An Introduction to Python for use with GNU Radio

Version 1.0 (18th April 2014)

Balint Seeber
Ettus Research

Comments & suggetions welcome:
balint@ettus.com
@spenchdotnet

At the shell prompt, enter the Python interpreter by typing:
`python`

You will see:
```
Python 2.7.6 (default, Jan 11 2014, 14:34:26)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```
You are now in the interpreter and can type in Python expressions after the prompt `>>>`.

Print strings to the screen with `print`:
```
>>> print "Hello World"
Hello World
```

Python will evaluate mathematic expressions:
```
>>> 1 + 1
2
```

If only integers are used, then results are rounded:
```
>>> 1 / 2
0
```

To indicate a floating-point variable, use a decimal point. This will avoid rounding because subsequent operations will use floating-point calculations:
```
>>> 1. / 2
0.5
```

To perform integer divisions, regardless of whether the operands are integers or floating-point values, you can use the double slash `//`:
```
>>> 1. // 2
0.0
```

For more advanced mathematical operations, `import` the math module. Modules are external libraries that can be pulled in for use in your program.
```
>>> import math
>>> math.log10(1000)
3.0
```

To list what is available in a module, use `dir`:
```
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
```

When an expression is evaluated, the result is printed to the screen:
```
>>> math.sin(0.4)
0.3894183423086505
```

Don't forget to prepend the module name (here: `math.`) when calling a function inside a module imported as above:

```
>>> 5 * sin(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
```

Assign the result of an expression to a variable. In this case the result is not printed.

```
>>> my_number = 5 * math.sin(5)

To show the result, treat it as a simple expression:
>>> my_number
-4.794621373315692
```

Or `print` it out:

```
>>> print my_number
-4.79462137332
```

Lists are denoted by square brackets. Assign an empty list to a variable:

```
>>> my_list = []
```

Populate a list by separating items with commas:

```
>>> my_list = [1,2,3,4,5]
```

Print the contents of the list:

```
>>> my_list
[1, 2, 3, 4, 5]
```

Lists can contain items of any type (e.g. integers and strings – strings can be denoted by a matched single or double quote):

```
>>> my_list = [1,2,3,4,5,'a','b',"Hello World",my_number]
>>> my_list
[1, 2, 3, 4, 5, 'a', 'b', 'Hello World', -4.794621373315692]
```

Obtain the type of a variable with the `type` function:

```
>>> type(my_list)
<type 'list'>
```

To select one element from a list, use square brackets with the item's zero-based index:

```
>>> my_list[4]
5
>>> my_list[7]
'Hello World'
```

A negative index can be used to select items counting back from the end of the list:

```
>>> my_list[-1]
-4.794621373315692
```

The len function returns the length of a list:

```
>>> len(my_list)
9
```

A range of items can be selected from the list by using `[`(start index)`:`(one greater than the end index)`]`:

```
>>> my_list[1:3]
[2, 3]
```

Leaving off the start or end parts will return the rest of list before/after the given index. If the end is omitted, the rest of the list is returned beginning at the start index:

```
>>> my_list[1:]
[2, 3, 4, 5, 'a', 'b', 'Heloo World', -4.794621373315692]
```

If an index is used that is greater than the list's length, an empty list is returned:

```
>>> my_list[9:]
[]
>>> my_list[10:]
[]
```

If the start index is omitted, list elements up to the end index are returned:

```
>>> my_list[:3]
[1, 2, 3]
```

In addition to the start and end indices, a third optional argument can be passed after the end index to set the index step size. Here the start and end indices are omitted so all items are considered, but the step of -1 means item indices will be counted in reverse. This effectively reverses the list:

```
>>> my_list[::-1]
[-4.794621373315692, 'Hello World', 'b', 'a', 5, 4, 3, 2, 1]
```

To set a new item at an existing index, perform a norma variable assignment:

```
>>> my_list[0]
1
>>> my_list[0] = "Good day:"
>>> my_list
['Good day:', 2, 3, 4, 5, 'a', 'b', 'Hello World',
-4.794621373315692]
```

To append a list to an existing one, use the addition operator. It is not possible to append a single item on its own to a list, so you must wrap the item in a new list first:

```
>>> my_list += ["A new item"]
>>> my_list
['Good day:', 2, 3, 4, 5, 'a', 'b', 'Hello World',
-4.794621373315692, 'A new item']
```

A list instance is an object with functions. For example, `remove` will delete the first instance an item from a list (e.g. `2` here is not an index – it is the actual item):

```
>>> my_list.remove(2)
>>> my_list
['Good day:', 3, 4, 5, 'a', 'b', 'Hello World',
-4.794621373315692, 'A new item']
>>> my_list.remove('A new item')
>>> my_list
['Good day:', 3, 4, 5, 'a', 'b', 'Hello World',
-4.794621373315692]
```

A Python dictionary is a mapping of keys to values, and is indicated by curly braces. This is an empty dictionary:

```
>>> b = {}
>>> b
{}
```

Initialise a dictionary with some keys and values. Keys and values are paired with a colon, and key-value pairs are separated by a comma:

