Python for the busy Java Developer

Deepak Sarda

Python for the busy Java Developer

Deepak Sarda

Revision 1.0
15 September, 2014

Table of Contents

Introduction

1. Hello There!

If you are reading this book, then chances are that you are a busy Java® developer who is interested in learning Python®. If so, I hope that by the time you are done reading this short book:

1. you will have gained sufficient familiarity with the Python language syntax so that you'd be able to read some code and understand what it is doing.
2. get enough orientation to be able to navigate the Python ecosystem of libraries & tools.

This book is *not* for the beginner programmer. I assume that you are comfortable programming in Java (or some other OOP language like C#) and hence, I will not bore you with explanations of basic concepts like variables, functions and classes.

2. About the book

This book is divided into three broad chapters:

- The Language
- The Syntax
- The Ecosystem

In the first chapter, we will take a brief look at the Python language itself and understand what it has to offer. Next, we'll get down to the details of The Syntax before wrapping up with a look at the wider Ecosystem surrounding the Python language.

3. About the author

I've been working as a software developer for more than a decade now, spanning multiple business domains in a variety of programming languages & environments. I've worked on several high-performance, server-side applications written in Java. I've also done web development and systems automation work in Python. I've had the opportunity to work in a startup, in Fortune 50 companies and a couple of places in between. My full resume is available on my website[1].

[1] http://antrix.net/resume/

Offline, I live in Singapore with my lovely wife and our adorable daughter. Online, I can be found blogging at antrix.net[2] and tweeting as @antrix[3].

I'd love to hear what you've to say about this book — please email me at deepak@antrix.net[4].

4. Acknowledgments

I'd always heard it being said, but only now do I truly realize it: *writing a book is hard work!* It would have been harder still had it not been for the support from my family and friends.

I wish to especially thank Hitesh Sarda, Rohit Sharma and Srijith Nair for the incredibly detailed and thoughtful feedback that they provided as I wrote this book. It is much better thanks to them.

I must also acknowledge the constant encouragement that I received from my wife, Sonika. I can't thank her enough for her patience and support as I took time out to write this book.

[2] http://antrix.net
[3] https://twitter.com/antrix
[4] mailto:deepak@antrix.net

Chapter 1. The Language

1.1. What is Python?

> Python is an open-source, general purpose programming language that is dynamic, strongly-typed, object-oriented, functional, memory-managed and above all, fun to use!

Those are a lot of adjectives for one sentence! Let's unpack them one at a time.

Python is distributed under an **open source**, BSD-style license called the *Python Software Foundation License Agreement*[1]. This is a very permissive license that allows great flexibility in how Python can be used. Python's development is done in the open by a large and diverse community of volunteers.

Python is **general purpose** in that you can use it to build a variety of applications running the gamut from simple scripts and command-line tools to web applications, network servers, GUIs, scientific applications, etc.

We know that Java is a statically typed language, i.e. the types are checked and enforced at compile time. In contrast, Python is **dynamic** which means that the types are checked only at runtime. But Python is also **strongly typed**, just like Java. You can only execute operations that are supported by the target type.

Another way to think about this is that while in Java, both variables and objects have types associated with them, in Python only objects have types, not the variables that they are bound to. In Java, when we declare:

```
MyType obj = new MyType()
```

The variable `obj` is declared of type `MyType` and then the newly instantiated object of type `MyType` is assigned to it. In contrast, in Python, the same declaration would read:

```
obj = MyType()
```

Ignoring the missing `new` keyword (which Python doesn't have), `obj` is simply a name that is bound to the object on the right, which happens to be of type `MyType`. We can even reassign

[1] https://docs.python.org/2/license.html

`obj` in the very next line – `obj = MyOtherType()` – and it wouldn't be a problem. In Java, this re-assignment would fail to compile[2] while in Python, the program will run and will only fail at runtime if we try to execute an operation via `obj` that is incompatible with the type assigned to it at that point in time.

Python is **object oriented** and supports all the standard OOP features that Java has like creation of types using classes, encapsulation of state, inheritance, polymorphism, etc. It even goes beyond Java and supports features such as multiple inheritance, operator overloading, meta-programming, etc.

Python also supports a rich set of **functional** programming features and idioms. In Python, functions are first-class objects that can be created, manipulated and passed around just like any other object. While not as functional as say *Clojure*, Python certainly offers much more to the functional programmer than Java.[3]

Another similarity between the languages is in terms of manual **memory management**, in that there is none. The language runtime takes care of correctly allocating and freeing up memory, saving the programmer from the drudgery – and mistakes – of manually managing memory. Having said that, the JVM garbage collectors are much, much better performing than the Python GC. This can become a concern depending on the type of application you are building.

Finally, and above all, Python is *fun* and a joy to use. This is a strong claim to make but I hope that by the time you are done reading this book, you'll agree with me and the millions of other Python programmers out there!

1.2. History

Java	**Python**
James Gosling	Guido van Rossum
From C++/Oak	From ABC
1.0 - Jan 1996	1.0 - Jan 1994
8.0 - Mar 2014	3.4 - Mar 2014
JSR	PEP
Commercial	Community

[2] Unless `MyOtherType` happens to be a subclass of `MyType`.

[3] Even after the introduction of lambdas in Java 8.

> I'll use this table format to compare and contrast Python and Java whenever it makes sense.

Python is the brain-child of a Dutch programmer named Guido van Rossum. He started working on it when he got frustrated with the ABC language in the late 80s and after some years of private development, released the first version of Python in 1994. This actually makes Python older than Java, the first version of which was released in 1996, a full two years later! Since then, the language has continued to refine and evolve, with Python 2.0 being released in 2000. As of this writing, the 2.x versions are the most widely deployed.

In version 3.0, the language designers decided to break backwards compatibility in order to clean up some of the accumulated language warts. Although this has been good from a language perspective, it has been a significant hindrance to those upgrading from 2.x to 3.x. Imagine if Sun had decided to introduce generics in Java 5 *without* type erasure, thus breaking backwards compatibility? The Java language would've been much nicer today but the transition period would've been difficult, to say the least. That is the kind of transition the Python user community is going through right now.

> Since 2.x is still the most widely used version of Python, this book will cover Python 2.x features and syntax, calling out any differences with 3.x from time to time.

From the outset, Python's development has been done in the open with a community of volunteers contributing to the language and the core libraries. Any language changes are proposed and discussed through a process called the *Python Enhancement Proposals*, with Guido having final say in deciding the outcome. For his stewardship and continuous involvement in the development of Python, Guido is affectionately called the *Benevolent Dictator For Life*. He also periodically writes a Python History blog[4] chronicling the evolution of various language features.

1.3. Installation

This book is full of example code and the best way to follow along is to actually try out these examples by yourself. To do this, you'll obviously need to install Python on your system. But an easier way is to check if you already have access to a system with Python installed! Almost all systems running Linux should have Python pre-installed. Recent versions of Mac OS X also come with Python pre-installed. Just open a command shell on either of these two systems

[4] http://python-history.blogspot.com/

and type in `python`. If you get a Python shell prompt, you are all set! The version of Python installed may be a bit outdated but it should be sufficient to get started.

If you can't find a system with Python installed on it, just head over to python.org/downloads/[5] and follow the instructions to get Python.

1.4. Tools

Java	**Python**
.java	.py
.class	.pyc
java.exe + javac.exe	python.exe
IntelliJ IDEA	PyCharm
Eclipse JDT	PyDev
Beanshell (?)	REPL

Python source code is organized in files with a `.py` extension. The `python` executable interprets the source code and translates it into a Python language specific bytecode which is stored in `.pyc` files. This bytecode is then executed by the Python Virtual Machine which is also invoked by the same `python` executable. Although this sounds like two steps, in reality, it is just one step with the bytecode generation happening on the fly.

This is in contrast to Java, where the responsibilities for the parsing & compilation of source code and the actual execution of the compiled bytecode are split between `javac` and `java` respectively. In Python, the `python` executable handles both steps. In fact, `.pyc` files are in-effect just intermediate caches to hold the translated bytecode. They are not strictly necessary for execution.

There are multiple IDEs available for writing Python code. PyDev, based on the Eclipse framework and PyCharm, based on the IntelliJ IDEA framework are two of the more popular choices. While having an IDE is nice, it is perfectly feasible to write Python code using a plain text editor such as Vim[6] or Sublime Text.

