Introduction to Python

For existing programmers

Overview

This document aims to act as a quick reference guide for writing Python code for people who already have experience with programming.

Interpreter

Python is an interpreted language. This means that at run-time the python code is read, parsed and then executed (as a byte-code). When installed, the interpreter can be invoked on most (appropriately configured) systems by writing:

python path/to/script.py

Which would cause Python to interpret the contents of the **script.py** file stored in **path/to/script/** On some platforms it is also possible to double-click a python script in order to execute it.

By only typing **python** into a command-line without a filepath afterwards it is possible to run the Python interpreter as a *REPL*. This stands for **R**ead, **E**valuate, **P**rint and **L**oop. What means the interpreter:

1. **R**eads a line of code that you type in at a prompt
2. **E**valuates the code
3. **P**rints the resulting value (if there is one) before,
4. **L**ooping back to the beginning (returning to step 1)

Code meant for the *REPL* is identifiable as starting with three greater-than symbols, for example:

>>> **2 + 4**
6

This shows a user typing **2 + 4** into the Python *REPL*, which is evaluated and results in the value **6** being printed. In this document, the code entered will be **bold**, and the resulting output will be in ***bold italics***. Sometimes you need to enter more than one line of code at once, when you do this the interpreter will print three dots instead of three greater-than symbols at the start of each extra line:

>>> **(1, 2,**
… 3)
(1, 2, 3)

This shows a *tuple* being created with 3 values: 1, 2 and 3.

Simple Values/Literals

Numbers

Python numeric literals are essentially the same as most other languages:

- Integer literals: **0, 10, 12345, -4, -200**
- Decimal (floating point) literals: **0.0, 0.1, 100.24, 3.14159**
- Hexadecimal integers: **0x0, 0x10, 0xFF, 0xC0FFEE**
- Binary integers: **0b0, 0b1001, 0b01000001**

Strings

String literals are also like that in most other languages:

- Normal string literal with escape character support:
  “Hello this is a string”,
  “First line\nSecond line”,
- Normal string literal ignoring escape characters:
  “””Hello this text contains \n backslash and 'n', not a newline”””,
  “””These strings also ignore quotation marks “ until they see three consecutively”””
- Unicode strings:
  u”Hello this is a unicode string”
- Byte strings (just bytes, not characters with an encoding):
  b”So many bytes right here”

Expressions

Simple arithmetic expressions use typical infix notation:

>>> **2 + 4**
6
>>> **18 / 2**
9
>>> **2 ** 8**
256
>>> **1 – 8**
-7
>>> **2 * 5**
10
>> **(2 * 8) + 13**
29

Variables

Declaring a value in Python is as simple as assigning a value to a variable that does not currently exist:

```
>>> apple_count = 10
```

To retrieve the value of a variable in the *REPL* simply enter its name:

```
>>> apple_count
10
```

These can be used in expressions like many other languages:

```
>>> apple_count – 3
7
```

Updating a variable's value just involves assigning to it, the same way the value was created:

```
>>> apple_count
10
>>> apple_count = apple_count – 3
>>> apple_count
7
```

Containers

Python supports three built-in container types:

1. *Tuples*:

   These represent a fixed-length group of values. For example, you could use them to represent an X,Y,Z co-ordinate in 3D space:
   (10, 4, 8)

   Or group related information together, such as name and birth year:
   >>> **me = (“Oliver”, 1991)**

   You can extract values using *sub-script* notation, to extract the name from the tuple above:
   >>> **me[0]**
   “Oliver”

   For the birth year:
   >>> **me[1]**
   1991

2. *Lists*:

   Similar to tuples these represent many values but are not fixed-length. You can therefore use them for things like, recording temperature in degrees over time:

   >>> **temperature = [20.5, 20.7, 20.6]**
   >>> **temperature**
   [20.5, 20.7, 20.6]
   >>> **temperature.append(20.3)**
   >>> **temperature**
   [20.5, 20.7, 20.6, 20.3]

   Again, like tuples you can use *sub-script* notation to extract a specific value:
   >>> **temperature[0]**
   20.5
   >>> **temperature[2]**
   20.6

3. *Dictionaries*:

   This type represents a key-value map, where the key is commonly a string and the value is of any type. This sort of collection is often explained using a phonebook example:
   >>> **phonebook = {'Harry': '01234 567 891', 'John': '01234 576 819'}**
   >>> **phonebook['Harry']**
   01234 567 891
   >>> **phonebook['John']**
   01234 576 819

Control Flow

Conditional Statements

In Python conditional logic can be implemented with if-statements as follows:

```
>>> if True:
...         print "True is True! Who'd have thought?"
...     else:
...         print "True isn't True? How can this be!?"
```

True is True! Who'd have thought?