```
>>> b = {1:2, 3:4, 5:6}
>>> b
{1: 2, 3: 4, 5: 6}
```

To retrieve the value of a given key, supply the key (note this is the same syntax as with a list, however here the value is a key, not an index):

```
>>> b[1]
2
>>> b[5]
6
```

To assign a new value to a key (which may, or may not, exist):

```
>>> b[5] = 10
>>> b[5]
10
>>> b
{1: 2, 3: 4, 5: 10}
```

Dictionaries can also store arbitrary types:

```
>>> b['hello'] = 'bye'
>>> b
{1: 2, 3: 4, 5: 10, 'hello': 'bye'}
>>> b['hello']
'bye'
>>> b[5]
10
```

If a key does not exist, an exception is thrown:

```
>>> b[11]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 11
```

A dictionary is also an object with functions. To retrieve a dictionary's keys as a list:

```
>>> b.keys()
[1, 3, 5, 'hello']
```

The `in` keyword can be used to test if an item is in a list:

```
>>> 11 in b.keys()
False
>>> 1 in b.keys()
True
```

The `if` construct evaluates an expression and executes the following code if the evaluation is true. Here it is done in a one-liner (usually it occupies multiple lines, and the

interpretter allows you to enter more code by prompting you with … . To indicate the last line, press the Enter key and the code will be evaluated).

```
>>> if 1 in b.keys(): print "All good"
...
All good
```

Python has many built-in functions. For example, `sum` will add together all the elements of a list (provided they support being added together). They are listed here: http://docs.python.org/2.7/library/functions.html

```
>>> nums = [1,2,3,4,5,6]
>>> sum(nums)
21
```

Another is `sorted`, which will sort a list. Typing a function on its own doesn't call the function – it returns the function as an object.

```
>>> sorted
<built-in function sorted>
```

Since `nums` is already in ascending order, we reverse it first, and then sort it to demonstrate `sorted`:

```
>>> sorted(nums[::-1])
[1, 2, 3, 4, 5, 6]
```

The sorted list can be ordered in a descending manner by re-using the list step-size trick:

```
>>> sorted(nums[::-1])[::-1]
[6, 5, 4, 3, 2, 1]
```

A list can be reversed using the `reverse` function, but this modifies the existing list in-place (as opposed to creating a copy of the list, reversing that and returning that copy as has been done previously with `[::-1]`). This is shown here as there is no objected returned and printed after evaluating the expression:

```
>>> sorted(nums[::-1]).reverse()
```

Instead we operate on a variable that we can print out ourselves:

```
>>> the_list = sorted(nums[::-1])
>>> the_list
[1, 2, 3, 4, 5, 6]
>>> the_list.reverse()
>>> the_list
[6, 5, 4, 3, 2, 1]
```

Strings can be treated like a list of characters:

```
>>> a_string = "ICTP"
>>> "ICTP"[1:2]
'C'
>>> "ICTP"[1:]
'CTP'
>>> a_string
'ICTP'
```

The built-in function `ord` returns the ASCII character code for a single character:

```
>>> ord('a')
97
```

The built-in `map` applies a function (the first argument) to a list of items (the second argument), and returns a new list with the result of each evaluation of the function with each item. Here we transform a string into a list of the constituent characters as ASCII codes by having `map` call `ord` for each element in the supplied string:

```
>>> map(ord, "ICTP")
[73, 67, 84, 80]
```

We can then perform another map using the built-in function `chr`, which converts ASCII character codes back into the actual character, to restore the original letters. Note this produces a list, not a string:

```
>>> map(chr, map(ord, "ICTP"))
['I', 'C', 'T', 'P']
```

To create a single string from a list of strings (or single characters, which are a string of length one), the `join` function can be used. `join` is a member function of the string object. Here we use it on an empty string, since the given string is inserted between each item in the supplied list of strings to be joined.

```
>>> "".join(map(chr, map(ord, "ICTP")))
'ICTP'
```

Lambda functions are anonymous functions that can be used as arguments to other functions, such as `map`. This is a short-hand way of describing a simple expression without having to write a proper Python function. Here we will use it to perform a simplistic tranposition cipher. A lambda function is denoted as `lambda <arguments' variables>: <expression>`

```
>>> step1 = map(ord, "ICTP")
>>> step1
[73, 67, 84, 80]
>>> step2 = map(lambda x: x+1, step1)
>>> step2
[74, 68, 85, 81]
```

Note that `x` is the arbitrarily-chosen variable name to represent the single item to be evaluated by the lambda expression (`x+1`). For each ASCII character code, `map` applies the lambda function to the code, resulting in it being incremented by one. A new list is returned where each code is one greater than the original code.

We can then use `join` to create a new string:

```
>>> "".join(map(chr,step2))
'JDUQ'
```

If we need to store the result, a variable can be used:

```
>>> step3 = map(chr,step2)
>>> step3
['J', 'D', 'U', 'Q']
>>> "".join(step3)
'JDUQ'
```