One interesting feature of Python that's missing in Java is the REPL, short for *Read Eval Print Loop*. A quick demo would be useful here. If you've got access to a Python installation following the instructions in Section 1.3, "Installation", go ahead and launch a `python` shell:

[5] https://www.python.org/downloads/

[6] Yes, Emacs is fine too.

```
antrix@dungeon:~$ python
Python 2.7.5+ (default, Feb 27 2014, 19:39:55)
[GCC 4.8.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

When you run the `python` executable in this manner, it starts up in an *interactive* mode. The first few lines contain information such as the version and the underlying operating system. After that, you are presented with the `>>>` prompt. This is where all your interaction with Python will occur. The python shell is running a *loop* which will *read* everything that you type in at this prompt, *evaluate* what it has read and then *print* the result. Thus the name, *REPL*.

Let's try it out:

```
>>> 10 + 10
20
>>>
```

We typed in `10 + 10` at the prompt and hit the **Enter** key. The Python REPL read this value, evaluated it and printed out the result. Then it went back to the prompt waiting for our next input. Let's try some variable assignment:

```
>>> x = 10
>>>
```

In this case, we didn't see any output because what we entered was just a statement, not an expression. But it did modify the state of the python shell. If we query for `x` again, we'll find:

```
>>> x = 10
>>> x
10
>>>
```

Let us call one of the built-in functions named `help`:

```
>>> help(x)

Help on int object:
```

```
class int(object)
 |  int(x=0) -> int or long
 |  int(x, base=10) -> int or long
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 are given
:q

>>>
```

Calling `help` on any object brings up a paged view of what is, effectively, the *JavaDoc* for the object's class. To exit the help view, just type `q` at the `:` prompt and you'll be back at the `>>>` prompt.

The full documentation view for an object can be quite verbose. If you just want a quick overview of what attributes an object exposes an object supports, use the `dir` function:

```
>>> dir(x)

['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
 '__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
 '__floordiv__', '__format__', '__getattribute__', '__getnewargs__',
 '__hash__', '__hex__', '__index__', '__init__', '__int__', '__invert__',
 '__long__', '__lshift__', '__mod__', '__mul__', '__neg__', '__new__',
 '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__',
 '__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
 '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
 '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
 '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
 'bit_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']

>>>
```

Ignoring the funky double-underscores for now, what `dir(x)` returned is effectively a *directory* of all the attributes available on the object. You can access any of them using the `.` syntax:

```
>>> x.numerator
10
>>> x.denominator
1
>>> x.conjugate
```

```
<built-in method conjugate of int object at 0x9e9a274>
>>> x.conjugate()
10
```

You can also use the `dir()` function without any argument to get a list of *built-ins*:

```
>>> dir()

['__builtins__', '__doc__', '__name__', '__package__']

>>> dir(__builtins__)

['ArithmeticError', 'AssertionError', 'AttributeError',
 'BaseException', 'BufferError', 'BytesWarning', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
 'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError',
 'OverflowError', 'PendingDeprecationWarning', 'ReferenceError',
 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring',
 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
 'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
 'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',
 'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
 'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
 'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']

>>>
```

This gives a list of functions and other objects that are *built-in* and do not have to be imported from other packages. This is analogous to how everything defined in the `java.lang` package is available everywhere in Java without having to explicitly import it.

> The dir and help functions are extremely useful when doing exploratory development in a Python interactive shell.

There's one last thing I wish to show before we wrap up this section. Let's create a new file named hello.py with the following contents:

```
print "Hello There!"
x = 10 + 10
print "The value of x is", x
```

Now execute python passing in this file as an argument:

```
antrix@dungeon:~$ python hello.py
Hello There!
The value of x is 20
antrix@dungeon:~$
```

This is more along the lines of a traditional development process: write code in a file and then execute that file. This also demonstrates how the python executable combines the role of javac and java in one process.

With that brief demo of Python, we are ready to dive into the nitty-gritty details of the language's syntax!

Chapter 2. The Syntax

2.1. Hello World

```
>>> print "Hello World"
Hello World
>>>
```

Well, that was easy! Moving on …

2.2. Basic constructs

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]          ❶
>>> odd_numbers = []                               ❷
>>>
>>> # What are the odds?                           ❸
>>> for num in numbers:                            ❹
...   if num % 2 != 0:                             ❺
...     odd_numbers.append(num)                    ❻
...
>>> print "the odd numbers are:", odd_numbers      ❼
the odd numbers are: [1, 3, 5, 7, 9]
>>>
```

Here's a bit of Python code that .. well, I don't have to tell you what it does, do I? Most Python code is like this, eminently readable and almost pseudo-code like. But you must have noticed a few things like the lack of semicolons as statement separators. Let's work through this code one line at a time and see what else is new and different compared to Java.

❶ In the first line, we are declaring a *list* of `numbers`. Lists are one of the built-in data structures in Python, along with *tuples*, *dicts* and *sets*. Notice that we didn't declare any types nor did we use any `new` keyword to allocate the list.

❷ Next, we declare another list, named `odd_numbers` which is initialized empty.

❸ Moving further down, we find a *comment* starting with the `#` token. Comments extend to the end of line, just like they do in Java with the `//` token.

❹ Here, we come upon a `for` loop which should remind you of a more *English* version of Java's *foreach* loop. Block scopes, like in this `for` loop or the `if` conditional in the next line, are denoted using indentation instead of curly (`{..}`) braces. Using just whitespace indentation to define blocks may sound weird and even prone to errors! But just give it a chance and you'll find it to quickly become second nature.

❺ On this line is an `if` statement that is quite similar to Java, except for the lack of paranthesis. Parantheses around `for` and `if` conditional expressions are optional. Include them only when they add clarity. Apart from the `for` and `if` constructs shown here, Python also has `elif`, `while`, etc.

❻ Here, we append the current loop number to the `odd_numbers` list. The list, just like almost everything in Python, is an object that supports several operations on it, including the `append` operation.

❼ Finally, we `print` the results to console. No more typing of the decidedly more verbose `System.out.println`!

> ⚠ *A word of caution:* **never mix tabs and whitespaces in Python source code**. While you can use either tabs or whitespace to denote indentation, mixing the two in the same source file may lead to parsing errors. My recommendation: just don't use tabs and stick to spaces. Set your text editor to insert 4 space characters per tab.

2.3. Basic Types

Some of the basic data types in Python are *numbers*, *strings* and *collections*.

2.3.1. Numbers

Numbers come in the following variety:

Type	Example Value
`int`	`1000`
`long`	`1000L`
`float`	`1000.12`
`complex`	`1000 + 12j`

Although `int` and `long` are different data types, in practice you only need to worry about them when declaring literal values, i.e. literal longs need to be declared with a `L` suffix. During arithmetic operations, Python automatically converts `int` values to `long` values as needed. This also prevents overflow related bugs.

> ℹ In Python 3, there's no distinction between `int` and `long`; there's only one arbitrary length integer type.

2.3.2. Strings

Like in Java, strings are immutable in Python. String values can be wrapped in either single or double quotes. To differentiate between vanilla ASCII strings and Unicode strings, Python uses the `u` prefix to denote the latter. Unicode strings provide additional operations related to encoding/decoding from various character sets.

Type	Example Value
`str`	`'apple'`
`unicode`	`u’äþþĺė'`
`str`	`r’C:\temp'`

A third type of string is the *raw string* denoted by the `r` prefix. This is just an indication to the Python parser to not apply any backslash escaping rules to the string. Here’s a quick example that illustrates the difference.

```
>>> print 'c:\temp\dir'    ❶
c:      emp\dir
>>> print 'c:\\temp\dir'   ❷
c:\temp\dir
>>> print r'c:\temp\dir'   ❸
c:\temp\dir
>>>
```

❶ The `\t` is interpreted as the *tab* character, resulting in a tab being printed.

❷ Escaping the `\t` with an additional backslash helps, but makes the string harder to read.

❸ Now the `\t` is left as-is since the `r` prefix is used to mark the string as a *raw* string.

As you can imagine, this is extremely useful when denoting file system paths or regular expressions.

