Introduction to Python

#### Variables, conditionals, functions

Author: W.P.G.Peterson

Assignment Contents

EXPECTED TIME: 2.5 HRS

Overview

This assignment is designed to get you up and running with `Python` using `Jupyter Notebooks`. Initially the assignment proceeds in a measured fashion - showing and explaining syntax explicitly, offering lists of commonly used features and functions of both `Python` and `Jupyter`. Most of the material will be a review of the lectures from this week.
Eventually, the pace picks up - less is explained and you are required to either search for answers or already know them. This is by design. Initially the goal is to make you comfortable in **this** particular `Python` environment, but after that, the assignment switches to predominantly testing `Python` knowledge gained in lecture rather than teaching or re-teaching concepts.

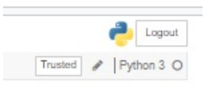
The concepts used in this assignment are quite basic, so, after your lectures, hopefully you are fluent enough with the concepts that you can focus on becoming comfortable with this form of assesment and this coding environment.

Activities in this Assignment

- Assigning and creating variables
- Use string indexing
- Use string methods
- Build simple functions
- Use if/else/elif control flow

Introduction to Jupyter Notebook

If you are reading this line, it hopefully means you have successfully opened up a `Jupyter Notebook`. If you are simply viewing this notebook and do not have a `Python 3` Kernel running [and do not see something like this:



on the upper right corner of your screen] seek technical help!

`Jupyter Notebook` offers plenty of functionality which can be extended with plug-ins and and other modifications, but this lesson will only review basic functionality.

First: what are we looking at?
Below is an empty code cell

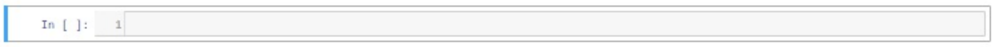
Note the `In [ ]` to its left - thats the clearest indication that a cell is ready to read and execute code.

These words, on the other hand, are appearing in a `markdown` cell. Below is an empty `markdown` cell, and below that is an empty `markdown` cell that has been executed.

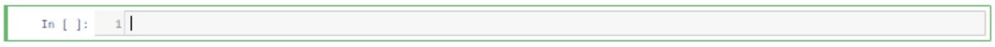
`Markdown` cells can be used to integrate text, figures, etc. into your code. It is also possible to use `LaTeX` in `markdown` cells to control formatting.

When navigating through the various cells in a `Notebook`, you will use two modes: `Command Mode` and `Edit Mode`.

In `Command Mode`, cells will feature a blue border:



`Edit Mode` features a green border:



While in `Command Mode`, double-clicking on a cell or hitting `enter` will activate `Edit Mode`. In `Edit Mode`, hitting `esc` returns to command mode.

A full list of keyboard shortcuts available in both `Command` and `Edit` modes can be found under the `<Help>` menu at the top of your screen. Alternatively, the list can be accessed by hitting `<h>` while in `Command Mode`.

A few important `Command Mode` shortcuts:

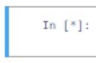
- `<a>` creates a new cell [a]bove the current cell
- `<b>` creates a new cell [b]elow the current cell
- `<x>, <c>, <v>` will cut, copy and paste cells (mutiple cells selected with `<shift-click>`
- `<d , d>` will delete the selected cell(s)
- `<s>` or `<ctrl-s>` will save the notebook
- `<i , i>` will interrupt the current process [e.g. when a process is taking too long or there is some issue]

In `Edit Mode` the important shortcuts are:

- `<ctrl-enter>` runs the current cell.
- `<shift-enter>` runs the current cell and highlights the next cell.
- `<alt-enter>` runs the current cell and creates a new cell below.

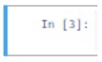
For example, using this cell practice sw itching betw een `Command` and `Edit` mode; Running the cell, and creating new cells.

During execution of a cell you w ill see the `In [ ]` change to:



```
In [*]:
```

And after completion, the number next to the `In` w ill increment:



```
In [3]:
```

Indicating a cell has run.

