# Basic Python

Carver J. Bierson

June 26, 2017

## 1 Why Python

On GitHub (the major code repository on the internet) Python is the third most language. The two languages higher than Python, Java and JavaScript, are not as well suited to scientific applications as Python. Because of its popularity there are many libraries that you can take advantage of (why write code someone else has already done). Python is also used extensively in industry (Google's tensor flow machine learning software has a Python interface).

### 1.1 When to use Python

Programming languages are tools and you need to know when a language is the right tool for the job. Python is a easy language to program in but the trade off is that it is fairly slow at executing code. Generally your time is more valuable than the computers. I generally prototype large codes in python, then translate them to a more efficient language (C, C++, Fortran). Near the end of the class we may compare the speed of using a pure python program vs having python call a compiled program.

### 1.2 Python 2.7 or Python 3.6

About ten years ago Python 3 came out. Python 3 has many advantage over Python 2 but is not backwards comparable (you can't always run python 2 in python 3. Because of this many groups still have not fully converted to python 3 (for the same reason I still get code from collaborators in Fortran 77). For what we do in this class you probably won't notice any difference (except the print statements are a little different). **I will accept assignments in either version but you must always specify in the comments which version you used!** I recommend learning python 3 but the choice is yours.

## 2 Python as a calculator

```
>>> 5*2
10
>>> 5+2
7
>>> 4/2
2
```

```
>>> 5/2
2
>>> 5.0/2.0
2.5
>>> 1+2*3
7
>>> (1+2)*3
9
>>> 3**2
9
>>> 9**1/2
4
>>> 9**(1/2)
1
>>> 9**(1.0/2.0)
3.0
>>> cos(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
```

Core python has basic mathematical operations but for functions like cosine we will use libraries.

The two main libraries we will use in this class are numpy (numerical python) and matplotlib (plotting). You can load libraries like

```
>>> import numpy as np
```

Here numpy is the library name and I am calling it np for later use. After this I can do

```
>>> np.cos(0)
1.0
```

You may see some places use

```
>>> from numpy import *
```

This makes it so you do not have to use the np prefix when you want to use those functions. The danger with this method is if you load 2 libraries with the same function name (or you have a function with that name) it becomes ambiguous which one you are using. You won't lose points in this class for doing this but you may have bugs that are very hard to find!

## 3 Variables and Data Types

Variables are a way of storing information in memory for later use. Variable names must start with a letter but can also contain numbers and underscores (technically they can start with underscores as well but that has more meaning). Variable names are case sensitive. I highly recommend using meaningful variable names in your programs (call it "Temperature" not "T").

In programming a =sign indicates you are assigning a value to a variable. Think of

```
>>>> a=4
```

as $a \leftarrow 4$. Note that $4 = a$ makes no sense because 4 cannot be a variable.

```
>>> a=4
>>> a
4
>>> 4=a
  File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> b=a+5
>>> b
9
>>> a=a+5
>>> a
9
>>> b
9
>>> type(a)
<type 'int'>
```

Variables have types that denote what kind of information they contain

List of common data types:

- int -integers
- float - decimal point numbers
- string - a set of characters
- dict - a variable that uses keywords to reference values
- list - a group of values
- array - Similar to a list but designed for mathematical problems

```
>>>> type(c)
<type 'float'>
>>> d= 'I am a string'
>>> d
'I am a string'
>>> d+'hi'
'I am a stringhi'
```

Strings can be defined with either ' or " in python. Strings can be comined with a simple +.

```
>>> TestList=[9,8,7,"hi"]
>>> type(TestList)
<type 'list'>
>>> TestList
[9, 8, 7, 'hi']
>>> TestList[0]
9
```

```
>>> TestList[3]
'hi'
>>> TestList[-1]
'hi'
>>> TestList[1:3]
[8, 7]
>>> TestList[1:-1]
[8, 7]
>>> TestList[1:]
[8, 7, 'hi']
>>> TestList+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

Lists do not have to all contain the same type of data. Each element of a list can be accessed via an index starting with 0. You can also access a group of elements using start:end, where the end index is not included. You can also use negative indicates to reference from the end of the list. Note because lists can contain different data types you can't perform normal operations on them. **Lists can also have some unexpected behavior if you are not careful!**

```
>>> weird=TestList
>>> weird[0]=5
>>> weird
[5, 8, 7, 'hi']
>>> TestList
[5, 8, 7, 'hi']
```

```
>>> TestDict={'Number':5, 'Name':'John'}
>>> TestDict['Number']
5
>>> TestDict.keys()
['Number', 'Name']
```

Dicts are similar to lists only instead of referencing each value with an index, you find them with a key.

The primary data type we will use in this class is arrays.

```
>>> test_arr=np.array([4,5,6])
>>> test_arr[0:2]
array([4, 5])
>>> test_arr[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 0 with size 3
>>> test_arr[2]
6
>>> test_arr=test_arr+1
>>> test_arr
array([5, 6, 7])
```

Array elements can be accessed like lists. They can also have mathematical operations act on them (assuming all elements are numbers).

## 4 Some helpful functions

- **range(stop), range(start,stop[,step])**: range creates a *list* of integers. Note that range is non-inclusive on the stop value. This is often used for loops (next lecture).

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

- **len(object)**: *len* will return the length of a list or array passed to it.

```
>>> a=range(2,5)
>>> a
[2, 3, 4]
>>> len(a)
3
```

- **np.arange(stop), np.arange(start,stop[,step])**: Same as range only returns an array instead of a list and can handle non-integer values.

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

- **np.linspace(start,stop[,num=50])**: Creates an array from start to stop (inclusive) with *num* values. By default *num=50.* This is one of the best ways to setup arrays.

```
>>> np.linspace(2,5)
array([ 2.        ,  2.06122449,  2.12244898,  2.18367347,
2.24489796,
        2.30612245,  2.36734694,  2.42857143,  2.48979592,
2.55102041,
        2.6122449 ,  2.67346939,  2.73469388,  2.79591837,
2.85714286,
        2.91836735,  2.97959184,  3.04081633,  3.10204082,
3.16326531,
        3.2244898 ,  3.28571429,  3.34693878,  3.40816327,
3.46938776,
        3.53061224,  3.59183673,  3.65306122,  3.71428571,
3.7755102 ,
        3.83673469,  3.89795918,  3.95918367,  4.02040816,
4.08163265,
        4.14285714,  4.20408163,  4.26530612,  4.32653061,
4.3877551 ,
        4.44897959,   4.51020408,   4.57142857,   4.63265306,
4.69387755,
        4.75510204,   4.81632653,   4.87755102,   4.93877551,
5.         ])
>>> np.linspace(2,5,5)
array([ 2.   ,  2.75,  3.5  ,  4.25,  5.   ])
```

- **np.logspace(start,stop[,num=50])**: Save as linspace only values are evenly spaced on a log scale (as opposed to a linear scale).

```
>>> np.logspace(2,5,5)
array([     100.          ,     562.34132519,     3162.27766017,
          17782.79410039,   100000.          ])
```

- **np.pi, np.e, np.exp**: Numpy also includes values for $\pi$ and $e$. However you should use *np.exp(value)* over *e**value* for both readability and speed.

```
>>> np.pi
3.141592653589793
>>> np.e
2.718281828459045
>>> np.exp(1)
2.7182818284590451
```

- **shape method**: In python all variables are objects that have attributes and methods (for this class we will mostly ignore this side of python). One of the most helpful attributes is shape which gives the size of a ndarray in each direction.

```
>>> a=np.linspace(2,5,5)
>>> a.shape
(5,)
>>> test=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> test.shape
(3, 3)
```