ℹ In Python 3, all strings are `unicode` by default. Plain strings are just dumb bytes.

2.3.3. Collections

The built-in Python collections come in four varieties:

Type	Java Equivalent	Example Value
`list`	`java.util.ArrayList`	`['apple', 'ball', 'ball']`
`tuple`	`java.util.ArrayList`	`('apple', 'ball', 'ball')`
`dict`	`java.util.HashMap`	`{'fruit': 'apple', 'toy': 'ball'}`
`set`	`java.util.HashSet`	`{'apple', 'ball'}`

Each of these collection types provide several useful operations such as sort, subsequence, etc. Another key property is that all these data types are heterogeneous and can host values of differing data types. Think `Collection<Object>` and not `Collection<T>`.

> While `tuple` and `list` may look similar, the distinction is that a `tuple` is immutable.

Having these basic collections built into the language syntax is immensely useful. It makes a lot of day-to-day code quite succinct without the overhead of importing collection APIs and their associated baggage.

2.4. Fun with Lists

Lists are the workhorse data structure in Python and I exaggerate only slightly when I say that mastering them is the key to mastering Python! Although earlier I said that they are like `java.util.ArrayList`, they are truly much more than that. But first, let's look at a short example demonstrating their use as a basic array:

```
>>> numbers = [0, 1, 2, 'three', 4, 5, 6, 7, 8, 9]  ❶

>>> numbers[0]                                       ❷
0

>>> numbers[-1]                                      ❸
9
```

❶ The first thing to notice is that the `numbers` list is not homogeneous and can host values of different types, be it numbers, strings, other objects or even other lists!

❷ Individual element access is using the well-known array index syntax: `L[index]`.

❸ Python allows passing in a negative value for the index in which case, it adds the length of the list to the index and returns the corresponding element.

Apart from single element access, what sets apart Python lists is the ability to extract element ranges from lists. This is accomplished using the **slice syntax**. Here’s how:

```
>>> numbers[0:4]                ❶
[0, 1, 2, 'three']

>>> numbers[:4]                 ❷
[0, 1, 2, 'three']

>>> numbers[4:]                 ❸
[4, 5, 6, 7, 8, 9]

>>> numbers[2:-2]               ❹
[2, 'three', 4, 5, 6, 7]

>>> numbers[0:9:2]              ❺
[0, 2, 4, 6, 8]

>>> numbers[::2]                ❻
[0, 2, 4, 6, 8]

>>> numbers[::-1]               ❼
[9, 8, 7, 6, 5, 4, 'three', 2, 1, 0]
```

❶ The slice syntax – `L[start:stop]` – returns a subsequence of the list as a new list.

❷ The `start` index is optional and when omitted, defaults to `0`.

❸ The `stop` index is optional too and defaults to `len(L)`.

❹ A negative `stop` index counts off from the end, in accordance with the rule described above for line 3 of this code example.

❺ The full slice syntax is actually `L[start:stop:step]` where the `step`, when omitted, defaults to `1`. Here, we set it to `2` and it skips every other element of the list.

❻ Another example showing default values for `start` and `stop`.

❼ A negative `step` reverses the direction of iteration.

The copy returned by the slice notation is a shallow copy and we can demonstrate this with a quick example:

```
>>> copy = numbers[:]           ❶

>>> copy == numbers             ❷
```

```
True

>>> copy is numbers          ❸
False

>>> id(copy), id(numbers)    ❹
(3065471404L, 3075271788L)
```

❶ Creates a shallow copy

❷ The two lists are logically equal. This is similar to comparison using `equals()` in Java.

❸ But the two lists are not the same object. This is similar to comparison using `==` in Java.

❹ We can confirm by checking the object references using the `id()` built-in function. This is similar to the default `hashCode()` in Java.

Now let us turn our eye towards performing operations on lists. The first thing one would want to do with a list is to iterate over its elements. There are a few different variants of list iteration in Python, either using the vanilla *foreach* style syntax or augmenting it with the `enumerate`, `range` and `len` built-in functions.

```
>>> numbers = [10, 20, 30]
>>> for number in numbers:
...   print number
...
10
20
30

>>> for index, number in enumerate(numbers):
...   print index, number
...
0 10
1 20
2 30

>>> for  index in range(len(numbers)):
...   print index
...
0
1
2
```

Next, let us see how we can mutate lists.

```
>>> toys = ['bat', 'ball', 'truck']
>>> if 'bat' in toys:
...   print 'Found bat!'
...
Found bat!

>>> toys.append('doll')
>>> print toys
['bat', 'ball', 'truck', 'doll']

>>> toys.remove('ball')
>>> print toys
['bat', 'truck', 'doll']

>>> toys.sort()
>>> print toys
['bat', 'doll', 'truck']
```

Lists can also be used as simple stacks and queues.

```
>>> stack = []
>>> stack.append("event")  # Push
>>> event = stack.pop()    # Pop
>>>
>>> queue = []
>>> queue.append("event")  # Push
>>> event = queue.pop(0)   # Pop from beginning
```

There are many more operations that a list provides, such as `extend`, `insert` and `reverse`. But let us now look at one of the most interesting features of lists: **Comprehensions**.

Consider the following code which computes the factorial of the first few integers:

```
>>> import math

>>> numbers = range(5)
>>> numbers
[0, 1, 2, 3, 4]

>>> factorials = []

>>> for num in numbers:
...   factorials.append(math.factorial(num))
...
>>> factorials
```

```
[1, 1, 2, 6, 24]
```

The above procedural loop can be replaced by a *functional* one-liner using the built-in `map` function as follows:

```
>>> factorials = map(math.factorial, range(5))
```

Python defines list comprehensions using the syntax: `new_list = [function(item) for item in L]`. We can re-write the factorial loop using this syntax as follows:

```
>>> factorials = [math.factorial(num) for num in range(5)]
```

> List comprehensions are one of the most *Pythonic* language features. Anytime that you see or think of a `map(fn, iter)`, it can be better expressed as `[fn(x) for x in iter]`.

Here's another variant that introduces a conditional in the comprehension:

```
>>> factorials_of_odds = [math.factorial(num) for num in range(10) if num
  % 2 != 0]
```

If the list/object being iterated over is large (or even unbounded), then a variant of the list comprehension syntax called *generator expressions* can be used. In this snippet, `factorials_of_odds` is lazily computed as you iterate over it.

```
>>> factorials_of_odds = (math.factorial(num) for num in xrange(10**10) if
  num % 2 != 0)
```

Syntactically, the only difference between list comprehensions and generator expressions is that while the former are enclosed in square brackets, the latter are enclosed in round brackets.

Aside

In the *generator expressions* example, I used a function `xrange(10**10)`. The `**` is the exponent operator, i.e. `10**10` is `10000000000`. The usual `range` function, when called with `10**10` as an argument, would have to allocate and keep in memory a ten billion element list. Instead of pre-allocating such a big list, `xrange` just returns an iterator which, only when iterated over, produces elements upto ten billion one at a time.

With that rather verbose introduction to *lists*, let us turn our attention towards another building block of procedural programming: **Functions**.

2.5. Functions

Functions in Python are quite a bit more flexible than Java.

- They are first class objects that can live by themselves without the need to be wrapped inside classes. They can be created at runtime, assigned to variables, passed as arguments to other functions and returned as values from other functions.
- In addition to a simple list of positional parameters, Python functions also support named parameters, varargs and keyword based varargs.
- Python also supports anonymous functions in the form of *lambda expressions*, a feature added to Java in 2014 as part of the Java SE 8 release.

Here's what a function definition in Python looks like:

```
def a_function(arg1, arg2="default", *args, **kwargs):
    """This is a short piece of documentation for this function.
       It can span multiple lines.
    """
    print "arg1:", arg1      # arg1 is a mandatory parameter
    print "arg2:", arg2      # arg2 is an optional parameter with a default
 value
    print "args:", args      # args is a tuple of positional parameters
    print "kwargs:", kwargs # kwargs is a dictionary of keyword parameters
```

Functions definitions begin with the `def` keyword (Hello Scala & Groovy!) followed by the parameter list in parentheses. Once again, there are no curly braces and just the indentation defines the scope of the function body.

