Python for Scientists: Basics

May 13, 2014
Outline

1. Basic Python
2. Scripting
3. Lists and Indexing
4. Basic Flow Control
5. Objects in Python
6. String Manipulation
7. os: Interface to the OS
Starting up Python

Open a terminal and type the following:

```
$ python
```

This is the *interactive shell*.

- What version of Python are you running?
- Quit the shell by typing Control-D (EOF)
Python distinguishes between integers and floating point:

- >>> 1 + 2 - 3 * 4 / 5

- >>> 1. + 2. - 3.* 4./ 5.
Python distinguishes between integers and floating point:

- `>>> 1 + 2 - 3 * 4 / 5`
  
  1

- `>>> 1.+ 2.- 3.* 4./ 5.`
  
  0.6000000000000001

Watch your decimal points!

Note: 4 / 5 = 0.8 in Python 3.x
New Syntax

- Exponents: >>> 3**4
- Complex numbers: >>> (1+2j)*(2+3j)
- Last output: >>> _ + 1
- Truncated division: 7./ 4.
New Syntax

- Exponents: 
  ```python
  >>> 3**4
  81
  ```
- Complex numbers: 
  ```python
  >>> (1+2j)*(2+3j)
  (-4+7j)
  ```
- Last output: 
  ```python
  >>> _ + 1
  (-3+7j)
  ```
- Truncated division: 
  ```python
  7.// 4.
  1.0
  ```
Logical Operators

Python has the usual logical operators:

- >>> 1 < 2
- >>> 3 >= 2
- >>> 5 > 4 and 2 == 3
- >>> 1 < 2 < 3
Python has the usual logical operators:

- >>> 1 < 2
  True

- >>> 3 >= 2
  False

- >>> 5 > 4 and 2 == 3
  False

- >>> 1 < 2 < 3
  True

1 < 2 < 3 is shorthand: 1 < 2 and 2 < 3

Try 1 < 2 == 2 and (1 < 2) == 2.
Modular arithmetic is not quite like C and Fortran:

- `>>> 7 // 4`
- `>>> -7 // 4`
- `>>> 7 % 4`
- `>>> -7 % 4`
Modular arithmetic is not quite like C and Fortran:

- \[ \gggg 7 \div 4 \]
  - 1

- \[ \gggg 7 \mod 4 \]
  - 3

- \[ \gggg -7 \div 4 \]
  - -2

- \[ \gggg -7 \mod 4 \]
  - 1
Modular arithmetic is not quite like C and Fortran:

- >>> 7 // 4
  1
- >>> 7 % 4
  3
- >>> -7 // 4
  -2
- >>> -7 % 4
  1

For \( q = x / N \) and \( r = x \% N \), Python solves

\[
x = qN + r
\]

so that \( 0 \leq r < N \) for \( N > 0 \).
Create variables at any time (*dynamic typing*):

```python
x = 123.
y = 456.
z = x + y
x = 'Now x is a string.'
z = x + y  # ???
```

Assigning several variables:

```python
x = y = z = 1
```
math and Modules

*Modules* let you import external functions, variables, etc.

math provides standard mathematics functions. Import the *math* module and try some functions:

```python
import math
x = math.pi
y = math.sin(x/2.)
```

(Note: *numpy* provides most of the same functions.)
You can rename module imports

```python
import math as m
x = m.pi
y = m.sin(x/2.)
```

or import specific functions:

```python
from math import pi
```

or the entire contents (not recommended though):

```python
from math import *
```
Basic I/O

Basic statement printing ('echo'):

```python
>>> print 'Hello Python.'
Hello Python.
```

Commas separate quantities by spaces:

```python
>>> x = 1
>>> y = 2
>>> print 'x =', x, 'and y =', y
x = 1 and y = 2
```
Outline

1. Basic Python
2. Scripting
3. Lists and Indexing
4. Basic Flow Control
5. Objects in Python
6. String Manipulation
7. os: Interface to the OS
Create a file named hello.py containing the following:

```python
print "This is my first Python script."
name = raw_input("Type your name: ")
print "Hello", name
print "Goodbye."
```