Onto Python

Python can simply be thought of as a big calculator. As a calculator, it features a number of built-in operators:

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division
- `**` for exponents, e.g. `2**4` is $2^4$
- `//` for floor division e.g. `9/4 = 2.25; 9//4 = 2; 9/2 = 4.5, 9//2 = 4`
- `%` as the modulo operator. e.g. `9%3 = 0; 9%2 = 1; 9%4 = 1; 9%5 = 4` e.g. - n1%n2 returns the remainder from n1/n2

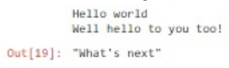
Run (`<ctrl-enter> or <shift-enter>`) the below cell for an example:

```
2+2, 1-3, 5.2*3, 9/2, 9//2, 2**4
```

Python can also print out messages to the "console" using the `print()` command.
Run the follow ing:

```
print("Hello world")
print("Well hello to you too!")
"What's next"
```

You should see something like this after running the previous cell:



```
         Hello world
         Well hello to you too!

Out[19]: "What's next"
```

The top tw o lines ("Hello w orld", and "Well hello…") are both w rapped in `print()` statements -- This means Python has been told to print each to the console.
"What's next" is **not** in a `print()` statement, yet, it has appeared in our console -- Notice the `Out [19]` appearing adjacent to "What's next". One of the convenience features of `Jupyter Notebook` (**not** Python) is that it w ill print out the *returned value* of the last line of a code cell.
Please run the below cell:

```
print("Hello world")
"What's next"
print("Well hello to you too!")
```

Notice that "What's next" does not print to the console, but the `Out[#]` has also disappeared. It is important to remember this difference betw een the behavior of *Python* and the behavior of *Jupyter Notebook*

The four basic types in Python are `Integers`, `Floats`, `Strings` and `Booleans`.

```
print(7, "is a ", type(7))
print(7.1, "is a ", type(7.1))
print("Joe", "is a ", type("Joe"))
print(True, "is a ", type(True))
```

Notice the tw o "primitve" types of numbers: `int`egers and `float`ing point.

Variable assignment

Variables may be assigned (almost) any name by the user. There are a few reserved w ords and names of builtin functions that cannot be used for variable names. In general, Python style guides suggest using `lower_case_words_connected_by_undersore_for_variable_names`.

Assignment in Python occurs using the `=` operator. As seen in the next cell

```
my_num = 1
another_num = my_num
```

```
print("my_num = ", my_num)
print("another_num = ", another_num)

print("\nChanging my_num\n")

my_num = 2

print("my_num = ", my_num)
print("another_num = ", another_num)
```

Notice here how changing the value of `my_num` did not change the value of `another_num`.

There are some cases (covered next w eek) w here the changing of one variable *can* change the value of another variable. But, as a general rule, if a variable's value is changed using the `=` operator, everything should be safe. On the other hand if a variable's value is changed via a method (covered below) then it is possible that the values of multiple variables might change.

Below w e create one variable for each type of "primitive". Watch as the `floats` and `ints` are re-cast.

```
my_int = 7
my_float = 7.0
my_float2 = 7.2
my_string = "Joe"
my_bool = True

print("my_float = ", my_float, type(my_float))
print("my_float2 = ", my_float2, type(my_float2))
print("my_int = ", my_int, type(my_int))
print("my_float == my_float2?: ", my_float == my_float2)
print("my_int == my_float?: ", my_int == my_float)
print("my_int == my_float2?: ", my_int == my_float2)

print('\nRecasting Numbers\n')
my_float = int(my_float)
my_float2 = int(my_float2)
my_int = float(my_int)

print("my_float = ", my_float, type(my_float))
print("my_float2 = ", my_float2, type(my_float2))
print("my_int = ", my_int, type(my_int))
print("my_float == my_float2?: ", my_float == my_float2)
print("my_int == my_float?: ", my_int == my_float)
print("my_int == my_float2?: ", my_int == my_float2)
```

`types` in Python (w hich as seen above can be found using the `type()` built-in function) are determined *dynamically*; this is to say that a single variable may have its type changed by the user.