For now, please ignore the strange looking asterisk prefix in front of `args` and `kwargs` in this function's parameter declaration. It is a special bit of syntax that I'll describe in the next section.

The documentation within triple quotes is called a *docstring*, similar to Javadoc. Calling `help(a_function)` will display this docstring.

```
>>> help(a_function)
Help on function a_function in module __main__:

a_function(arg1, arg2='default', *args, **kwargs)
    This is a short piece of documentation for this function.
    It can span multiple lines.
(END)
```

We don’t declare the types of the parameters, relying instead on *duck typing* i.e. as long as the parameter argument has the attributes our function expects to operate upon, we don’t care about its real type.

Aside

Wikipedia has a nice, concise explanation of *Duck typing:*

> Duck typing is a style of typing in which an object’s methods and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of an explicit interface. The name of the concept refers to the duck test, attributed to James Whitcomb Riley, which may be phrased as follows:
>
> > When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
>
> In duck typing, a programmer is only concerned with ensuring that objects behave as demanded of them in a given context, rather than ensuring that they are of a specific type. For example, in a non-duck-typed language, one would create a function that requires that the object passed into it be of type Duck, in order to ensure that that function can then use the object’s walk and quack methods. In a duck-typed language, the function would take an object of any type and simply call its walk and quack methods, producing a run-time error if they are not defined.

Let’s see how `a_function` as defined earlier behaves when called with different argument values.

```
>>> a_function(10)                    ❶
```

```
arg1: 10
arg2: default
args: ()
kwargs: {}

>>> a_function(10, "ten")          ❷
arg1: 10
arg2: ten
args: ()
kwargs: {}

>>> a_function(10, 20, 30, 40)     ❸
arg1: 10
arg2: 20
args: (30, 40)
kwargs: {}

>>> a_function(10, "twenty", arg3=30, arg4="forty")    ❹
arg1: 10
arg2: twenty
args: ()
kwargs: {'arg3': 30, 'arg4': 'forty'}

>>> a_function(arg2="twenty", arg1=10, arg3=30, arg4="forty")   ❺
arg1: 10
arg2: twenty
args: ()
kwargs: {'arg3': 30, 'arg4': 'forty'}
```

❶ Only `arg1` is provided, the other parameters are initialized to default values.

❷ The positional arguments are provided.

❸ This is like Java's varargs. All the positional arguments that aren't explicitly declared in the parameter list are populated in the `args` tuple.

❹ This demonstrates usage of keyword or named arguments.

❺ The order isn't important when the parameter names are made explicit.

2.5.1. Functions and Tuples

Python functions hold one more trick up their sleeve: support for multiple return values!

```
def multi_return():
    # These are automatically wrapped up
    # and returned in one tuple
    return 10, 20, 'thirty'
```

```
>>> values = multi_return()
>>> print values
(10, 20, 'thirty')
```

When a function returns multiple comma separated values, Python automatically wraps them up into a tuple datastructure and returns that tuple to the caller. This is a feature called *automatic tuple packing*. You may make this packing more explicit by wrapping up your return values in a tuple yourself but this is neither required, nor encouraged.

The real interesting part comes when this feature is combined with its counterpart, *automatic tuple unpacking*. Here's how it works:

```
>>> numbers = (1, 2, 3)        ❶
>>> print numbers
(1, 2, 3)

>>> a, b, c = (1, 2, 3)        ❷
>>> print a, b, c
1 2 3

>>> a, b, c = multi_return()   ❸
>>> print a, b, c
10 20 thirty
```

❶ Here, `numbers` is just a regular tuple.
❷ The tuple on the right of the assignment got *unpacked* into the variables on the left.
❸ The tuple returned from `multi_return` got *unpacked* into the variables on the left.

What happened here is that first, Python packed the multiple return values from `multi_return` into a single tuple. Then, it transparently unpacked the returned tuple and assigned the contained values to the corresponding variables on the left of the assignment.

For this to work, the number of variables on the left must match the number of elements being returned by the called function, otherwise an error is raised:

```
>>> a, b = multi_return()
ValueError: too many values to unpack
```

Now that you know how tuple packing and unpacking works, let's revisit the strange looking asterisks in `*args` and `**kwargs` that we encountered in the previous section. The leading single asterisk is Python notation to *unpack* the tuple values while the leading double asterisk unpacks the dict values. Here's an example that demonstrates this:

```
def ternary(a, b, c):
    print a, b, c
```

```
>>> ternary(1, 2, 3)
1 2 3

>>> args = (1, 2, 3)
>>> ternary(args)
TypeError: ternary() takes exactly 3 arguments (1 given)

>>> ternary(*args)  # Unpacks the args tuple before function call
1 2 3

>>> kwargs = {'a': 1, 'b': 2, 'c': 3}
>>> ternary(kwargs)
TypeError: ternary() takes exactly 3 arguments (1 given)

>>> ternary(**kwargs) # unpacks the dictionary before function call
1 2 3
```

2.5.2. Functions inside Functions

Now that you are familiar with the basic function definition syntax, let us look at a more advanced example. Consider the following function:

```
def make_function(parity):                                              ❶
    """Returns a function that filters out `odd` or `even`
       numbers depending on the provided `parity`.
    """
    if parity is 'even':
        matches_parity = lambda x: x % 2 == 0                           ❷
    elif parity is 'odd':
        matches_parity = lambda x: x % 2 != 0
    else:
        raise AttributeError("Unknown Parity: " + parity)               ❸

    def get_by_parity(numbers):                                         ❹
        filtered = [num for num in numbers if matches_parity(num)]
        return filtered

    return get_by_parity                                                ❺
#
```

There's a lot to digest here! Let's take it line by line.

❶ Here, we begin defining a function named make_function, starting with the docstring.

❷ Next, we use the the lambda keyword to define a one line, anonymous function which we assign to matches_parity. The lambda function assigned to matches_parity depends on the value of the parity function argument.

❸ If the parameter argument value is neither odd nor even, we raise the built-in AttributeError exception.

❹ We now define a get_by_parity function within the enclosing function's body. You'll notice that the value of matches_parity is used here. This is a *closure*. It is similar to capturing final fields from enclosing scopes inside anonymous class declarations in Java. In fact, the lambda functionality in Java 8 is much closer to this Python feature than Java anonymous classes.

❺ Finally, we return the get_by_parity function object from make_function.

Functions in Python are first-class objects of type function. They can be passed around and assigned to variables just like any other object. In this case, when someone calls make_function, it returns another function who's definition depends on the parameter passed to make_function. Let's see how this works with a quick example.

```
>>> get_odds = make_function('odd')      ❶
>>> print get_odds(range(10))            ❷
[1, 3, 5, 7, 9]

>>> get_evens = make_function('even')    ❸
>>> print get_evens(range(10))
[0, 2, 4, 6, 8]
```

❶ We called make_function with odd as the parity parameter value and it returns to us a function which we assign to the get_odds variable.

❷ Now, for all practical purposes, get_odds is just another function. We invoke it by passing in a list of numbers (range(10) returns a list of 0..10) and out comes a filtered list of odd numbers.

❸ We can repeat this exercise for the even parity and verify that make_function is working as expected.

> *"Functions as first-class objects"* is a powerful idea to digest and necessitates a change in how you structure your programs. Coming from a Java background to Python, you must learn to resist the urge to model

> everything as a class. After all, not everything is a noun and some things are best described using verbs![1]
>
> A lot can be accomplished using just functions and Python's built-in data structures like lists and dicts. In doing so, you'll find that more often than not, your programs turn out to be simpler and easier to understand.

2.6. Classes

Everything in Python is an Object and as you'd expect, the way to create objects is to start from Classes. Consider the following definition of a simple `Person` class.

```
class Person(object):
    def __init__(self, first, last):
        self.first = first
        self.last = last

    def full_name(self):
        return "%s %s" % (self.first, self.last)

    def __str__(self):
        return "Person: " + self.full_name()
```

As in Java, `object` is at the root of the class hierarchy but unlike Java, it needs to be specified explicitly in Python (although not in Python 3).[2] Inheritance declarations do not use a special keyword like `extends`. Instead, the parent class' name is enclosed within paranthesis after the declaring class' name.