Now run the script by typing the following:

```bash
$ python hello.py
```
Self-executing scripts

Put the following as your first line in the script:

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

Then make the script executable by typing in the terminal:

```bash
$ chmod +x hello.py
```

Now you can run the script as an executable:

```bash
./hello.py
```
Lists are sequences of data

```python
x = [0, 1, 2, 2]
y = [4, 4.0, 'four']
z = [x, y, 1, [2,3]]
```

Lists can contain anything, even other lists.
List Indexing

Indexing begins at zero, denoted by []:

```python
>>> x = [1, 2, 3, 4, 5]
>>> x[0]
1
>>> x[4]
5
```

`range(N)` will generate an 'index list' of size $N$:

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

\[ x[m:n] \text{ outputs a segment from } m \leq i < n. \]

For \( x = \text{range}(8) \), try the following:

- \( x[3:6] \)
- \( x[:6] \)
- \( x[3:] \)
- \( x[:] \)

When omitted, \( m = 0 \) and \( n = \text{len}(x) \)
Slicing

\[ x[m:n] \] outputs a segment from \( m \leq i < n \).

For \( x = \text{range}(8) \), try the following:

- \( x[3:6] \)
  
  \[ [3, 4, 5] \]

- \( x[:6] \)
  
  \[ [0, 1, 2, 3, 4, 5] \]

- \( x[3:] \)
  
  \[ [3, 4, 5, 6, 7] \]

- \( x[:] \)
  
  \[ [0, 1, 2, 3, 4, 5, 6, 7] \]

When omitted, \( m = 0 \) and \( n = \text{len}(x) \).
A second colon denotes stride, e.g. \( \text{x[m:n:k]} \)
\( \text{x[::3]} \) returns every third element

```python
>>> x = range(10)
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[2::3]
[2, 5, 8]
>>> x[::-1]  # Reverse stride
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```
Python supports *negative indexing*:

```python
x = range(8)

- x[-1]
- x[:-3]
- x[2:-2]
```

`x[-k]` is just shorthand for `x[N-k]`. 
Python supports *negative indexing*:

```python
x = range(8)
x[-1]
[7]
x[:-3]
[0, 1, 2, 3, 4]
x[2:-2]
[2, 3, 4, 5]
```

`x[-k]` is just shorthand for `x[N-k]`. 
Write a script that reads a number \( N \) using this command:
\[
N = \text{int}(\text{raw_input}('Enter \ N: '))
\]

Create a list using \( \text{range}(N) \) and print the following:
- The entire list
- The reversed list
- The first six elements
- The last three
- Everything but the last three
- Every third element between the first four and last four
- The first half of the list
- the last five elements, reversed
One solution, at least.

```python
N = int(input('Enter a list size: '))
x = range(N)

print 'List:', x
print 'Reversed:', x[::-1]
print 'First six:', x[:6]
print 'Last three:', x[-3:]
print 'Minus last three:', x[:-3]
print 'Middle minus 4:', x[4:-4:3]
print 'First half:', x[:len(x)//2]
print 'Last five, rev:', x[-6:-1]
```
Tuples are like lists, except they cannot be modified:

```python
>>> x = tuple([1, 2, 3])
>>> x
(1, 2, 3)
>>> x[0] = 5
```

A favorite swap trick:

```python
>>> x = 1
>>> y = 2
>>> x, y = y, x
```

Parentheses are often implicit.
Dicts are mappings:

```python
>>> d = {'one': 1, 'two': 2, 'three': 3}
```
```
>>> d['one']
1
```

Like lists, with generic indexing

Fun way to create a dict:

```python
>>> x = ['one', 'two', 'three']
>>> y = [1, 2, 3]
>>> zip(x, y)
[('one', 1), ('two', 2), ('three', 3)]
>>> d = dict(zip(x,y))
```
Outline

1. Basic Python
2. Scripting
3. Lists and Indexing
4. Basic Flow Control
5. Objects in Python
6. String Manipulation
7. os: Interface to the OS
Python flow control is minimal:

```python
x = 1
if x > 0:
    print "x is positive"
elif x < 0:
    print "x is negative"
else:
    print "x is zero"
```

Notice:
- Conditionals end with :;
- There is no endif, closing bracket, etc.
- Indentation is mandatory!
Python has a `while` loop:

```python
x = 0
while x < 10:
    print x
    x += 1
```

but in practice they are rarely used.
The most common Python loop:

```python
for i in range(10):
    print i
```

In Python, you *iterate* through the list:

```python
x = [1, 3, 7, 'apple', 'banana']
for i in x:
    print i
```

Iterating through elements is preferred (and usually fastest).
List comprehensions are rapid methods for list generation:

Create a list of even numbers:

```python
x = [i for i in range(10) if i%2 == 0]
```

Create a list of squared numbers:

```python
y = [i**2 for i in range(10)]
```
Minimal quadratic function:

```python
def square(x):
    return x*x
```

Anything can be a function argument, even other functions:

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

def g(f, N):
    return [f(i) for i in range(N)]
```

Try $f(4)$ and $g(f, 10)$. 
Outline

1. Basic Python
2. Scripting
3. Lists and Indexing
4. Basic Flow Control
5. Objects in Python
6. String Manipulation
7. os: Interface to the OS
In Python, everything is an object: integers, strings, lists, functions, etc. All objects have a type():

```python
>>> type(3)
>>> type([1,2,3])
>>> type('Hello')
>>> type(type)
```

Variables are just references to the objects in memory:

```python
>>> x = 3
>>> type(x)
>>> x is 3
```
Object Methods

dir(x) returns all the contents (i.e. objects) of x.
Create a list and look inside:

```python
>>> x = range(4)
>>> dir(x)
```

The dot (.) lets you access these contents of an object, e.g.

```python
>>> x.append(9)
>>> x
[0, 1, 2, 3, 4, 9]
```

Use help() on some of the methods and try them out. (Ignore the ones with underscores)
Variables and References

There are *immutable* and *mutable* objects. Immutable objects cannot be changed: numbers, strings, tuples.

```python
>>> x = 3  # x points to 3
>>> y = x  # y points to what x points to
>>> x = 4  # Now x points to 4
>>> x
4
>>> y
3
```

The number 3 did not (and cannot) change, so `y` still points to 3.

Do not think that `y` ‘contains’ the value of 3!
Now let x point to a mutable list:

```python
>>> x = [1,2,3]  # [1,2,3] is created
      # and x points to it
>>> y = x     # Now y points to the list
>>> x[0] = 7  # Change part of x
>>> y        # What is y?
```

NumPy arrays are mutable, and behave similarly.
Outline

1. Basic Python
2. Scripting
3. Lists and Indexing
4. Basic Flow Control
5. Objects in Python
6. String Manipulation
7. os: Interface to the OS
Strings support a \texttt{printf}-like syntax:

```python
>>> x = 1
>>> print 'x = %i' % x
>>> y = 2.0
>>> print 'x = %i and y = %.2f.' % (x,y)
```

There are also several methods to parse and manipulate strings.
Outline

1. Basic Python
2. Scripting
3. Lists and Indexing
4. Basic Flow Control
5. Objects in Python
6. String Manipulation
7. os: Interface to the OS
import os provides OS-level commands, e.g.:

- **os.system(cmd)**
  Run cmd in the shell (but see subprocess)

- **os.getcwd()**
  Return your current directory

- **os.listdir(path)**
  List of all filenames in path

- **os.mkdir(path)**
  Create a directory named path

- **os.remove(path)**
  Delete the file at path

- **os.path.join(path1, path2)**
  Append path2 to path1 as an absolute path

Also: sys, shutil, subprocess