The `my_float` and `my_float2` variables w ere *recast* to integers by w rapping the variable name in a call to the `int()` function.

Note how the comparison betw een the `floats` and `int` changed -- the recasting of `7.2` as an `int` changed it's value to a `7`, recasting the `7.0` as an `int` w as also changed to `7`. It is important to remember that naive recasting *might* change an underlying value.

Assignment Grading Example:

Having covered "Python as a calculator" and variable assignment, w e are ready for our first graded code cell.

These assignmets w ill be graded automatically. Thus it is very important to follow submission instructions very carefully.

The below cell w ill be graded. The full question and answ er code is already provided so *if you want credit for this question, do not change anything*

Question 1:

```
### GRADED
### Example question
### Assign the value of 15 to var1; Use an exponential function to assign var2 the value of to the square root of 35.

### NB: Unlike the above code cells, you will be able to edit this cell
### YOUR ANSWER BELOW

var1 = 15
var2 = 35**.5


### Testing answers below:
print(var1)
print(var2)
```

A Couple of Easy Math Questions:

Question 2:

```
### GRADED
### Assign the value of 2683 divided by 6432 to ans1.
### YOUR ANSWER BELOW

ans1 = 2683/6432
```

Question 3:

```
### GRADED
### Find the value of 1.678 squared times 32. Assign to ans1.
### YOUR ANSWER BELOW

ans1 = (1.678 **2)* 32
```

Useful operators:

Before departing from math, a note on some useful operators:

- `x+=1` is the same as `x=x+1`
- `x*=2` is the same as `x=x*2`
- Same w ith `-` and `/`

Strings:

The "String" primitive type in Python are denoted using either single `<'>` quotation or double `<">` quotations marks.

```
string1 = "I'm a string"
string2 = 'Me too'
string3 = str(3)

print(type(string1), type(string2), type(string3))
```

Strings feature both `indexing` support as w ell as a number of useful `methods`.

Indexing

Indexing allow s a user to return a subset of the characters from a string, specified w ithin - `[]` - square brackets.

The simplest subset is a single letter

```
print("Hi"[0])
print("Joe"[2])
```

Note how `"Hi"[0]` returned the first letter of "Hi" and `"Joe"[2]"` returned the third letter of "Joe".
Python uses 0-based indexing, w hich means that the first letter is at `index 0`, the second letter is at `index 1`, and the final letter is at `index n-1` w here `n` is the total number of letters.

```
my_string = "Python"
print("Python"[0:3])
print(my_string[0:3])
print(my_string[:3])
print(my_string[::2])
```

Here, a range of letters are desired, and `<:>` appears in the brackets. As seen above, the syntax `<string>[0:3]` returned the 1st, 2nd and 3rd letters from that `<string>`.
`<string>[::2]` on the other hand returned every other letter from `<string>` from the start to finish.

The indexing notation has three positions `[start : stop : step]`.

As partially modeled in `my_string[::2]`, the `start`, `stop`, and `step` functionally have default values of `[0:n:1]` w here n is the length. Thus, w hen Python sees one of those positions empty, it substitutes the "defaults".

Finally, it is w orth noting that, as seen in `<string>[0:3]`, the character at the `start` index is returned, but **the character at the `stop` index is not returned.**

String Indexing Problems:

```
### THIS STRING IS USED FOR ASSIGNED PROBLEMS
test_string = "The quick Brown Fox Jumped over the LAZY Dog JUST THAT one too manytimes.andthedogJUST SNAPPed"
```

Question 4:

Example Problem / solution:

```
### GRADED
### Example:
### Return the 10th character of test_string. Assign to ans1.
### YOUR ANSWER BELOW

ans1 = test_string[9]

### Checking answer
print("ans:", ans1)
print(test_string[:11])
print("1234567890")
```

Question 5:

```
### GRADED
### Return the first through 23rd characters of test_string. Assign to ans1
### YOUR ANSWER BELOW

ans1 = test_string[0:23]
```