The `__init__` method is the *initializer* method and is analogous to the Java class constructor. There is also a *constructor* method called `__new__` but you won't use it unless you are doing metaprogramming like writing factory classes, etc. Within `__init__`, all the instance fields — called *attributes* in Python — are initialized. Note that we did not have to pre-declare all the attributes of the class.

The first argument of all instance methods is `self`. It is the `this` reference that is implicit in Java but made explicit in Python. Note that the literal name `self` isn't mandatory; you could name it whatever you want. If you named it `current`, then `full_name` would be defined as:

[1] See steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html [http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html]

[2] You may skip specifying `object` as the base class in Python 2, but it'll have implications as explained in New-style Classes [https://www.python.org/doc/newstyle/].

```
# `current` instead of the conventional `self`
def full_name(current):
    return "%s %s" % (current.first, current.last)
```

Aside

I've sneaked in an example of string interpolation in the `full_name` method definition. Python's string interpolation works on tuple arguments and is similar to Java's `String.format(s, arg…)`. There's another variant which works on named parameters and takes a dictionary argument:

```
>>> "The name is %(last)s, %(first)s %(last)s" % {'first': 'James',
 'last': 'Bond'}
'The name is Bond, James Bond'
```

The double underscore notation used in the `__init__` method name is a Python convention for declaring *special* methods. The `__str__` is another special method. Its behaviour is exactly like that of the `toString()` method in Java. I'll explain what makes these methods special when we talk about Section 2.7, "Protocols".

Now here is some example usage of this `Person` class.

```
>>> person = Person('Clark', 'Kent')    ❶

>>> print person                        ❷
Person: Clark Kent

>>> print person.first                  ❸
Clark

>>> print person.full_name()            ❹
Clark Kent

>>> print Person.full_name(person)      ❺
Clark Kent
```

❶ Object creation is just like in Java, except that you don't need to use the `new` keyword.

❷ `print` is equivalent to `System.out.println()` and it'll call the argument's `__str__` method just like the latter calls to `toString()`.

❸ The fields of the class, called *attributes* in Python, are accessed using the dotted syntax.

❹ Methods are accessed using the dotted syntax too. Although `self` is explicit during the method definition, it is implicitly passed when the method is called on an object.

❺ But you can make it explicit by calling the method from the class and passing in an instance. Can’t do *that* in Java!

> In Python 3, the `print` keyword has been replaced with a `print()` function.[3]

You’ll notice that we didn’t declare whether our fields are private or public. We just accessed them as if they are public. In fact, Python does not have the concept of visibility at all! Everything is just public. If you wish to indicate to the user of your class that a particular attribute or method is an internal implementation detail, then the *convention* is to prefix the attribute/method name with a single underscore and the person using the code will know to tread carefully.

2.6.1. Inheritance

Python supports single inheritance as well as multiple inheritance, i.e. the inheritance model is closer to C++ than Java. With multiple inheritance, there’s always the question of how methods are resolved when declared at multiple places in the class hierarchy. In Python, the *method resolution order* is in general, depth-first. The class attribute `__mro__` can be inspected to check the actual method resolution order being used for the class.

Here’s a `SuperHero` class that extends the `Person` class that we previously defined. We’ve added one new attribute, `nick` and one new method, `nick_name` in the `SuperHero` class.

```
class SuperHero(Person):
    def __init__(self, first, last, nick):
        super(SuperHero, self).__init__(first, last)
        self.nick = nick

    def nick_name(self):
        return "I am %s" % self.nick
```

`super` works like in Java, but once again, you need to be explicit about the class at which it should start climbing up. Let’s see how `SuperHero` behaves with a few examples.

```
>>> p = SuperHero("Clark", "Kent", "Superman")
```

[3] https://docs.python.org/3.0/whatsnew/3.0.html#print-is-a-function

```
>>> p.nick_name()
I am Superman

>>> p.full_name()
'Clark Kent'

>>> type(p)                              ❶
<class '__main__.SuperHero'>

>>> type(p) is SuperHero
True

>>> type(type(p))
<type 'type'>

>>> isinstance(p, SuperHero)             ❷
True

>>> isinstance(p, Person)
True

>>> issubclass(p.__class__, Person)      ❸
True
```

❶ The built-in `type()` function gives the type of any object. The *type* of a `class` object is `type`. The `__main__` that you see in the class name here is just the default namespace in which Python places your objects. You'll learn more about namespaces in Section 2.8, "Organizing Code".

❷ The `isinstance()` built-in function is the Python counterpart of Java's `instanceof` operator and the `Class.isInstance()` method.

❸ Similarly, the `obj.__class__` attribute is like Java's `obj.class` field.

2.6.2. Polymorphism

Let us look at the canonical example that is used to demonstrate polymorphic behaviour, the *Shape*.

```
class Square(object):
    def draw(self, canvas):
        ...

class Circle(object):
    def draw(self, canvas):
        ...
```

Given these two Square and Circle classes, the Java developer inside you would already be thinking of extracting a Shape class or interface that defines the draw(canvas) method. *Resist that urge!* Since Python is dynamic, the following code works just fine without an explicit Shape class:

```
shapes = [Square(), Circle()]
for shape in shapes:
    shape.draw(canvas)
```

There is no real advantage to having a common Shape base class that defines draw(canvas) since there's no static type check to enforce that anyway. If the objects in the shapes list did not implement draw(canvas), you'll find that out at runtime. In short, use inheritance for shared behaviour, not for polymorphism.

2.6.3. Getting Dynamic!

So far, what we've seen of classes in Python is pretty tame. Nothing that you couldn't accomplish in Java. Time to make it interesting! Consider the following:

```
>>> p = SuperHero("Clark", "Kent", "Superman")        ❶

>>> def get_last_first(self):                         ❷
...   return "%s, %s" % (self.last, self.first)
...
>>> Person.last_first = get_last_first                ❸

>>> print p.last_first()                              ❹
Kent, Clark
```

❶ We start with an instance of the SuperHero class.

❷ Next, we define a new top-level function named get_last_first().

❸ Then we assign the reference of the get_last_first() function to a new attribute named last_first of the Person class.

❹ Thanks to the previous step, all instances of the Person class, including instances of derived classes, have now sprout a new method.

To summarize, what we've done here is bound a new function as an instance method to the Person class. Once bound, the method becomes available to all instances of the Person class, including those already created!

This technique can also be used to define a new implementation for an existing method. Doing so is usually called *monkey patching* and is generally frowned upon in the Python community since it can quite easily cause surprising & unexpected behaviour.

Now that we've seen how we can add behaviour to a class after the fact, can we go the other way and *remove* behaviour? Sure!

```
>>> print p.last
Kent
>>> del p.last
>>> print p.last
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  AttributeError: 'SuperHero' object has no attribute 'last'

>>> del Person.full_name
>>> p.full_name()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  AttributeError: 'SuperHero' object has no attribute 'full_name'

>>>
```

Because Python is dynamically typed, accessing non-existent fields and methods causes exceptions at runtime. Conversely, we can define new attributes at runtime!

```
class Person(object):
   ...

   def __getattr__(self, item):
       # This special method is called when normal attribute lookup fails
       if item is 'hyphenated_name':
           return lambda x: "%s-%s" % (x.first, x.last)
       else raise AttributeError(item)

>>> p = Person('Clark', 'Kent')
>>> p.hyphenated_name()
'Clark-Kent'
```

Imagine the amount of bytecode rewriting trickery you would have to do to achieve this same effect in Java! Case in point, think of the code gymnastics that Java mock libraries are forced to go through. In Python, mocks are trivial to implement using `__getattr__` and `__setattr__`.

2.7. Protocols

Protocols are like Java interfaces in that they define one or more methods that offer a particular behaviour. However, unlike Java interfaces, protocols are not explicitly defined in source code. The closest equivalent would be the `equals()`, `hashcode()` and `toString()` methods in Java. These methods aren't part of any explicit interface. Yet, we have an implicit convention[4] that these methods will be invoked in certain situations. So it is with Python Protocols.

The Python language defines several different protocols such as sequences, numeric, containers, etc. These protocols manifest themselves in terms of special syntactic support in the language grammar. Pretty much every language syntax in Python is implemented in terms of protocols and thus, can be made to work with your own types by implementing the relevant protocols.

