# Python Basics

September 29, 2018

## 1 Python Basics: A Review

This is a basic review of python basics
Part of Datacamp's "Intro to Python for Data Science"

### 1.1 Printing (it has to begin somewhere)

```
In [1]: # Some basic printing
        print(7)
        print(7+5)
        print("Hello World")
        b = "Noah"
        print("Hello there " + b + " You're number")

        # Print something multiple times
        print("Hey " * 2)
```

```
7
12
Hello World
Hello there Noah You're number
Hey Hey
```

### 1.2 Basic Math Operators

```
In [2]: print(4**2) # The ** is exponentiation
        print(18%7) # The % is called Modulo and returns remainders
```

```
16
4
```

### 1.3 Variable Types

- int, or integer: a number without a fractional part. savings, with the value 100, is an example of an integer.

- float, or floating point: a number that has both an integer and fractional part, separated by a point. factor, with the value 1.10, is an example of a float.
- str, or string: a type to represent text. You can use single or double quotes to build a string.
- bool, or boolean: a type to represent logical values. Can only be True or False (the capitalization is important!).

You can easily find out what type of variable you're dealing with using the function `type()`

```
In [3]: a = "my first var"
        type(a)
```

```
Out[3]: str
```

### 1.3.1 Changing Variable Types

You can also change what type of value a variable is holding using:

- `str()`
- `int()`
- `bool()`
- `float()`

```
In [4]: # Make floats into strings
        savings = 100
        result = 100 * 1.10 ** 7
        print("I started with $" + str(savings) + " and now have $" + str(result) + ". Awesome

        # Convert pi_string into float: pi_float
        pi_string = "3.1415926"
        pi_float = float(pi_string)
```

```
I started with $100 and now have $194.87171000000012. Awesome!
```

## 1.4 Lists: The super variable

Lists can hold any kind of information, even those of different types

To create a list, surround information with brackets "[ ]"

```
In [5]: # area variables (in square meters)
        hall = 11.25
        kit = 18.0
        liv = 20.0
        bed = 10.75
        bath = 9.50
```

```
        # Create list areas
        areas = [hall,kit,liv,bed,bath]

        # Print areas
        print(areas)
```

```
[11.25, 18.0, 20.0, 10.75, 9.5]
```

```
In [6]: # Lists can also contain lists
        A = [1, 3, 4, 2]
        B = [[1, 2, 3], [4, 5, 7]]
        C = [1 + 2, "a" * 5, 3]
        print(A)
        print(B)
        print(C)
```

```
[1, 3, 4, 2]
[[1, 2, 3], [4, 5, 7]]
[3, 'aaaaa', 3]
```

```
In [7]: # Nicely organize data
        house = [["hallway", hall],
                 ["kitchen", kit],
                 ["living room", liv]]
        print(house)
```

```
[['hallway', 11.25], ['kitchen', 18.0], ['living room', 20.0]]
```

### 1.4.1 Subsetting Lists

Python uses 0-based indexing, so the first element is called the zeroth element. Index lists with brackets "[ ]"

```
In [8]: # Create the areas list
        areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "ba

        # Print out second element from areas
        print(areas[1])

        # Print out last element from areas
        print(areas[-1])

        # Print out the area of the living room
        print(areas[5])
```

```
11.25
9.5
20.0
```

"Slicing" is a indexing an array of elements. It's as so: `my_list[start:end]` All elements from start until BUT NOT INCLUDING end will be included in the subset

```
In [9]: x = ["a", "b", "c", "d"]
        x[1:3]

Out[9]: ['b', 'c']
```

```
In [10]: # Create the areas list
         areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "ba

         # Use slicing to create downstairs
         downstairs = areas[0:6]

         # Use slicing to create upstairs
         upstairs = areas[6:11]

         # Print out downstairs and upstairs
         print(downstairs)
         print(upstairs)
```

```
['hallway', 11.25, 'kitchen', 18.0, 'living room', 20.0]
['bedroom', 10.75, 'bathroom', 9.5]
```

Without spcifying either start or end while slicing, python will include the rest of the elements

```
In [11]: x = ["a", "b", "c", "d"]
         print(x[:2])
         print(x[2:])
         print(x[:])
```

```
['a', 'b']
['c', 'd']
['a', 'b', 'c', 'd']
```

```
In [12]: # Alternative slicing to create downstairs
         downstairs = areas[:6]

         # Alternative slicing to create upstairs
         upstairs = areas[6:]
```