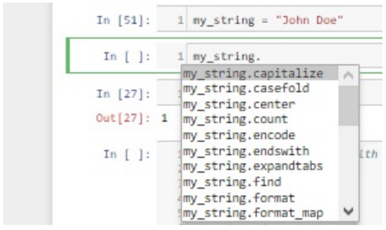
Question 6:

```
### GRADED
### return every fifth character from test_string, starting at the beginning, ending at the end. Assign to ans1
### YOUR ANSWER BELOW

ans1 = test_string[::5]
```

String Methods

Strings have many useful methods associated w ith them. While documentation is available online, `Jupyter Notebook` also has a useful feature for finding these methods.

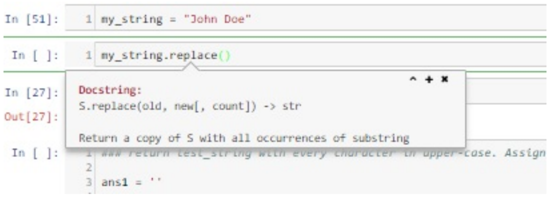
In the picture below I have created a string and assigned it to the variable "my_string". After that variable name, I put a period and I hit `<tab>`. A list of available `attributes` and `methods` w ill then pop up.



```
In [51]:  my_string = "John Doe"

In [ ]:   my_string.

In [27]:
Out[27]: 1

In [ ]:
```

Tw o important notes: 1. The variable must already be assigned to an object. (e.g. the cell w ith the assignment must have been run). 2. there must be a period after the variable name.

Another useful `Jupyter` tool displays documentation. After having selected a method and having placed the required parentheses; w hile the cursor is inside the parentheses, hitting `<shift-tab>` w ill bring up the documentation.



```
In [51]:  my_string = "John Doe"

In [ ]:   my_string.replace()

Docstring:
S.replace(old, new[, count]) -> str

Return a copy of S with all occurrences of substring

In [27]:
Out[27]:

In [ ]:   ### return test_string with every character in upper-case. Assign

          ans1 = ''
```

Below demos the method `.strip()` w hich removes the trailing and leading w hite-space from a string

```
my_string = "      John Doe      "
print( my_string)
print(my_string.strip())
```

The follow ing questions require the use of `test_string` (found above) and string methods w hich have not been explicitly covered here. It is up to you to find the appropriate methods and apply them.

Question 7:

```
### GRADED
### Return test_string with every character in upper-case. Assign to ans1
### YOUR ANSWER BELOW

ans1 = test_string.upper()
```

Question 8:

```
### GRADED
### Return the character that is 15th from the end of test_string.
### ### e.g. 1st from the end is "d", 2nd from end is "e". etc.
### Assign to ans1
### YOUR ANSWER BELOW

ans1 = test_string[-15]
```

Question 9:

```
### GRADED
### Replace every "t" - upper and lower case - in test string with a "*". Assign to ans1.
### YOUR ANSWER BELOW

ans1 = test_string.replace("t","*").replace("T","*")
```

Question 10:

```
### GRADED
### Count the number of times the lower-case letter "s", and upper-case letter "T" appears in test_string.
### Assign to ans1.
### YOUR ANSWER BELOW

ans1 = test_string.count("s") + test_string.count("T")
```

Functions:

The rest of the questions in this section rely on functions to test Python skills. Below is a graded example of a function. Again, do not change if you w ant credit for this question.