Let me explain this further using one of the protocols, the *Container* protocol, as an example. Consider the following `OrderRepository` class definition which provides access to a database backed collection of `Order` objects.

> ⚠ Please do not use this example as the basis of production code! It is wide open to SQL Injection attacks. For relational database access, consider using the SQLAlchemy library discussed in The Ecosystem.

```
class OrderRepository(object):
    ...
    def __contains__(self, key):
        return 1 == db.query("select count(1) from Orders where id='%s'" %
 key)

    def __getitem__(self, key):
        return Order(db.query("select * from Orders where id='%s'" % key))

    def __setitem__(self, key, value):
        d = value.as_dict()
        update_params = ", ".join( ["%s = '%s'" % x for x in
 d.iteritems()] )
        db.update("update Orders set %s where id='%s'" % (update_params,
 key)
```

[4] A convention documented in the Java Language Specification, but a convention nevertheless; not a source code level contract.

I've elided the full class definition and only shown the three methods which are part of the Container protocol. Since `OrderRepository` can now be said to implement the Container protocol, it allows us to use it in the following way:

```
>>> orders = OrderRepository(db)
>>> if "orderId123" in orders:          ❶
...     order = orders["orderId123"]    ❷
...     order.status = "shipped"
...     orders["orderId123"] = order    ❸
>>>
```

❶ Because we've implemented the `__contains__` method for the `OrderRepository`, we can now use the `if x in y` syntax to operate on it. What's happening under the covers is that Python is translating that `if` statement into `if orders.__contains__("orderId123"):`

❷ Similarly, the `__getitem__` method unlocks the dictionary like access using the order id, translating the key lookup to `orders.__getitem__("orderId123")`.

❸ Finally, dictionary-like assignment works via the `__setitem__` method call.

You can think of this as operator overloading or syntactic sugar, whichever fits your mental model!

Here's a list of some of the other protocols that Python supports and the syntax that they power.

Protocol	**Methods**	**Supports Syntax**
Sequences	Support `slice` in `__getitem__`, etc.	`seq[1:2]`
Iterators	`__iter__`, `next`	`for x in collection:`
Comparison	`__eq__`, `__gt__`, `__lt__`, ...	`x == y`, `x > y`, `x < y`, ...
Numeric	`__add__`, `__sub__`, `__and__`, ...	`x + y`, `x - y`, `x & y`, ...
String like	`__str__`, `__unicode__`, `__repr__`	`print x`
Attribute access	`__getattr__`, `__setattr__`	`obj.attr`
Context Managers	`__enter__`, `__exit__`	`with open('out.txt') as f: f.read()`

2.8. Organizing Code

The most basic unit of source code organization in Python is the **module** which is just a `.py` file with Python code in it. This code could be functions, classes, statements or any combination thereof. Several modules can be collected together into a **package**. Python packages are just like Java packages with one difference: the directory corresponding to a package *must* contain an *initializer* file named `__init__.py`. This file can either be empty or can optionally contain some bootstrap code that is executed when the package is first imported.

Suppose we have a code base organized as follows:

```
.
|-- cart.py
|-- db
|   |-- __init__.py
|   |-- mysql.py
|   +-- postgresql.py
+-- model
    |-- __init__.py
    +-- order.py
```

Given this directory listing, we can see that there are two *packages*: `db` and `model`; and four *modules*: `cart`, `mysql`, `postgresql` and `order`.

2.8.1. Importing code

Importing code defined in one file (or *module*, in Python terms) into another is accomplished using the `import` statement. The `import` statement works pretty much like it does in Java: it brings the declarations from the target module into the current namespace.

There are two syntactical variants of the `import` statement. The first one is in the familiar Java style, `import` … while the second one follows a `from` … `import` … pattern.

Suppose we have a class named `SellOrder` defined in the `order` module, i.e. inside the `order.py` file:

```
$ cat model/order.py

class SellOrder(object):
    ...
    ...
```

There are a few different ways in which we can *import* and use this class in our main app, `cart.py`.

```
import model

sell_order = model.order.SellOrder()
```

In this example, we use the `import <package|module>` syntax to import the target *package* — `model` — into the current namespace and then use a dotted notation to get to our `SellOrder` class. We can use the same syntax to import the specific *module* instead of the containing *package*:

```
import model.order

sell_order = order.SellOrder()
```

Here, we imported the `order` *module* directly. Note the distinction between the way the `import` syntax works in Java and in Python. In Java, we always import a *class* from within our package hierarchy. In Python, an `import …` statement can **only** be used to import *packages* or *modules*. If you want to access a class or function definition, you must refer to it via the containing module. Or use the alternate syntax, `from <package|module> import <item>`:

```
from model.order import SellOrder

sell_order = SellOrder()
```

Here, we use the `from <package|module> import <item>` syntax variant which directly imports `SellOrder` into the current namespace. This style of `import` can be used to import any top-level item from within the source module, be it a function, class or variable definition.

Python offers one more enhancement over Java imports: the ability to rename the import using the `as` keyword.

```
from model.order import TYPES as ORDER_TYPES
from db import TYPES as DATABASE_TYPES

print ORDER_TYPES
# ['buy', 'sell']
```

```
print DATABASE_TYPES
# ['mysql', 'postgresql']
```

As you can imagine, this feature comes in very handy when trying to avoid namespace conflicts without having to use the full package/module hierarchy for disambiguation like we do in Java.

> In Java, each `.java` file must contain only class or interface declarations at the top level. Moreover, in typical usage, each such file has just one public class or interface defined. Python modules have no such restrictions. It is perfectly fine to create modules with just functions or just classes or a mix of both. Do not restrict yourself to just one class or function definition per module. This is not only unnecessary, but also considered a bad practice. Instead, strive to collect constructs having conceptual similarity into one module.

2.8.2. The main() method

Now that we've organized our application's source code into multiple files, you must be wondering, what defines the entry point for the application? As Java programmers, `public static void main` is forever burnt into our brains! What's the equivalent in Python?

There is no formal notion of an *application entry point* in Python. Instead, when Python executes a file, e.g., when you run `python foo.py`, execution begins at the start of the file and all statements defined at the top-level are executed in order till we reach the end of the file. Consider a file named `odds.py` which contains the following code:

```
# odds.py

def get_odds(numbers):
    odds = [n for n in numbers if n % 2 != 0]
    return odds

odds_until = 10

numbers = range(odds_until)

print get_odds(numbers)
```

When this file is executed by running `python odds.py` from your favourite operating system shell, the Python interpreter starts at the top, runs down the file till it finds the definition of the `get_odds` function. It makes note of this function by adding its name to the current namespace. It does so by making an entry in a lookup table that it maintains for each namespace. Once done with adding `get_odds` to the current namespace's lookup table, Python then skips the rest of the function declaration since statements inside the function's body aren't at the *top level*.

Moving further down the file, Python encounters the declaration of the `odds_until` variable and executes that statement causing the value `10` to be assigned to it. Once again, an entry is made in the current namespace's lookup table for the `odds_until` variable.

On the next line, it encounters an assignment statement which involves a function named `range`. Python looks up this function in the current namespace, where it can't find it. It then looks for it in the *built-in* namespace and finds it there. Recall that the *built-in* namespace is equivalent to `java.lang.*` – things defined here don't have to be explictly imported. Having found the `range` function, it calls it assigning the return value to `numbers`. As you can guess by now, another entry is made for `numbers` in the current namespace.

Proceeding further, we reach the last line of the file where there's a call to `get_odds` with `numbers` as a parameter. Since both these names have entries in the current namespace, Python has no trouble calling `get_odds` with the list of numbers. Only at this point in time is the `get_odds` function body parsed and executed. The return value is then supplied to `print` which writes it out to the console:

```
$ python odds.py
[1, 3, 5, 7, 9]
$
```

Having seen how Python executes a script, we can try and simulate a *main* method as follows:

```
# odds.py

def get_odds(numbers):
    odds = [n for n in numbers if n % 2 != 0]
    return odds

def main():
    odds_until = 10
    numbers = range(odds_until)
    print get_odds(numbers)

main()
```

All we've done here is wrapped up all our top-level statements into a function that we conveniently gave the name of main! We call this function at the end of the file, in effect, making main the entry point of our app.

