Module P01: Python Basics

The Python training modules will be mainly hands-on sessions using **Jupyter notebooks**. A Jupyter notebook is a nice way of learning Python on the browser via an integrated service running the IPython kernel. You will see a mix of texts (to guide your learning) and executable 'cells' containing Python code. Of course, to work on an executable Python file (.py), you can also choose to write with **Spyder IDE**, which also comes with the Anaconda suite.

Contents:

Programs

Most machines are designed to do one thing. For example, a fan is designed to blow air.

Because they are designed to do one thing only, most machines are limited in what they can do. You can try and vary their basic functionality, but you will not get very far. If you want to change a fan into something else, like a dynamo, you have to alter the design of the machine in order to get it to do something else.

> "A computer does two things: perform calculation & remember the results."
> * - John V. Guttag*

With those two things, you can run any process, which is defined as a very precise sequence of steps. Why do processes need to be precise?

If you can run any process, well, you can run any algorithm, because an algorithm is just a process that always finishes. You can also run any program, which is just a fancy name for a collection of interlinked processes.

And that's it. Your computer does two things, but it's designed to do one: run programs. To write programs, you have to learn programming, and that will be the focus of the Python Module.

Python

Why Python?

1. It’s easy to learn

- Now the language of choice for 8 of 10 top US computer science programs (Philip Guo, CACM)

2. Full featured

- Not just a statistics language, but has full capabilities for data acquisition, cleaning, databases, high performance computing, and more

3. Strong Data Science Libraries

- The SciPy Ecosystem

It's important to note that Python isn't the only way that we could teach data science. We can teach data science in R, in Excel, in any number of software packages. That's because Python is a tool that data scientists use, but the concepts are more important for you to understand.

A computer's purpose is to store (and manipulate) a sequence of instructions, and has a set of elements that will execute any instruction in that sequence. Once we program using instructions and data objects at the level of the machine (e.g., move 64 bits of data from this location to that location) then we do **low-level programming**.

If we program using more *abstract* operations (e.g., pop up a menu on the screen) that have been provided by the language designer then we do **high-level programming**. Since you don't have to tell Python how to move bits and bytes around, when you write you are doing something abstract - your computer understands something else.

The Python *interpreter* is a program that interprets the programs that we write in Python by running a program that a computer understands directly. The language that your computers understand directly is called *binary*. Python is a **language**, not a program in itself, although the *interpreter* is.

So Python is an *abstract* **high-level language**.

Your first program: Print!

Python can handle strings of letters that are enclosed in quotation marks like:

```
"Hello Class" or 'Hello World'
```

One of the simplest programs is print, which works like the following:

In [1]:

```
print("'Hello Class!'")
```

```
'Hello Class!'
```

In [2]:

```
# Write your first bit of code: print Hello World!
# How is the print function abstract?
print("Hello World!")
print("Hello")
print("123")print("'Hello Class!'")
print("More stuff here")
print("5342435343253fdaf;jlakf;lkfladkfad")
```

```
Hello World!
Hello
123
More stuff here
5342435343253fdaf;jlakf;lkfladkfad
```

Arithmetic

Python can also handle numbers, and basic arithmetic using the following functions.

Symbol	Task Performed
+	Addition
-	Subtraction
/	division
%	modulo
*	multiplication
//	floor division
**	to the power of

In [3]:

```
print(3)
print(3*5)
print(3+3)
print(3-2)
print(8/2)
print(2**5)
print(5/2)
print(5//2)
```

```
3
15
6
1
4.0
32
2.5
2
```

Parentheses work too! And they are used similarly to arithmetic rules.

In [4]:

```
print(2+3*4)
print((2+3)*4)
```

```
14
20
```

Practice: Some Basic Arithmetic Exercises

In [5]:

```
# How many seconds are there in a year?
print(365*24*60*60)

# you can write some comments here
```

```
31536000
```

In [6]:

```
# What is the duration of our 6-day class in minutes?
print(6*7*60)
```

```
2520
```

In [7]:

```
# Find 1/2
print(1/2)
```

```
0.5
```

Integers vs. Float

For Python 2.x, it interprets 1/2 = 0, while Python 3.x the result will be 0.5

Python allows you to work with both integers (int) and numbers containing decimals (float). Of course, every integer can be represented as a float, but there are some things you can do with integers that you cannot do with floats that mostly have to do with precision. Python can tell automatically if a number is an int or a float, though you can override this as you see fit.