We also have to know how to subset lists within lists

```
In [13]: x = [["a", "b", "c"],
              ["d", "e", "f"],
              ["g", "h", "i"]]
         print(x[2][0])
         print(x[2][:2])
```

```
g
['g', 'h']
```

Gotta also know how to assign new values and add on to the list!

```
In [14]: # Correct the bathroom area
         areas[-1] = 10.50

         # Change "living room" to "chill zone"
         areas[4] = "chill zone"
```

```
In [15]: # Add poolhouse data to areas, new list is areas_1
         areas_1 = areas + ["poolhouse", 24.5]
         print(areas_1)

         # Add garage data to areas_1, new list is areas_2
         areas_2 = areas_1 + ["garage", 15.45]
         print(areas_2)
```

```
['hallway', 11.25, 'kitchen', 18.0, 'chill zone', 20.0, 'bedroom', 10.75, 'bathroom', 10.5, 'po
['hallway', 11.25, 'kitchen', 18.0, 'chill zone', 20.0, 'bedroom', 10.75, 'bathroom', 10.5, 'po
```

### 1.4.2 Deleting List Elements

The function `del()` allows you to delete elements within a list

```
In [16]: del(areas[-4:-2])
         print(areas)
```

```
['hallway', 11.25, 'kitchen', 18.0, 'chill zone', 20.0, 'bathroom', 10.5]
```

### 1.4.3 Copying Lists

When you make a copy of a list like so: `list_copy = list_orig`, any change that happens to `list_copy` will happen to `list_orig` as well

You can avoid this by using `[:]` or the `list()` function

```
In [17]: # Create list areas
         areas = [11.25, 18.0, 20.0, 10.75, 9.50]
         # Create areas_copy
         areas_copy = areas
```

```
        # Change areas_copy
        areas_copy[0] = 5.0
        # Print areas
        print(areas)
```

```
[5.0, 18.0, 20.0, 10.75, 9.5]
```

Now using `list()`

```
In [18]: # Create list areas
         areas = [11.25, 18.0, 20.0, 10.75, 9.50]

         # Create areas_copy
         areas_copy = list(areas)

         # Change areas_copy
         areas_copy[0] = 5.0

         # Print areas
         print(areas)
```

```
[11.25, 18.0, 20.0, 10.75, 9.5]
```

## 2 Functions

A function is a piece of reusable code aimed at solving a particular task. One of the most important functions is `help()`

```
In [19]: help(round)
         # can also execute using "?round"
```

```
Help on built-in function round in module builtins:

round(...)
    round(number[, ndigits]) -> number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
```

For the `round()` function, there are two inputs:

1. `number`
2. `ndigits`

When you see brackets "[ ]" around one of the arguments, it means it's optional.
Here are some examples below

```
In [20]: # Create variables var1 and var2
         var1 = [1, 2, 3, 4]
         var2 = True

         # Print out type of var1
         print(type(var1))

         # Print out length of var1
         print(len(var1))

         # Convert var2 to an integer: out2
         out2 = int(var2)
         print(out2)
```

```
<class 'list'>
4
1
```

You can also tell optional arguments if you see they have default values.

```
In [21]: help(sorted)
```

```
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

Function `sorted()` takes three arguments: iterable, key, sorted
If you don't specify a value for `key`, it will assume a default value of `None`. Same thing for `reverse` but it will assume default value of `False`

```
In [22]: # Create lists first and second
         first = [11.25, 18.0, 20.0]
         second = [10.75, 9.50]

         # Paste together first and second: full
         full = first + second

         # Sort full in descending order: full_sorted
         full_sorted = sorted(full, reverse = True)
```

```
        # Print out full_sorted
        print(full_sorted)
```

```
[20.0, 18.0, 11.25, 10.75, 9.5]
```

## 2.1 Methods

Functions that belong to Python objects (such as strings, floats, lists etc.)
Strings objects have methods such as `capitalize()` and `replace()`
To use Methods, use dot notation such as `my_string.capitalize()`
Methods can also change the object they're called on
You can discover more methods by calling help on an object such as `help(str)`
Some string methods include:

- `upper()`: Make all characters upper case
- `count()`: Count the amount of times a character appears in a string

