Lab 1: Jupyter Notebook Tips

Submission Instructions

You are required to **submit your notebook on JupyterHub**. Please navigate to the Assignments tab to submit

- fetch
- modify
- validate
- submit

your notebook. Consult the instructional video (https://nbgrader.readthedocs.io/en/stable/user_guide/highlights.html#student-assignment-list-extension-for-jupyter-notebooks) for more information about JupyterHub.

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the homework, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** at the top of your solution.

Collaborators: *list names here*

Rubric

Question	Points
Question 1	2
Question 2	2
Total	4

0. Set-Up

If you're not using the course JupyterHub, then check out set up your personal computer correctly (https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/install.html) for instructions on installing Jupyter.

1. Keyboard Shortcuts

Here are some useful Jupyter notebook keyboard shortcuts. To learn more keyboard shortcuts, go to **Help -> Keyboard Shortcuts** in the menu above.

Here are a few we like:

1. `ctrl` + `return` : *Evaluate the current cell*
2. `shift` + `return` : *Evaluate the current cell and move to the next*
3. `esc` : *command mode* (may need to press before using any of the commands below)
4. `a` : *create a cell above*
5. `b` : *create a cell below*
6. `dd` : *delete a cell*
7. `m` : *convert a cell to markdown*
8. `y` : *convert a cell to code*

2. numpy

For a numpy reference, please review the following:

DS-UA 111 Textbook (https://www.inferentialthinking.com/chapters/05/1/Arrays)

3. Tip

Jupyter tip: Pull up the docs for any function in Jupyter by running a cell with the function name and a `?` at the end...you can escape by hitting `esc` several times

In [2]:

```
import numpy as np

np.arange?
```

4. Tip

Another Jupyter tip: Pull up the docs for any function in Jupyter by typing the function name, then `<Shift>-<Tab>` on your keyboard. This is super convenient when you forget the order of the arguments to a function. You can press `<Tab>` multiple times to expand the docs and reveal additional information.

Try it on the function below:

In [ ]:

```
np.linspace
```

Jupyter Notebook Tutorial:

For a detailed tutorial on Jupyter Notebooks, refer https://www.datacamp.com/community/tutorials/tutorial-jupyter-notebook#gs.ikwSeZc (https://www.datacamp.com/community/tutorials/tutorial-jupyter-notebook#gs.ikwSeZc)

Python

In case you are wondering, we will be using Python 3.3.

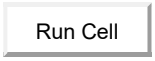
iPython Notebook

We will be working with Python in your browser using iPython Notebook. Here is an example block of code:

In [ ]:

```
print('Hello, World!')
# This is a comment, it isn't run as code, but often they are helpful
```

To run a block of code like the one above, click on it to select it and then you can either click the run button in the menu above:



or type shift-enter. The output of the block is shown below the block.

All of the code blocks on this page are interactive. Please make sure you run them all at least once. Feel free the change the code and see what the affect is.

Part 1

======

Arithmetic

Like every programming language, Python is a good calculator. Run the block of code below to make sure the answer is right!

In [ ]:

```
1 + 1
```

Hopefully it worked.

Now lets say you won the lottery. You are about to collect your millions, but first you have to answer this skill testing question:

"8+623-(15-13)"

Fortuntely Python can help. The order of operations you learned in school applies, BEDMAS (brackets, exponents, division, multiplication, addition, subtraction)

In [ ]:

```
8 + 6*2*3 - (15 - 13)
```

Numbers are valid Python code as are the common operators, +, /, * and -. You can write different types of numbers including integers, real numbers (floating point) and negative integers.

In [ ]:

```
42 + 3.149 + -1
```

Since 42 is literally 42, we call these numbers *literals*. You are literally writing numbers in your Python code.

Check Please

So you just had a big meal to celebrate your winnings, and now you need to calculate the tip.

In [ ]:

```
meal = 200.00
# as a decimal, 15% would be 0.15
tip_percent = 0.15
meal * tip_percent
```

If you want to make it more user friendly you could do the following.