Let's complete our Java-like main method implementation by taking care of the arguments to main, i.e. the args in public static void main(String[] args). In Java, all command-line parameters passed to the application during launch would be populated in the args arrays. In Python, this information is available using the built-in sys standard library module. This module defines sys.argv which is a list of the command line arguments passed to the Python script on startup. The first value in the list, sys.argv[0] is the name of the script itself. The remaining items in this list are the command line arguments, if any.

Let us modify our odds.py script to take the number until which we should print odd numbers as a command line parameter, which we retrieve using the sys module.

```
# odds.py

def get_odds(numbers):
    odds = [n for n in numbers if n % 2 != 0]
    return odds

def main(args):
    try:
        odds_until = int(args[1])
    except:
        print "Usage: %s <number>" % sys.argv[0]
        sys.exit(1)

    numbers = range(odds_until)
    print get_odds(numbers)

import sys
main(sys.argv)
```

In this modified odds.py, we invoke the main function with the list of command line arguments as a parameter. Within main, we initialize the odds_until variable using the first command line argument. If this fails for any reason, we print a helpful message on how to use the script before exiting with a 1 error code. Here's how this modified example works in practice:

```
$ python odds.py
Usage: odds.py <number>
```

```
$ python odds.py abc
Usage: odds.py <number>
$ python odds.py 15
[1, 3, 5, 7, 9, 11, 13]
$
```

Finally, we have a main function that works like Java! It even looks like the main method in Java; `def main(args)` is more or less identical to `public static void main(String[] args)` once all the type related declarations are dropped.

However, there's a wrinkle here that I wish to talk about. Imagine that we found our `get_odds` function to be so useful that we wanted to use it as a utility function elsewhere in our project's codebase. Since we just talked about modules, the obvious way would be to just use `odds.py` as a module and import the module wherever we find the use of this utility function. For example, in `demo.py`:

```
# demo.py

import odds

print odds.get_odds(range(10))
```

When we run this `demo.py` script, we expect it to import the `odds` module and then use the `get_odds` defined in there to print the odd numbers till ten. Instead, here's what happens:

```
$ python demo.py
Usage: demo.py <number>
$
```

That's odd. Why are we getting this message? We passed in `10` as an argument to `get_odds` in `demo.py`. Which other `<number>` is it expecting? In fact, even though the message says "Usage: **demo.py**", this usage message looks very much like the one we defined inside `odds.py`.

Here's what is actually happening. When Python imports a *module*, it actually *executes* that module just like it would have if the module was run as a script! During the execution of `demo.py`, when Python encounters the `import odds` statement, it first attempts to locate the `odds.py` file. Having found it, Python executes the entire file just like I described in our discussion earlier. Specifically, it executes the following piece of top-level code in `odds.py`:

```
import sys
main(sys.argv)
```

Since there was no command line parameter supplied during execution of `demo.py`, the value of `sys.argv[1]` is missing. This raises an exception within `main()` which causes the *Usage* message to get printed. Moreover, since the actual command executed in this case was `python demo.py`, the value of `sys.argv[0]` is `demo.py` and not `odds.py`. This explains the output message.

In order to be able to use `odds` as module, we would have to remove all top-level statements from it. In fact, this is a very important point!

> Unless you have a very good reason, **do not define any side-effect causing top-level statements in your modules**. If you do so, these statements will be executed whenever your module is imported causing all sorts of headaches.

Here's a modified `odds.py`, stripped of all top-level code:

```
# odds.py

def get_odds(numbers):
    odds = [n for n in numbers if n % 2 != 0]
    return odds
```

Now, running the `demo.py` script yields the expected output.

```
$ python demo.py
[1, 3, 5, 7, 9]
```

Having made this change, while we gained the ability to use `odds` as a module, we lost the ability to run it as a script. Wouldn't it be nice if we could do both? Even in Java land, just because a class defines a `main` method does not mean that it can't be imported and used as a vanilla class elsewhere!

To achieve this module/script duality, we'll have to dig a little bit deeper into the notion of *namespaces*. I've mentioned the term *namespace* several times in this chapter without defining it in more detail. In computer science terms, we understand namespaces as isolated contexts or containers for names. Namespaces allow us to group logical things together and allow reuse of names without causing conflicts.

During program execution, whenever a module is imported, Python creates a new namespace for it. The name of the module is used as the name of the namespace. Thus, all that a piece of code such as odds.get_odds(…) is doing is invoking the get_odds function in the odds namespace. When a namespace qualifier is left out, then the object is looked up in the current namespace or failing that, in the built-in namespace.

At runtime, you can get access to the namespace encapsulating your code by referring to the special __name__ variable. This variable is always bound to the current namespace. Let's see an example of __name__ in action by modifying our demo.py and odds.py scripts.

```
# odds.py

print "In odds, __name__ is", __name__

def get_odds(numbers):
    odds = [n for n in numbers if n % 2 != 0]
    return odds
```

```
# demo.py

import odds

print "In demo, __name__ is", __name__

print odds.get_odds(range(10))
```

Now when we run the demo script, we see this output:

```
$ python demo.py
In odds, __name__ is odds
In demo, __name__ is __main__
[1, 3, 5, 7, 9]
```

As we just discussed, on import, the odds module is bound to a namespace with the same name, i.e. odds. Hence, in the context of code within odds.py, the value of the __name__ variable is odds. However, for code in demo.py, we see that the value of __name__ is the curiously named __main__. This is a special namespace that Python assigns to the *main* context of your application. In other words, the entry point of your application, typically the script which is executed by Python, is assigned the namespace __main__.

We can put this knowledge to use to achieve the script/module duality. Here is our odds.py file once again, in a form that can be executed directly as a Python script, but can't be imported as a module.

```
# odds.py

def get_odds(numbers):
    odds = [n for n in numbers if n % 2 != 0]
    return odds

def main(args):
    try:
        odds_until = int(args[1])
    except:
        print "Usage: %s <number>" % sys.argv[0]
        sys.exit(1)

    numbers = range(odds_until)
    print get_odds(numbers)

import sys
main(sys.argv)
```