```
In [23]: # STRING METHODS
         # string to experiment with: place
         place = "poolhouse"

         # Use upper() on place: place_up
         place_up = place.upper()

         # Print out place and place_up
         print(place + place_up)

         # Print out the number of o's in place
         print(place.count('o'))
```

```
poolhousePOOLHOUSE
3
```

Some List methods include:

- `index()` - Location of an element in a list
- `count()` - Amount of times a character appears in a list
- `append()` - Append input to the end of a list
- `reverse()` - Reverse order of a list

```
In [24]: # LIST METHODS
         # Create list areas
         areas = [11.25, 18.0, 20.0, 10.75, 9.50]
```

```
# Print out the index of the element 20.0
print(areas.index(20))

# Print out how often 9.50 appears in areas
print(areas.count(9.50))
```

```
2
1
```

```
In [25]: # Create list areas
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Use append twice to add poolhouse and garage size
areas.append(24.5)
areas.append(15.45)

# Print out areas
print(areas)

# Reverse the orders of the elements in areas
areas.reverse()

# Print out areas
print(areas)
```

```
[11.25, 18.0, 20.0, 10.75, 9.5, 24.5, 15.45]
[15.45, 24.5, 9.5, 10.75, 20.0, 18.0, 11.25]
```

## 2.2 Packages

Like a directory of Python scripts

Each Script is called a "Module". Each Module includes Methods, Functions and Object Types

One of the most popular packages is numpy. It can be imported with the following code:

`import numpy` Now you can use its' function `array()` as so: `numpy.array([1,2,3])`

`import numpy as np` You can shorten how much you need to type by importing numpy with the above code so now the function can be called as `np.array([1,2,3])`

`from numpy import array` You can also import specific functions from a package with the above so now the `array()` function can be used like this: `array([1,2,3])`. The downside to this approach is that others may not know you called this specific function from the numpy package

```
In [26]: # Now here's the Math package

# Definition of radius
r = 0.43

# Import the math package
import math
```

```
        # Calculate C
        C = 2*math.pi*r

        # Calculate A
        A = math.pi*r**2

        # Build printout
        print("Circumference: " + str(C))
        print("Area: " + str(A))
```

```
Circumference: 2.701769682087222
Area: 0.5808804816487527
```

```
In [27]: # Definition of radius
         r = 192500

         # Import radians function of math package
         from math import radians

         # Travel distance of Moon over 12 degrees. Store in dist.
         dist = radians(r*12)

         # Print out dist
         print(dist)
```

```
40317.10572106901
```

There are a few different ways to import functions and packages. The scipy subpackage called linalg has a function called `inv()` and you have the capability to call it as:
`my_inv([[1,2], [3,4]])` by entering
`from scipy.linalg import inv as my_inv`

## 3 Numpy

Lists store values but are not efficient for doing calculations on arrays of numbers. Numpy arrays are powerful because they don't have this issue

```
In [28]: # Create list baseball
         baseball = [180, 215, 210, 210, 188, 176, 209, 200]

         # Import the numpy package as np
         import numpy as np

         # Create a numpy array from baseball: np_baseball
         np_baseball = np.array(baseball)
```

```
# Print out type of np_baseball
print(type(np_baseball))
print(np_baseball*0.5)
```

```
<class 'numpy.ndarray'>
[ 90.  107.5 105.  105.   94.   88.  104.5 100. ]
```

You can also filter out arrays using boolean values/arrays. First you must have a conditional statement

```
In [29]: # Create the light array
         light = np_baseball < 200

         # Print out light
         print(light)

         # Print out BMIs of all baseball players whose BMI is below 21
         print(np_baseball[light])
```

```
[ True False False False  True  True False False]
[180 188 176]
```

```
In [30]: # These two lines of code produce the same results
         np.array([True, 1, 2]) + np.array([3, 4, False])
         np.array([4, 3, 0]) + np.array([0, 2, 2])
```

```
Out[30]: array([4, 5, 2])
```

Indexing numpy arrays work similarly to indexing lists

```
In [31]: print(np_baseball[3])
         print(np_baseball[2:5])
```

```
210
[210 210 188]
```

### 3.1 2D Numpy Arrays

If you enter `type(np_baseball)` you'll get:

```
In [32]: type(np_baseball)
```

```
Out[32]: numpy.ndarray
```

ndarray = N-Dimensional array. Thus far we've create 1D numpy arrays

Can create 2D Numpy array. You can easily think of a 2D array as a list containing multiple lists. Below is an example

```
In [33]: # Create baseball, a list of lists
         baseball = [[180, 78.4],
                     [215, 102.7],
                     [210, 98.5],
                     [188, 75.2]]

         # Create a 2D numpy array from baseball: np_baseball
         np_baseball = np.array(baseball)
         print(np_baseball)
```

```
[[180.   78.4]
 [215.  102.7]
 [210.   98.5]
 [188.   75.2]]
```

If we print `type()` of this array, it will still be a numpy array

```
In [34]: # Print out the type of np_baseball
         print(type(np_baseball))
```

```
<class 'numpy.ndarray'>
```

There's a special method for numpy arrays called `.shape` that displays the dimensions of the numpy array

```
In [35]: # Print out the shape of np_baseball
         print(np_baseball.shape)
```

```
(4, 2)
```

### 3.1.1 Subsetting 2D Numpy Arrays

If your 2D numpy array has a regular structure, i.e. each row and column has a fixed number of values, complicated ways of subsetting become very easy. Supply 2 inputs for subsetting, `[rows,columns]`

Entering `Variable[:,0]` will select all rows and the first column

```
In [36]: # Print out the 2nd row of np_baseball
         print(np_baseball[1,:])

         # Select the entire second column of np_baseball: np_weight
         np_weight = np_baseball[:,1]

         # Print out height of 4th player
         print(np_baseball[3,0])
```

```
[215.  102.7]
188.0
```

### 3.1.2 Basic 2D Arithmetic

For numpy arrays, it's very easy to perform matrix wise or element-wise operations

```
In [37]: np_mat = np.array([[1, 2],
                            [3, 4],
                            [5, 6]])
         print(np_mat * 2)
         print(np_mat + np.array([10, 10]))
         print(np_mat + np_mat)
```

```
[[ 2  4]
 [ 6  8]
 [10 12]]
[[11 12]
 [13 14]
 [15 16]]
[[ 2  4]
 [ 6  8]
 [10 12]]
```

## 3.2 Numpy Basic Statistics

For large data, generating summarizations is necessary

There's a mean function: `np.mean()`
There's a median function: `np.median()`
Standard Deviation: `np.std()`
Correlated Coefficients: `np.corrcoef(x,y)`

There is also a numpy `sort()` and `sum()` functions that run faster because they're operating on a single data type!

For easy data simulations, you can generate pseudo-random numbers

Ex: >Create 5000 data points from a normal distribution with mean 1.75 and SD 0.20. Round all values to 2 decimal places:

```
height = np.round(np.random.normal(1.75,0.20,5000),2)

weight = np.round(np.random.normal(60,0.20,5000),2)
```

Then I can stack the columns into a single Numpy Array

```
np_dataset = np.column_stack((height,weight))
```

```
In [38]: np.random.normal(10,1,5)
```

```
Out[38]: array([ 9.12651334,  9.84474543, 10.25689553, 11.20858775,  9.1085513 ])
```

Now here's an example of summarizing data

```
In [39]: # Create the dataset
         h = np.round(np.random.normal(1.75,0.20,1015),2)
         w = np.round(np.random.normal(60,0.20,1015),2)
         a = np.round(np.random.normal(30,5,1015),2)
         np_baseball = np.column_stack((h,w,a))

         # Now for some basic summarizations
         # Create np_height from np_baseball
         np_height = np_baseball[:,0]

         # Print out the mean of np_height
         print("Mean is: ",str(np.mean(np_height)))

         # Print out the median of np_height
         print("Median is: ",str(np.median(np_height)))

Mean is:  1.7381182266009854
Median is:  1.73


In [40]: # Now a little fancier
         # Print out the standard deviation on height. Replace 'None'
         stddev = np.std(np_baseball[:,0])
         print("Standard Deviation: " + str(stddev))

         # Print out correlation between first and second column. Replace 'None'
         corr = np.corrcoef(np_baseball[:,0],np_baseball[:,1])
         print("Correlation: " + str(corr))

Standard Deviation: 0.19608610206102214
Correlation: [[1.         0.01653367]
 [0.01653367 1.        ]]
```