This is, as may be apparent, the expression between **if** and the colon on the first line is used to decide whether to execute the *true branch* or the *false branch*. If the expression results in **True** then the first statement is run, if **False** the second statement is run. In this example we cheated and used the constant value **True** as the expression, **True** always evaluates to **True** (thankfully) and as a result the first branch is always run.

This guide won't explain how all of the types can or can not be evaluated to **True** or **False**, but here is a quick list of *boolean operators* that produce **True/False** outputs depending on their arguments:

- For testing equality:

  \>>> **1 == 1**
  True
  \>>> **1 == 2**
  False

- Less than:

  \>>> **1 < 2**
  True
  \>>> **1 < 1**
  False

- Greater than:

  \>>> **1 > 4**
  False
  \>>> **4 > 1**
  True

- Checking for difference:

  \>>> **1 != 1**
  False
  \>>> **2 != 3**
  True

For Loops

To make a piece of code repeat 5 times, such as

```
>>> for x in range(5):
...     print "Hello!"
...
Hello!
Hello!
Hello!
Hello!
Hello!
```

This creates a variable **x** that is visible within the *for-loop* that represents the current value in the loop. This makes more sense when you consider that a *for-loop* in python is actually something that takes a list of values and iterates over each value in turn.

Suppose you have a list of numbers:

```
>>> my_list = [1, 1, 2, 3, 5, 8, 13]
```

You can print each one on a new line with:

```
>>> for number in my_list:
…       print number
1
1
2
3
5
8
13
```

Looking back at the first example, you can now consider **range(5)** equivalent to

[0, 1, 2, 3, 4]

and in fact, typing **range(x)** into the *REPL* with different values of **x** will hint at this:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
```

This isn't technically exactly how **range()** works, but suffices as an explanation for now.

While Loops

In Python a *while-loop* is similar again to most other languages, you can use one to print numbers 0 to 5 as follows:

```
>>> counter = 0
>>> while counter < 6:
...     print counter
...     counter = counter + 1
...
0
1
2
3
4
5
```

Functions

To define a function, such as squaring a given number:

```
>>> def square(x):
...     return x ** 2
...
```

Then to call this function:

```
>>> square(10)
100
```

A function does not have to return a value:

```
>>> def say_hello_to(person):
...     print “Hello,”, person
...
>>> say_hello_to(“John”)
Hello, John
```

A function also does not need parameters:

```
>>> def fetch_answer():
...     return 42
...
>>> fetch_answer()
42
```

Classes / Objects

Using the 3D co-ordinate example to show off Python classes:

```
>>> class Coordinate(object):
…       def __init__(self, x, y, z):
…               self.x = x
…               self.y = y
…               self.z = z
…
```

This defines a class called *Coordinate*. We define a *method* of this class called __*init*__. This is a special function, Python has several such functions with different names. Each is used for a specific task. The init function is Pythons way of describing a *constructor*. In this case we are declaring a *constructor* for the *Coordinate* class that takes four parameters: **self**, **x**, **y**, and **z**. **self** is a reference to the instance we are creating and is passed automatically by Python. **x**, **y** and **z** are the parameters passed to the constructor manually. This code then creates *member* variables (or *fields*) with the same names (**x, y, z**) and stores the paramter values in them.

To create an instance of our *Coordinate* class:

```
>>> origin = Coordinate(0, 0, 0)
>>> my_point = Coordinate(100, 15, 82)
```

and to access values in them:

```
>>> origin.x
0
>>> my_point.y
15
>>> my_point.x + my_point.z
182
```

As an example, defining the class again with a *distance method* to calculate the distance from the origin (0,0,0) using Pythagorus' Theorem:

```
>>> class Coordinate(object):
…       def __init__(self, x, y, z):
…               self.x = x
…               self.y = y
…               self.z = z
…       def distance(self):
…               return (self.x ** 2 + self.y ** 2 + self.z ** 2) ** 0.5
…
>>> point = Coordinate(5, 6, 7)
>>> point.distance()
10.488088481701515
```