Question 11

```
### GRADED
### Example
### Build a function called "mult_by_two"
### ACCEPT one input either numeric or a string
### RETURN that input multiplied by 2

### YOUR ANSWER BELOW

def mult_by_two( input ):
    return input *2

### test the function to check functionality
print(mult_by_two(2))
print(mult_by_two(-1))
print(mult_by_two(0))
print(mult_by_two("hi"))
```

Note the syntax for functions:

- `def` , follow ed by the name of the function.
- follow ed by a set of parentheses w ith the inputs to the function (separated by commas; but parentheses can also be empty)
- follow ed by a colon.
- From then on, all the code of the function is indented four spaces (done automatically by `Jupyter`)

Below are a few more function examples.

```
def square_it( integer1 ):
    val_to_return = integer1 ** 2
    return val_to_return

def how_many_es( string_input):
    counter = 0
    counter = string_input.count("e")
    return counter

def multiply(num1, num2):
    return num1 * num2

def do_nothing( cat, dog):
    pass
```

Note, the above "pass" is simply a place-holder that tells `Python` to do nothing if the "pass" w ere not present. You w ill see "pass" in the empty functions for subsequent questions.

NB: As soon as a `return` statement is reached, that value is returned, and the function exits.

Question 12:

```
### GRADED
### Build a function called 'cap_first'
### ACCEPT a non-empty string input.
### RETURN that same string, with first letter capitalized,
### ### and all subsequent letters in lower-case.


### YOUR ANSWER BELOW

def cap_first( input_string ):
    return input_string.capitalize()
```

Question 13:

```
### GRADED
### Code a function called "add_three"
### ACCEPT  three inputs, all numbers (floats or ints)
### RETURN the sum of the three numbers added together.

### YOUR ANSWER BELOW

def add_three( num1, num2, num3):
    return num1 + num2 + num3
```

Question 14:

```
### GRADED
### Code a function called 'every_other'
### ACCEPT a non-empty string as input
### RETURN a string that contains every other character of that input string.

### YOUR ANSWER BELOW

def every_other( input_string ):
    return input_string [::2]
```

If / Else / Elif

The first elements of control flow to cover are if/else/elif statements.

`if` statments are coupled with a boolean statement. **If** the boolean statement is `True` the code block below the if will execute.

```
if True:
    print("This will print")
if False:
    print("This won't")
```

`else` statments follow `if` statements. They *can* but *do not have to* be present. The `else` will evaluate only when the boolean associated with the `if` evaluates to `False`

```
if True:
    print("This will print")
### Note: No else statement

if False:
    print("This won't")
else:
    print("But I will")
```

`elif` statements are used to evaluate a sequence of `if` statments.

Feel free to play around in the next cell

```
my_num = 4
if my_num == 1:
    print("it's one")
elif my_num == 2:
    print("it's two")
elif my_num == 3:
    print("it's three")
elif my_num == 4:
    print("it's four")
else:
    print("it's greater than four")
```

The Python comparators are:

- `==` is equal to
- `>` is greater than
- `>=` is greater than or equal to
- `<` is less than
- `<=` is less than or equal to
- `!=` is not equal to

Also, frequently, `in` might be used to create a boolean for testing to if an element is part of a collection. Collections covered next week.

```
print("a" in "aeiou")
print(1 in [1,2,3,4,5])
print(6 in [1,2,3,4,5])
```

Finally, boolean operations of `<and>`, `<or>`, and `<not>` are achieved with `and`, `or`, and `not`.

```
print(1==2 and False)
print(3<4 or 4>5)
print(not True)
print(not True or False)
```

If/Else Questions

Question 15:

```
### GRADED
### Code a function called 'pos_neg'
### ACCEPT a number (float or int) as input
### RETURN the string "neg" if the integer is less than zero;
### ### string "zero" if the integer is zero;
### ### or string "pos" if the integer is greater than zero.


### YOUR CODE BELOW

def pos_neg( num1 ):
    if i > 0:
        return "pos"
    elif i == 0:
        return "zero"
    else:
        return "neg"
```

Question 16:

```
### GRADED
### Code a function called "intro"
### ACCEPT two inputs: a string and a number
### RETURN the string "<string input> was born in ####" where #### is the 2018 minus the integer.
### HOWEVER, if the string is empty ("") OR the number is negative, RETURN the string "invalid input"
### NB: Be aware of spaces
### YOUR CODE BELOW

def intro( name, age):
    if name == "" or age < 0:
        return "invalid input"
    else:
        return name + " was born in " + str(2018-age)
```