Coding in

python™

A tutorial

By
Ashley Fisher

Introduction

Python is a high-level programming language that emphasizes readability. It was first released in 1991 by Dutch computer programmer Guido van Rossum. Its syntax and semantics are minimalist while its standard library is large.

Many large projects make use of Python, such as Youtube, the original BitTorrent client, and Blender. NASA and Google also use Python. You may be familiar with Alice – it uses Python, too.

So why would you want to learn Python? There are a lot of benefits to coding in Python:

- Code is most often much shorter than C, C++, or Java code, meaning faster development
- Readability – People often compare Python to pseudo code
- Very easy to learn
- You can use Python for almost anything – It works on all operating systems and supports a wide variety of development tools, including GUI toolkits, web scripting, XML processing, and much more. You can even create games in Python.

Of course, every good thing has bad qualities. Python is an interpreted language. This means you do not compile your code. Because of this, Python is much slower than the compiled languages.

Unlike other languages which use curly brackets or keywords to delimit block statements, Python uses only indentation/whitespace. When beginning to learn Python, many programmers find this to be the most annoying aspect of the language. However, this adds to Python's philosophy that code should be neat and easy to read. (You'll be thanking Python later when you have to sift through hundreds of lines of code...)

Python uses duck typing (also known as latent typing, runtime typing, or dynamic typing). For the programmer, this means you do not need to declare your variables – instead, they are assigned a type at runtime. A phrase that is commonly used to describe duck typing is: “If it looks like a duck and quacks like a duck, it must be a duck.” In other words, the type of an object is determined based on how it is used.

Comments in Python start with a hash (#) and are single-line. Example:

```
# This is a comment
```

Before we begin, you may want an IDE in which to write and execute your code. Go to http://www.activestate.com/Products/activepython/ to download the ActivePython IDE.

Oh, and for the record: Python is named after the comedy “Monty Python's Flying Circus”, not after the snake.

Now that you know a little about Python, let's write some code!

Hello world!

The simplest form of a “Hello world” program in Python would look something like this:

```
print 'Hello world!'
```

The single quotes may be replaced with double quotes if you like – Python doesn't care. However, if we want our program to be able to run on all systems, our program would look like this:

```
#!/usr/bin/env python

print 'Hello world!'
```

This allows Linux and Unix users to make their Python scripts executable and call them by name rather than having to explicitly call the Python interpreter (if you're not a Linux user, trust me, this is important!). If you know your code will never be used outside of Windows, you may exclude that first line. However, I do encourage you to include it (“it” is actually called a shebang or hashbang, if you are interested to know).

Data Types

Data types in Python are lists, tuples, and dictionaries. Let's start with lists. Lists are like one-dimensional arrays, although you may have lists of other lists. A list may also contain mixed types (such as an integer, a string, and a boolean). They are written with square brackets. Some examples:

```
teachers = ['Kirkwood', 'Pyatt', 'Reams']

stuff = [[1, 2, 3], ['a', 'b', 'c']]
```

The first example creates a list containing three string elements. The second example creates a list containing two other lists: one with the integers 1, 2, and 3, and the other containing the strings 'a', 'b', and 'c'.

Like in many other languages, `list[0]` is always the first element in the list (or array, tuple, etc). However, Python allows us to use a negative index. An example:

```
print list[-1]
```

This will print the last element in `list`. If we wanted to print the second-to-last element

in list, we would use an index of -2, and so on.

Python has another nifty tool when it comes to dealing with lists. You can access array ranges with a colon (:). This is called slicing.

```
fact = ['Ashley', 'is', 'very', 'cool']
print fact[:]       #Prints the entire list
print fact[:2]      #Prints the list ['Ashley', 'is']
print fact[0:2]     #Prints the list ['Ashley', 'is']
print fact[2:]      #Prints the list ['very', 'cool']
```

As you can see in the above example, `list[:2]` will be every element up to the second, not including the second element. `list[2:]` will be every element after the second, including the second element.

You can add and delete elements in a list.

```
fact.append('indeed')
# fact is now ['Ashley', 'is', 'very', 'cool', 'indeed']
fact.remove('very')
# fact is now ['Ashley', 'is', 'cool', 'indeed']
fact.insert(1, 'Fisher')
# fact is now ['Ashley', 'Fisher', 'is', 'cool', 'indeed']
```

Python also lets you concatenate lists:

```
nums = [5, 3, 9]
num.extend([6, 1])
# nums is now [5, 3, 9, 6, 1]
```

You can search a list using `index`:

```
print nums.index(3) #Prints 1, the index of the element 3
```

You can also add two lists together using the + operator. This would return a new list, whereas extend will modify the existing list. Thus, extend is faster.

