates an annotation on the method. Allowable values are

name The name of the method. If not specifies, uses reflection to get the name.

__doc__ The documentation string of the method.

args An array of argument names.

varargs The name of the argument that should get all other values.

default_args An array of argument names that get “default” values. The handling of the default values should be done by checking the argument null‘

kwonlyargs An array of arguments that may only be supplied as a keyword argument.

kwargs A name of the argument that recieves the keyword arguments.

Examples

Because examples speak clearer than a thousand words.

A function with no arguments

Here is a sample of a function always returning the same value. Since it has no arguments, there is no need to supply any of the named

```
def constant_4():
    """Return 4, always and ever."""
    return 4
```

```
@org.python.Method(
    __doc__ = "Return 4, always and ever."
)
public org.python.Object constant_4() {
    return org.python.types.Int.getInt(4);
}
```

A function with two arguments

Another simple function is that of adding two given numbers.

```
def add(num1, num2):
    """Add two numbers."""
    return num1 + num2
```

```
@org.python.Method(
    __doc__ = "Add two numbers.",
    args = {"num1", "num2"}
)
public org.python.Object add(org.python.Object num1, org.python.Object num2) {
    // Left as exercise for the reader.
}
```

A function with a default argument

Similarly, we might want to make the second argument optional, allowing you to either add 1 to the number, or the supplied argument.

```
def inc(num, delta=1):
    """Increment a number."""
    return num + delta
```

```
@org.python.Method(
    __doc__ = "Add two numbers.",
    args = {"num"},
    default_args = {"delta"}
)
public org.python.Object inc(org.python.Object num, org.python.Object delta) {
    if (delta == null) {
        delta = org.python.types.Int.getInt(1);
    }
    // Left as exercise for the reader.
}
```

A function with variable arguments

Of course, sometimes you don't want to specify a specific number of arguments, but accept as many as you can get. For instance, the `min` function.

```
def min(first, *others):
    """Get the minimum of the supplied arguments."""
    val = first
    for other in others:
        if other < val:
            val = other
    return val
```

```
@org.python.Method(
    __doc__ = "Get the minimum of the supplied arguments.",
    args = {"first"},
    varargs = "others"
)
public org.python.Object min(org.python.Object first, org.python.types.Tuple others) {
    org.python.Object val = first;
    for (org.python.Object other: others) {
        if (other.__lt__(val)) {
            val = other;
        }
    }
    return val;
}
```

A function accepting keyword arguments

```
def loop_kwargs(**kwargs):
    """Loop over the kwargs to this function."""
    for k in kwargs:
        pass
```

```
@org.python.Method(
    __doc__ = "Loop over the kwargs to this function.",
    args = {},
    kwargs = "kwargs"
)
public org.python.Object loop_kwargs(org.python.types.Dict kwargs) {
    Map<org.python.Object, org.python.Object> kwargValues = kwargs.value;
    for (org.python.Object key : kwargValues.keySet()) {
        // The keys will always be python Str objects
        org.python.types.Str keyStr = (org.python.types.Str) key;
    }
    return org.python.types.NoneType.NONE;
}
```

2.4.2 The VOC type system

VOC works by operating on a layer of “Python-like” objects. A Python-like object is any object that implements the `org.python.Object` interface. This interface consists of all the “dunder” methods, like `__getattr__`, `__setattr__`, `__init__`, `__repr__` and `__str__`, that the Python interpreter might use on a Python object.

The default implementation of `org.python.Object` is `org.python.types.Object`. This is the VOC representation of the base `Object` class. As any Python object can be thrown as an exception, `org.python.types.Object` extends `java.lang.RuntimeException`.

The Python `dict` builtin type is implemented in the class `org.python.types.Dict`. This class is a subclass of `org.python.types.Object`. All methods and attributes of a Python `dict` are implemented as instance methods and attributes of this class.

The Python builtin type `type` is implemented as `org.python.types.Type`, which is *also* a subclass of `org.python.types.Object`. Instances of `org.python.types.Type` contain a reference to the Java class that instances of that Python type will be constructed from. As a result, instances of `org.python.types.Type` can be invoked as a function to create instances of the class wrapped by the type. All instances of Python-like objects can be interrogated for their type. There will only be one instance of `org.python.types.Type` for any given Python-like object.

So - there is an instance of `org.python.types.Type` that refers to `org.python.types.Dict`; and all instances of `org.python.types.Dict` contain a reference of that `org.python.types.Type` instance. The `org.python.types.Type` instance referring to `org.python.types.Dict` (which will be indexed as `"dict"`) can be invoked to create new `org.python.types.Dict` instances.

Type origins

VOC Types are classified according to their origin. There are four possible origins for a type:

- Builtin types
- Python types
- Java types
- Extension types

Builtin Types

These are data types built into the VOC support library. All the basic Python types like `dict` and `list` are Builtin types. The standard Python exceptions are also builtin types.

Python instance attributes are stored on the Java instance. When storing instance attributes, VOC will look for a `Field` on the Java class that matches the name of the Python attribute; if one exists, and it has been annotated in the Java source with a `@org.python.Attribute` annotation, that field will be used for storage. Otherwise, the value will be placed in the `__dict__` for the instance.

Python instance methods are instance methods on the Java class, with prototypes that match Python name-for-name, excluding the `self` argument, which will be implicitly added. `*args` is mapped to `org.python.Object [] args`, and `**kwargs` to `java.util.Map<java.lang.String, org.python.Object> kwargs`. Arguments with default values should be passed in as `null` (a Java `null`, not a Python `None`); the method implementation is then responsible for substituting an appropriate Python value if a `null` was provided in the argument list.

- Each `Object` class has a static `__class__` attribute, which is an instance
 - of `org.python.types.Type()`, constructed wrapping the Java class
 - implementing instances of the Python instances of that type. This type can
 - be retrieved by calling the `type()` method on the Java instance (which is
 - part of the `org.python.Object` interface)

Python Types

Python types are types that are declared in Python, extending the base Python `object` (either explicitly, implicitly, or as a subclass of a class that is itself an explicit or implicit subclass of `object`).

- All Python instance attributes are stored as values in `__dict__`.
- Python instance methods are rolled out as a pair of methods on the Java class:
 - a static method that takes an extra *self* argument
 - an instance method

Java Types

Any object that exists in the Java namespace can be proxied into the Python environment as a Java Type.

The Java object instance is wrapped in an implementation of `org.python.java.Object`, which acts as a proxy tying python `__getattr__` and `__setattr__` to the equivalent reflection methods in Java.

Extension Types

Extension types are types that have been declared in Python, but extend a Java type.

Implementation details

There are quirks to some of the implementations of some Python types.

Modules

- Implemented in a `__init__.class` file, regardless of whether one is actually used in the Python source.
- Instance of a class, extending `org.python.types.Module`
- Registered as `sys.modules[modulename]`

Class

- Implemented in a `<classname>.class` file

2.4.3 Differences between VOC and CPython

StopIteration

A `StopIteration` is a signal raised by an iterator to tell whomever is iterating that there are no more items to be produced. This is used in `for` loops, generator functions, etc. The `org.python.exception.StopIteration` exception differs from the CPython `StopIteration` exception in that it is a singleton. This was introduced in [‘PR #811<https://github.com/pybee/voc/pull/881>’](https://github.com/pybee/voc/pull/881) as part of a performance effort as it yields a non-trivial performance improvement for nested `for` loops. However, it also means that an equality comparison between two `StopIteration` exceptions will always be `True`, which is not the case in CPython.