# Python Stuff

June 12, 2018

# Chapter 1

# Useful Things To Know About Python

## 1.1 Types of Variables

Telling a computer how to store different types of information is vital to programming. A computer will see an input, 7, 7.0, and '7' all as different things. Languages like C and Java are "statically typed" meaning you have to explicitly tell the computer how to store different types of information. Python has the advantage, and curse, of being "dynamically typed", meaning Python will just know what kind of information you are giving it, most of the time. The advantage to this is that it save you, the programmer, time and effort. You don't have to time explicitly declaring each variable. Python can also jump between different variable types with ease. This saves you time and can make the program easier to read. The disadvantage is that trivial bugs that are normally caught by the compiler are not noticed until the program runs into them. This can be a problem for programs that take a long time to run, a program in C will tell you theres a problem with variable types as soon as you hit 'run' but a program in Python wont let you know until it runs everything before the line were the problem is, sometimes this is hours after you initially ran the program.

In Python there are four main types of variables,*int*, *float*, *string*, and *bool*.

- *int*: These are integer numbers with no decimal points 1, 10, 256. Python will automatically store any input number as an *int* if there is no decimal point.

```
x = 2
type(x)
        int
```

- *float*: These are numbers with decimal points 1., 3.1415, 2.7182. Python will automatically store numbers with decimal points as floats even if there are no number after the decimal, but it's good practice to add a zero after the decimal for clarity. [1]

```
y = 3.
type(y)
        float
```

- *string*: This is just text. String are denoted with either single or double quotations marks.

[1]Older versions of Python (2.X) could not automatically convert variables between *int* and *float*. If you tried to take x/y you would get 0. In any version of Python 3.X you will get the correct answer of 0.66667

```
a = 'This is a string with single quotation marks'
b = "This is also a string, with the double quotation marks"
```

- *bool*: boolean variables only have two possible values. True/False or 1/0. Booleans are used to initiate loops. If the condition is True then the command inside the loop with be performed, if the condition is False then the loop is skipped. Similar to how Python can jump between *int* and *float* without difficulty, Python can jump between a *bool* and an *int* when starting a loop without a problem.

```
n = 1
if n:
        print('Hello World')

i = False
if not i:
        print('Goodbye World')

Hello World
Goodbye World
```

## 1.2 Storing Variables

There are multiple ways to tell Python how to keep track of different variables. The three most common ways of handling variables are with *lists, tuples*, and *dictionaries.* While these are very similar there are some subtle but important differences.

- *list*: A list is the most common way of storing variables. Notated with square brackets [ ] a list can have any type of variable in each element.

  ```
  foo = ["Bob", 3, 4.0, True]
  ```

  I can change any element of a list with,

  ```
  foo[0] = 'Sally'
  print(foo)
          ['Sally', 3, 4.0, True]
  ```

  An important thing to remember is that indexing starts at 0. So foo[0] will return 'Bob' foo[1] will return 3 and so on. This can cause some confusion because if you check the length of a list you'll get the number of elements in a list.

  ```
  len(foo)
          4
  ```

  But if you try to call foo[4] you'll get the error "IndexError: list index out of range"

- *tuple*: A tuple, notated with parentheses ( ) works just like a list but a tuple is immutable (cannot be changed). If we tried this will a tuple we would get an error,

```
foo2 = ('Bob', 3, 4.0, True)
foo2[0] = 'Sally'
        TypeError: 'tuple' object does not support item assignment
```

The immutability of tuples make them perfect when you want to use a list but want to make sure that parts of the list are not changed or removed.

- *dictionary/dict*: Dictionaries are unique in that you can store variables with an associated 'keys', rather than an index element. Dictionaries are denoted with curly brackets { }. The general structure for a dictionary is {key: value, key2: value2, ... }

```
foo3 = {'Test Avg': 74, 'HW': [10,4,9,8,0], 'Grade': 'B-'}
foo3['Grade']
        'B-'
```

### 1.2.1 List vs. Array

There is an important distinction between the standard Python list and the numpy library array. While they function almost the same you cannot perform mathematical operations on a list. If you have a list of numbers and you try to perform some kind of mathematical operation to it you will be an error,

```
x = [1,2,3]
x / 2
        TypeError: unsupported operand type(s) for /: 'list' and 'int'
y = np.array([1, 2, 3])
y / 2
        array([0.5, 1. , 1.5])
```

You should try to use lists when you can as then are generally faster than the numpy arrays.

## 1.3 Importing Libraries and Functions

While Python does have many useful functions built in, most mathematical functions are not readily available. Python doens't know what to do with trig functions like sin and cos. To be able to do this we need to import extra libraries to handle special tasks. There are multiple ways to import a library. The simplest way is to use the **import** command.

```
import numpy
```

Once a library is imported functions in that library can be called by typing *library.function*

```
numpy.sin(numpy.pi/2)
        1.0
```

Writing out *numpy* every time can become a little cumbersome. We can use the **import** command to make an abbreviation for any functions called in that library.

```
import numpy as np
np.sin(np.pi/2)
        1.0
```

What you decide to call the library **as** is entirely up to you, I could have just as easily done **import** numpy **as** pepper and then used pepper.sin(). You'll find that certain libraries have a common notation,

```
import math as m
import numpy as np
import matplotlib.pyplot as plt
```

This can still be painful when you have a lot of functions that you're calling. You can call specific functions from a library using

```
from numpy import sin, pi
sin(pi/2)
        1.0
```

Or you can import all of the functions in the library using,

```
from numpy import *
from numpy import all
```

### 1.3.1 Why?

Why have all these different ways to import libraries? Often times libraries will have dozens if not hundreds of different functions. If you create a function that happens to have the same name as one of the functions in the imported library problems can arise. There's also the issue of speed. Running a few speed tests shows.

```
%%timeit
import numpy as np
np.sin(np.pi/2)
        775 ns +/- 7.61 ns per loop

%%timeit
from numpy import all
sin(pi/2)
        1.39   s +/- 7.84 ns per loop
```

## 1.4 Magic Commands

This is an ipython specific thing. Lots can be said about them but the main ones that will be used for this class are %writefile, %run, %load, %cd, %pwd, and %ls.