We will wrap up the top-level code behind a check for the current namspace:

```
# odds.py

def get_odds(numbers):
    odds = [n for n in numbers if n % 2 != 0]
    return odds

def main(args):
    try:
        odds_until = int(args[1])
    except:
        print "Usage: %s <number>" % sys.argv[0]
        sys.exit(1)

    numbers = range(odds_until)
    print get_odds(numbers)

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

As you can see, we took the side-effect causing top-level code, namely the invocation of the `main()` function, and put it behind a conditional that checks if the current namespace is `__main__`. When `odds.py` is run as a script, i.e. it is the entry point of the application, the value of `__name__` will be `__main__`. Thus, we will enter the conditional block and run that piece of code. On the other hand, when `odds.py` is imported as a module, the value of `__name__` will be `odds` and not `__main__`. Thus, the block of code behind the conditional is skipped.

As you read more and more Python code, you'll come across the `if __name__ == '__main__'` construct all the time. It is a standard idiom used in Python programs to get the effect of a *main* method. Python's creator, Guido van Rossum, has written a nice blog post on writing idiomatic Python main() functions[5] that takes this idea even further.

2.9. Syntax: Wrap Up

We covered quite a bit of ground in this chapter. We started with a look at the basic syntax and built-in types that Python provides. We then moved on to the building blocks of functions and classes. We then familiarized ourselves with Python's implicit interfaces, i.e. Protocols. Finally, we learned about organizing source code into modules and packages.

While this book ends our discussion of Python language syntax over here, there's more to Python than this! Some of the advanced topics that we didn't discuss include *decorators*, *properties*, *generators*, *context managers* and *I/O*. These are topics you'll find yourself exploring once you get comfortable writing basic Python code. To help you along in your learning, I've compiled some useful resources in References at the end of this book.

[5] http://www.artima.com/weblogs/viewpost.jsp?thread=4829

Chapter 3. The Ecosystem

3.1. A Rich Ecosystem

In the Java world, we know that Java is much more than the language itself. There's the JVM, the Java language runtime that allows deploying applications on a variety of hardware and operating system targets. There are the Java SE and EE Standard Libraries which provide a lot of useful functionality out of the box. And of course, there's an immensely rich variety of third-party libraries and frameworks to choose from. It is the strength of this ecosystem that makes Java a great platform to build upon.

So it is with Python!

Python too can be deployed on different hardware targets like x86, ARM & MIPS; on multiple operating systems like Linux, Mac OS X, Windows & Solaris. It can even be deployed on other software runtimes like the .NET CLR (c.f. IronPython[1]) or the Java Virtual Machine (c.f. Jython[2]).

Python has got a great standard library with lots of functionality delivered out of the box. *Batteries Included* is often used to describe the richness of the standard library.

Beyond the standard library, the 3rd party libraries & frameworks landscape is similarly rich, ranging from numerical and scientific computing packages, to NLP, to networking, GUI and web frameworks. You'd be hard pressed to find a domain for which there isn't already a Python library out there.

To help you get started, I've put together a couple of lists that'll help you pick out the Python counterparts of the Java tools and libraries that you are already familiar with. These lists are by no means exhaustive but I do believe that they represent the most popular options in use today.

3.2. Popular Tools

Java	Python
build.xml/pom.xml	requirements.txt
maven	pip
maven central repo	pypi

[1] http://ironpython.net/

[2] http://www.jython.org/

Java	Python
Classpath	PYTHONPATH
HotSpot	CPython
JRockit/IKVM/Dalvik	Jython/IronPython/PyPy

In the Python world, third-party library installation is typically done using a tool named `pip`. It can read the requisite third-party package dependencies for your project from a standardized `requirements.txt` file and install/upgrade packages as necessary. If needed, it can even download packages from a central repository named *pypi*, short for the *Python Package Index*. This is the Python world's equivalent of Maven's Central Repository.

With third-party packages comes the question of how they are located during application runtime. Java's solution is the *Classpath* and Python's equivalent is the *PYTHONPATH*. Conceptually, they are similar in that both specify a list of locations where the language runtime should search for packages being imported. Implemention wise, they differ a bit. *PYTHONPATH* is set as an environment variable, e.g. `PYTHONPATH=/path/to/foo:/path/to/bar`, a syntax similar to the typical system shell's `PATH` variable. Another flexibility that Python provides is that this library search path can be modified at runtime just by manipulating the `sys.path` attribute exposed by the built-in system module. As you might have guessed, `sys.path` is initialized from the value set for the `PYTHONPATH` environment variable.

I mentioned earlier that there are multiple runtime implementations of the Python language. The canonical implementation is CPython, named so because the core of the language runtime is implemented in C. It is the most widely used implementation and regarded as the reference for the Python language. Alternative implementations include Jython, which is Python running on the JVM; IronPython which is Python running on the .NET Common Language Runtime, and PyPy which is an upcoming implementation of Python that provides a number of improvements like a runtime JIT compiler, secure sandboxing and green-threads based concurrency.

As a Java developer, you may wish to explore Jython a bit more. Since it is essentially Python code translated to Java bytecode and running inside a Java Virtual Machine, it allows for easy interop between Python and Java. As an example, you could write high-level business rules in Python and call into a Java rule engine to execute those rules.

3.3. Popular Frameworks

Java	Python
JUnit	unittest/nose

Java	**Python**
Mockito	unittest.mock
findbugs/checkstyle	pylint
Javadoc	Sphinx
Swing	PyQT/PyGTK
Spring MVC	Django/Flask
Hibernate	SQLAlchemy
Velocity/Thymeleaf	Jinja2
Servlets	WSGI
Tomcat/Jetty	Apache/uWSGI

Since Python is a dynamic language, there's a class of programming errors that get revealed at runtime, instead of being caught at compile time like they would in a statically typed language. Thus, writing automated tests for your Python programs is even more important than it is in Java. The Python standard library ships with a unit testing framework creatively named **unittest**, that should serve as good start. If you find that *unittest* is missing some features you'd like or is just a bit too cumbersome to use, then **nose** is a good third-party alternative. In its own words, *nose* is *nicer testing for Python*.

Another tool in the modern developer's arsenal for keeping code quality high is automated static analysis of source code. The *findbugs* and *checkstyle* equivalent for Python is **pylint**. It can enforce a uniform code formatting style, detect basic programming errors and duplicate blocks of code. And as you would expect, you can integrate it in your IDE as well as CI build server.

If you are building a desktop GUI, then the **PyQT** and **PyGTK** libraries are popular options. These are Python wrappers around the popular Qt and GTK libraries, respectively. Both these frameworks are cross-platform, just like Java Swing.

While desktop GUI applications still have their place, it isn't a stretch to say that a majority of new applications being built today are web applications. Python is very well suited to this task. In fact, you may be surprised to know that many of the Internet's most popular websites, like YouTube and Reddit, are written in Python.

Spring MVC is a popular choice for *full stack* web development in the Java world. By *full stack*, I mean a framework that handles everything from frontend user authentication to backend database connectivity and everything in between. The most popular full stack web development framework for Python is **Django**. If you are just dipping your toes into web development with Python, I strongly recommend that you start with Django. You'll find it to be a very easy to learn and productive web development environment.

On the other hand, if you are the type of developer who finds full stack frameworks too restrictive to use, then you can mix and match best of breed libraries and roll your own stack. On the database access front, **SQLAlchemy** is the pre-eminent option. It provides a low-level DB interface like JDBC, a mid-level Spring JDBCTemplate like interface that provides convenient plumbing around writing raw SQL queries and finally, a high-level ORM layer like Hibernate. Depending on your requirements, you can choose just the level of abstraction that you need.

Another aspect of web development is generating responses for HTTP requests, typically HTML pages, using a template engine. This is the kind of work for which you'd reach for the Velocity or Thymeleaf libraries over in Java land. **Jinja2** is the go-to templating library for Python applications. It has a pleasant syntax, lots of features and is fairly fast too.

Python web sites are generally deployed using either the Apache web server or a smaller Python specific web server like uWSGI. These web servers rely on a Python web development standard called *WSGI – Web Server Gateway Interface*. Think of it as the equivalent of Java Servlets in the Python world.

Chapter 4. The Zen of Python

We are finally at the end of this short guide and I hope that, as promised at the start, you've gained sufficient familiarity with the Python language and ecosystem to start hacking on some Python code of your own.

While you dive in to writing Python code, I urge you to also take some time to read existing Python code written by long time Python programmers. Reading code written by experienced *Pythonistas* will give you an appreciation for what is considered to be good and idiomatic Python code. Because in the end, it is not just the pure mechanics of the language, but this pervasive appreciation by the community for simple, readable code that makes Python such a joy to use.

And there's no better distillation of this Pythonic mindset than **The Zen of Python**[1]:

```
antrix@cellar:~$ python
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one -- obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

[1] http://www.wefearchange.org/2010/06/import-this-and-zen-of-python.html

Chapter 5. References

- Python Ecosystem: An Introduction[1]
- The Hitchhiker's Guide to Python![2]
- Style Guide for Python Code[3]
- Python Is Not Java[4]
- Java is not Python, either…[5]
- Code Like a Pythonista: Idiomatic Python[6]
- Secrets of the Framework Creators[7]
- Generator Tricks for Systems Programmers[8]

[1] http://mirnazim.org/writings/python-ecosystem-introduction/

[2] http://docs.python-guide.org/

[3] http://legacy.python.org/dev/peps/pep-0008/

[4] http://dirtsimple.org/2004/12/python-is-not-java.html

[5] http://dirtsimple.org/2004/12/java-is-not-python-either.html

[6] http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html

[7] http://farmdev.com/src/secrets/

[8] http://www.dabeaz.com/generators-uk/

Colophon

Python for the busy Java Developer
Copyright © 2014 Deepak Sarda

Revision 1.0 published on 15 September, 2014
Latest revision available at antrix.net/py4java[9]

This book is written in the *AsciiDoc* format. EPUB3 and PDF versions of the book generated using the *Asciidoctor* toolchain. MOBI version generated using *kindlegen*.

The title font is *Cardo* by David Perry. The body typeface (in the PDF version) is *Source Sans Pro* by Adobe Systems Inc. Code samples and other monospaced text are using *Source Code Pro*, also by Adobe Systems Inc.

The admonition icons are from the *Font Awesome* project by Dave Gandy.

[9] http://antrix.net/py4java