In [8]:

```
print(1/2)
```

```
0.5
```

In [9]:

```
print(type('c'))
print(type("hello"))
print(type(10))
print((type(10.)))
print((10 / 3))
print((10 % 3))
print((float(10) / 3)) # Force integers to be float
print((13.0 / 3.5)) # You can also change the initial data type.
print((13.0 //3.5)) # Floor Division, rounds down the result to the nearest integer
print((round(7.5))) # You can round floats up to the neares integer.
```

```
<class 'str'>
<class 'str'>
<class 'int'>
<class 'float'>
3.3333333333333335
1
3.3333333333333335
3.7142857142857144
3.0
8
```

Int arithmetic

Arithmetic with integers works the same way as normal arithmetic, except for division. When you divide one integer by another, you drop the remainder. To get the remainder, you can use the % (mod) operation. To get the exact result, however, you need another data type - you turn one of the integers into a float. In general, if you perform an arithmetic operation with one integer and one float, Python will turn both numbers into floats before performing the operation.

Quick Exercise 1 Let's convert a temperature reading from Fahrenheit to Celsius. The "formula" that you would be given looks something like this:

$$C = \frac{(F - 32) \times 5}{9}$$

In [10]:

```
# Quick Exercise 1
# say F = 100


```

```
37.77777777777778
```

In [11]:

```
# Do the inverse. Celsius to Fahrenheit


```

```
100.0
```

Float arithmetic

In [12]:

```
# Examples:
print(5.5 + 2.2)
print(5.5 - 2.2)
print(5.5 * 2.2)
print(5.5 / 2.2)
print(5.5 // 2.2)
print(5.5 ** 2.2)
print(5.5 < 2.2)
print(5.5 != 2.2)
print(5.5 == 5.5)
print(5.5 >= 2.2)
print("hello" == "hi")
print("hello" == "hello")
```

```
7.7
3.3
12.100000000000001
2.5
2.0
42.540042248725975
False
True
True
True
False
True
```

Variables

A name that is used to denote something or a value is called a variable. This is how a computer *stores* data, by assigning it to a variable! In python, variables can be declared and values can be assigned to it as follows:

In [13]:

```
b = -10
c = 2.7

print(b)
print(c)
print((b + c))

print((type(b)))
print((type(c)))
```

```
-10
2.7
-7.3
<class 'int'>
<class 'float'>
```

In [14]:

```
# Note how changing the type to an int rounds down
b = float(b)
c = int(c)

print(b)
print(c)

print((type(b)))
print((type(c)))
```

```
-10.0
2
<class 'float'>
<class 'int'>
```

Multiple variables can be assigned with the same value.

In [15]:

```
x = y = 1
print((x,y))

# (1, 1) <-- this is called a TUPLE
```

```
(1, 1)
```

In [16]:

```
id(x)
```

Out[16]:

```
140717651076896
```

In [17]:

```
id(y)
```

Out[17]:

```
140717651076896
```

Tip: If you need a quick reference what a built-in function does, or how to use the function, place your cursor on the function name and press Shift-Tab. You will see a pop-out description of the function!

In [18]:

```
y = y+1
print(y)
print(x)
```

```
2
1
```

In [19]:

```
# Multiple Assignment - What does this do?
x,y = 2,3
x,y = y,x
print((x,y))
```

```
(3, 2)
```

Follow the following variable naming rules:

- Variable names can only contain letters, numbers, and underscores. Variable names can start with a letter or an underscore, but not a number.
- Spaces are not allowed in variable names, so we use underscores instead of spaces. For example, use student_name instead of "student name".
- You cannot use Python keywords as variable names.
- Variable names should be descriptive, without being too long. For example mc_wheels is better than just "wheels", and number_of_wheels_on_a_motorycle.
- Be careful about using the lowercase letter l and the uppercase letter O in places where they could be confused with the numbers 1 and 0.