You can use the * operator on a list as well.

```
stuff = [1, 2] * 3
# stuff is now [1, 2, 1, 2, 1, 2]
```

Tuples are immutable one-dimensional arrays. Immutable means it cannot be modified after it is created. You would use a tuple almost exactly as you would a list. The difference is that you cannot modify, add, remove, or find elements in a tuple (it has no `index()` method, whereas a list does). Instead of using square brackets, tuples use

parentheses.

```
tup = (2, 4, 'Bob', 9.7)
```

So why would you want to use a tuple instead of a list? There are a few good reasons. First, tuples are faster than lists. If you are only creating a set of constant values to iterate through, use a tuple instead of a list. Second, it “write-protects” data that does not need to be changed. Third, tuples can be used as dictionary keys (as long as they do not contain lists), while lists cannot. Finally, tuples are used in string formatting.

Slicing and negative indexes also work with tuples.

Dictionaries define one-to-one relationships between keys and values. A dictionary in Python is like a Hashtable in Java, or a hash in Perl. We surround a dictionary's contents with curly brackets.

```
person = {'name': 'Kirkwood', 'occupation': 'Teacher'}
```

In this example, `'name'` and `'occupation'` are keys. `'Kirkwood'` and `'Teacher'` are their respective values. If we want to get `person`'s occupation, we would use the expression `person['occupation']`. If we wanted to change the name , we would do the following:

```
person['name'] = 'Reams'
```

As with lists, dictionaries can contain other dictionaries, or even other lists. This allows you to create some pretty cool data structures!

```
student = {'name': 'Ashley', 'classes':
['Math', 'Programming'], 'gpa': 4.0}
```

Remember that you can get values by key, but you cannot get keys by value – that just wouldn't make any sense, now would it? You cannot have duplicate keys in a dictionary. You can, however, add new key-value pairs at any time. The syntax is exactly the same as modifying an existing value, so be careful when you are trying to add a new pair and it isn't working the way you expect – you may just be overriding an existing value.

```
person = {'name': 'Ashley'}
person['age'] = 16
```

This will add a new key-value pair to `person`. You can also delete from a dictionary:

```
del person['age']
# person is now {'name': 'Ashley'}

person.clear()
```

```
# person is now empty.
```

Flow Control Statements

The flow control statements in Python are `while`, `if`, and `for`. There is no `do` loop. Unlike in other languages, test expressions do not have to be enclosed in parentheses. An example `while` loop would look like this:

```
num = input('Enter the number of iterations: ')
while num <= 0:
     print num
     num -= 1
```

The above code will print all integers, starting with the user's input and decreasing to zero.

Something we have not previously talked about is `input`. This allows you to get a value from the user. The string you supply to it will be printed to the screen. Although this is not necessary in learning flow control statements, it is certainly useful!

`if` statements work how you might expect, but with a couple of minor tweaks.

```
grade = input('Enter the grade: ')
if grade >= 90:
     print 'The student earned an A'
elif 80 <= grade < 90:
     print 'The student earned a B'
elif 70 <= grade < 80:
     print 'The student earned a C'
elif 60 <= grade < 70:
     print 'The student earned a D'
else:
     print 'The student earned an F'
```

Instead of `else if`, Python uses the keyword `elif`. No big deal, right? However, you may have noticed something strange about the tests performed above. Notice that we were able to type `elif 80 <= grade < 90`. Python allows you to test a range of values. Cool, eh?

Of course, your usual `and` and `or` still work.

```
if gpa < 0 or gpa > 4.0:
     print 'This GPA is not valid!'
```

Our final flow control statement is the `for` loop. Python's `for` works like the for each loop used in many other languages. Say you simply wanted to print all values from 1 to

100:

```
for i in range(100):
     print i + 1
```

The `range(100)` function returns a list containing integers from 0 to 99. The `for` in Python is easier to use than in other languages, where you must specify a start, end, and step value. Instead, the `for` loop in Python simply iterates through a list.

Say you have some list `people` containing many names. You could iterate through the list and `print` the names very easily with a `for` loop.

```
for name in people:
     print name
```

You can iterate through a dictionary, too – just use the `items()` function. Assume you have defined a dictionary called `info`:

```
for key, value in info.items():
     # Do whatever
```

Functions

Of course, what would Python be without functions? Functions are declared using the `def` keyword.

```
def add(num1, num2):
     return num1 + num2
```

Every Python function returns a value. If the function does not contain a `return` statement, it returns `None`, the Python null value.