Practice using variables

In [20]:

```
# How fast does the speed of light travel in one nanosecond?
# Give your answer in seconds, and assign it to a new variable called nanostick

speed_of_light = 299792458 # meters per second
```

In [21]:

```
# How far does light travel in the time that your computer does one cycle?
print(1/2400000000)
print(speed_of_light/2400000000)
```

```
4.166666666666667e-10
0.12491352416666666
```

In [3]:

```
# Variables can vary!
# Guess the final value of guess_this_number of this code:

guess_this_number  = 17
guess_this_number = guess_this_number/2
guess_this_number = guess_this_number - 5

# you can print to get the answer :)
print(guess_this_number)
```

```
3.5
```

In [1]:

```
# Guess the output of this code:
minutes = 5
minutes = minutes + 10
seconds = minutes / 2
print(seconds)
```

```
7.5
```

In [4]:

```
# Some Jupyter Magic on variables
%whos
```

```
Variable            Type     Data/Info
-------------------------------------
guess_this_number   float    3.5
minutes             int      15
seconds             float    7.5
```

In [25]:

```
# You can also perform command line executions from within the Python kernel. Add a ! in
!dir
```

```
 Volume in drive D is Documentation
 Volume Serial Number is B8E9-4C1C

 Directory of D:\Forward School Trainer\NitroDegree_DataScience\W_1\Notebo
oks

10/20/2020  11:24 AM    <DIR>          .
10/20/2020  11:24 AM    <DIR>          ..
10/01/2020  02:37 PM    <DIR>          .ipynb_checkpoints
10/20/2020  11:24 AM            29,972 P01 - Python Basics.ipynb
10/01/2020  02:35 PM            17,519 P02 - Logic and Repetition.ipynb
10/01/2020  02:39 PM            24,041 P03 - Data Structures I.ipynb
               3 File(s)         71,532 bytes
               3 Dir(s)  115,994,734,592 bytes free
```

Strings

A string is a set of characters. Strings are used to store data that is *text*.

In Python, Strings are abbreviated "str" and there are many built-in features that allow you to manipulate those strings.

Strings can be enclosed by either double or single quotes, although single quotes are more common. Question: Why would you want to use double quotes on occasion?

In [6]:

```
# Which of these are valid ways of printing strings?
print('hello')
print("hello')
print('hello)
print('hello")
print("hello")
print "hello"
print hello
```

```
  File "<ipython-input-6-e27b93dea28d>", line 8
    print hello
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(he
llo)?
```

In [14]:

```
# Define a variable as your name.  Then print "My name is _____" using your variable.
name = "Spiderman"
print("My name is", name)
print("My name is " + name)
print("My name is %s"%name)
print("My name is " + name + ".")
```

```
My name is Spiderman , ,
My name is Spiderman
My name is Spiderman
My name is Spiderman,,
```

In [8]:

```
# String ASCII demo
ord('A')
```

Out[8]:

```
65
```

A side note:

One of the most useful tools for a curious programmer is the documentation, or docs that explain the functionality of a particular python type, function, or otherwise. Take a look at the Python Documentation on Strings (https://docs.python.org/3/library/stdtypes.html#str).

String Operations

s + t -- the concatenation of s and t

n * s -- equivalent to adding s to itself n times

In [16]:

```
b = 'French' # Concatenation works quite intuitively
print((b + ' Fries'))
```

```
French Fries
```

In [26]:

```
pi = 3.14
print(('pi is approximately: ' + 3.14)) # why does this throw an error?
```

```
pi is approximately: pi
```

In [33]:

```
a = 'oleh '
print((a*2 + 'Malaysia b' + a))
print((a*2, 'Malaysia b', a))
```

```
oleh oleh Malaysia boleh
('oleh oleh ', 'Malaysia b', 'oleh ')
```

In [30]:

```
from datetime import datetime, timezone
# str(datetime.now())
str(datetime.now(timezone.utc)  )
```

Out[30]:

```
'2021-07-30 01:55:17.111118+00:00'
```

Slices of String Indices

Characters in a string can be accessed using the standard [ ] syntax, and like Java and C++, Python uses zero-based indexing. This concept will extend much further than strings.

This is easiest to think about if you think of each letter as being in its own box.

s[1:4] is 'ell' -- chars starting at index 1 and extending up to but not including index 4

s[1:] is 'ello' -- omitting either index defaults to the start or end of the string

s[:] is 'Hello' -- omitting both always gives us a copy of the whole thing (this is the pythonic way to copy a sequence like a string or list)

s[1:100] is 'ello' -- an index that is too big is truncated down to the string length

s[-1] is 'o' -- last char (1st from the end)

s[-4] is 'e' -- 4th from the end

s[:-3] is 'He' -- going up to but not including the last 3 chars.

s[-3:] is 'llo' -- starting with the 3rd char from the end and extending to the end of the string.

h e l l o
0 1 2 3 4
-5 -4 -3 -2 -1

s[i] -- ith item of s, origin 0

s[i:j] -- slice of s from i to j

In [38]:

```
#[added indexing example]
a = 'text string here' #Strings are contained by either single or double quotes.

print(a)
print(("The first character in our string is " + a[0] +"."))
print(("The second, third, and fourth characters are " + a[1:4] +"."))

s = 'hello'

print((s[-1]))
print((s[:-3]))
print((s[-3:]))
```

```
text string here
The first character in our string is t.
The second, third, and fourth characters are ext.
o
he
hello
```

Quick Exercise 2 Let's do some mental exercises without running the following cells...

In [70]:

```
# Which pairs of slices have the same results?
s = "Good morning"

print(s[3], s[1+1+1])
print(s[0], (s+s)[0])
print(s[0]+s[1], s[0+1] )
print(s[1], (s+ ' ed')[1])
print(s[-1], (s+s)[-1])
```

```
d d
G G
Go o
o o
g g
```

In [45]:

```
# Which slices will return the string s exactly as it is?

print(s[:])
print(s+s[0:-1+1])
print(s[:-1])
print(s[0:])
print(s[:3]+s[3:])
```

```
Good morning
Good morning
Good mornin
Good morning
Good morning
```

String Indices related methods

len(s) -- length of s

s.index(x) -- index of the first occurrence of x in s

s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found

In [49]:

```
data_vs_info = "You can have data without information, but you cannot have information w
~ Daniel Keys Moran,  science fiction writer"

print(data_vs_info) # Note what \ does.
print((len(data_vs_info)))
print((data_vs_info.find('data')))
print((data_vs_info.index('data')))

# Slice the name of the author
# Slice the profession.
# Slice the quote only
```

```
You can have data without information, but you cannot have information wit
hout data. ~ Daniel Keys Moran,  science fiction writer
129
13
4
```

TASK 1 You are given a quote. Combine the find/index functions and string slicing to return:

- Just the first part of the quote
- Just the second part of the quote
- Just the author's name.

In [50]:

```
# the quote
pythagoras = "There is geometry in the humming of the strings, there is music in the spa
```

In [41]:

```
# Which of these will evaluate to -1?
print("test".find("t"))
print("test".find("st"))
print("Test".find("te"))
print("west".find("test"))
```

```
0
2
-1
-1
```

In [74]:

```
# Which of these will always return the value zero for any given string s?
print(s)
print((s.find(s[2:3])))
print((s.find("s")))
print(("s".find("s")))
print((s.find("")))
print((s.find(s+"!!!")+1))
```

```
Good morning
1
-1
0
0
0
```

In [75]:

```
# How to look for help.  Equivalent to looking at https://docs.python.org/2/library/stri
help("str.find")
```

```
Help on method_descriptor in str:

str.find = find(...)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

In [ ]:

```
# Write your code here
```

Common built- in methods

min(s) -- smallest item of s

max(s) -- largest item of s

s.count(x) -- total number of occurrences of x in s

s.lower(), s.upper() -- returns the lowercase or uppercase version of the string

s.strip() -- returns a string with whitespace removed from the start and end

s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes

s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string

s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new'

s.split('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.

s.join(list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

In [78]:

```
s = 'abcdefghijklMnoPqrstuvwxyz'
t = 'This is a fox, it is hungry, it is running home.'
split_already = t.split(',')
print(split_already)
print(split_already[0])
print(split_already[1])
print(split_already[2])
u = ','
print(u.join(split_already))
```

```
['This is a fox', ' it is hungry', ' it is running home.']
This is a fox
 it is hungry
 it is running home.
This is a fox, it is hungry, it is running home.
```

Format

The % operator takes a printf-type format string on the left (%d int, %s string, %f/%g floating point), and the matching values in a tuple on the right (a tuple is made of values separated by commas, typically grouped inside parenthesis)... more about tuples later.

In [45]:

```
# % operator
print("We were going nearly %.3f km/hr. But there were %d %s going faster!"%(130.2, 3, "
```

```
We were going nearly 130.200 km/hr. But there were 3 cars going faster!
```

In [80]:

```
print("Our %s class ends at %d pm."%("Python", 3))
```

```
Our Python class ends at 3 pm.
```

In [81]:

```
# format method
"We were going nearly {:.2f} km/hr. But there were {} {} going faster!".format(130.2666,
```

Out[81]:

```
'We were going nearly 130.27 km/hr. But there were 3 cars going faster!'
```