You may also assign a default value to any of your parameters like so:

```
def mult(num1, num2 = 2):
     return num1 * num2
```

By doing the above, the parameter `num2` is now optional. If it is not supplied, it will by default be 2.

Functions in Python are "Pass By Value". If we had the following function:

```
def double(num):
     num *= 2
     print num
```

And we called it up like so:

```
x = 3
double(3)
```

The function would print 6 to the screen, but `x` would still be 3.

Functions may return multiple values:

```
def reverse(arg1, arg2, arg3):
     return arg3, arg2, arg1

val1, val2, val3 = reverse(1, 2, 3)
print val1, val2, val3
```

The above will print `3 2 1` to the screen.

Python supports something called an anonymous function, using the `lambda` keyword. A `lambda` function is basically a one-line mini-function. Say you had the following one-line function:

```
def f(x):
     return x * 2
```

You can achieve the same task with a `lambda` function:

```
g = lambda x: x * 2
```

And to call it up:

```
print g(3)
```

You do not have to assign a `lambda` function to a variable, although it is rather pointless not to:

```
print (lambda x: x * 2)(3)
```

They all get the same job done – it's all a matter of what is most convenient to you at the time. By the way, you can assign any function to a variable, not just `lambda` functions. Say you already defined a function called `func`.

```
a = func
a(5)
```

Calling `a(5)` is the same as calling `func(5)`.

Classes

Python is an object-oriented language, so, naturally, you can define your own classes. There are no separate header/implementation files. Just define the class and use it! We use the keyword `class` to start a class definition.

```
class Student:
     school = 'North'
     def __init__(self, n, g):
          self.name = n
          self.gpa = g
     def show(self):
          print 'Name: ' + self.name + '\nGPA: ' + self.gpa
```

The variable `school` will be shared among all instances of `Student`, since it is defined outside of a function.

`__init__` is like the class's constructor, but it actually isn't. `__init__` is called after the object is constructed. It acts much the same way as a constructor might – just remember that it isn't really constructing the object.

Every function inside of a class will take the same first parameter: `self`. This is always a reference to the current instance. `self` is not a reserved word, but it is a very strong convention. Always follow it.

If we wanted to instantiate the class `Student`, we would do as follows:

```
s1 = Student('Bob', 3.75)
```

And if we wanted to call the object's show:

```
s1.show()
```

If you know Java, you'll know toString(). Well, Python has something like it. It's called `__str__`. This will define how the object wants to look when you try to print it.

```
def __str__(self):
     return 'Name: ' + self.name + '\nGPA: ' + self.gpa
```

So you can now do:

```
print s1
```

Python also allows for inheritance. Say you wanted to create a class `HonorStudent` that inherits `Student`:

```
class HonorStudent(Student):
     def __init__(self):
          Student.__init__(self)
```

Notice that the superclass is enclosed in parentheses after the subclass name. If you defined an `__init__` function in the superclass, you must call it in the subclass's `__init__` as shown in the above example. Also, remember that if you call any of `Student`'s methods from within `HonorStudent`, you must include the `self` argument.

You can even do multiple inheritance in Python. Say you wanted `HonorStudent` to inherit `Student` and some other class `Person`:

```
class HonorStudent(Student, Person):
     # Class definition
```

It can inherit more than two classes if you want. This is just an example. Just remember that if the superclass defines `__init__`, you must call it in the subclass's `__init__`.

Want More?

Obviously this is only a sampling of what Python has to offer. You can (and should!) check out more tutorials to learn more about Python. The following link has a multitude of tutorials for programmers who want to start learning Python.

http://wiki.python.org/moin/BeginnersGuide/Programmers

"Dive Into Python" is the tutorial I learned the most from. You can find it at:

http://www.diveintopython.org/

Python also allows you to quickly write and test code line-by-line without writing it to a file, using the Python Interactive Interpreter. You can find it at http://activestate.com. This is especially useful when you are first learning what you can and cannot do in Python.

Python Philosophy

I included this section for fun (and personal satisfaction). It is certainly not required for you to learn Python, although you may find it useful in understanding the language.

Code which adheres to Python's principles of readability is said to by “pythonic”. In his book “The Zen of Python”, Tim Peters characterized these principles into 19 statements:

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