In [ ]:

```
meal = 200.00
# as integer, 15% would be 15
tip_percent = 15
meal * tip_percent / 100
```

Because of BEDMAS we don't need brackets, but meal * (tip_percent / 100) would work too.

Variables

meal and tip_percent aren't literal numbers, they are variables.

In Python variables are like buckets (dump trucks?). You can put anything you want in them. Just give them a name and you can use them in place of the literal value.

Above meal was 200.00 but we could also set meal to the text 'Hello, World'

In [ ]:

```
meal = 200.00
print(meal)
meal = "Hello, World!"
print(meal)
```

The value a variable has only depends on what it was last assigned.

It is like a spreadsheet except you choose the names for the cells yourself.

Exceptional Python

Python only understands certain code. When you write something Python doesn't understand it throws an exception and tries to explain what went wrong, but it can only speak in a broken Pythonesque english. Let's see some examples by running these code blocks

In [ ]:

```
gibberish
```

In [ ]:

```
*adsflf_
```

In [ ]:

```
print('Hello'
```

In [ ]:

```
1v34
```

In [ ]:

```
2000 / 0
```

Python tries to tell you where it stopped understanding, but in the above examples, each program is only 1 line long.

It also tries to show you where on the line the problem happened with caret ("^").

Finally it tells you the type of thing that went wrong, (NameError, SyntaxError, ZeroDivisionError) and a bit more information like "name 'gibberish' is not defined" or "unexpected EOF while parsing".

Unfortunately you might not find "unexpected EOF while parsing" too helpful. EOF stands for End of File, but what file? What is parsing? Python does it's best, but it does take a bit of time to develop a knack for what these messages mean.

Part 2

The Written Word

Numbers are great... but most of our day to day computing needs involves text, from emails to tweets to documents.

We have already seen a text literal in Python, "Hello, World!"

In [ ]:

```
"Hello, World!"
```

Text literals are surrounded by quotes. Without the quotes Hello by itself would be viewed as a variable name.

You can use either double quotes (") or single quotes (') for text literals.

As we saw before we can also save text literals in variables.

What's a String?

Programmers call text literals *strings* because we are weird like that. From now on we will only refer to strings, but we just mean pieces of text inside our code.

Let's use strings with variables!

In [ ]:

```
your_name = "Albert O'Connor"
print("Hello, ")
print(your_name)
```

Strings in Python are a bit more complicated because the operations on them aren't just + and * (though those are valid operations).

Strings have their own operations we can call on them to change them. We can use dir to get an idea of what they are.

In [ ]:

```
dir("Hello, World!")
```

That is a really long list. For now you can ignore the ones which start and end with underscores ("_"), but that still leaves a lot! Let's start with upper and lower.

Let's say you are really happy to the world, and you want to make sure everyone knows it, you can use upper. "upper" is short for uppercase and you will see what it does by running the code block below.

In [ ]:

```
string = "Hello, World"
string.upper()
```

Maybe you are feeling a bit sad and you want to be quiet, you can then use lower.

In [ ]:

```
string = "Hello, World"
string.lower()
```

We use the dot (".") operator to call these operations on the string. What lower and upper operate on comes before the dot and needs to be a string variable or literal.

In [ ]:

```
"Hello, World".upper()
```

Formating

Sometimes you want to create a string out of a few other strings. Above we printed

```
Hello,
Your Name
```

by using two print statements, but it would be nice to output "Hello, Your Name!" instead. (Where Your Name is actually your name... oh variables)

We can do this with the string operation called format. Format is different from lower and upper because it can take other arguments. An *agrument* is what coders call the value passed to an operation or function, it doesn't mean we are fighting.

In [ ]:

```
your_name = "Albert O'Connor"
string = "Hello, {0}!"
print(string.format(your_name))
```

We can also used literal strings:

In [ ]:

```
print("Hello, {0}!".format("Albert O'Connor"))
```

{0} is kind of weird. {0} is where the first argument, "Albert O'Connor", to format() is placed in the resulting text. {1} would mean use the second value if there was one. See below!

Indexed by Zero

For better or worse, (and practically it is better most of the time) everything in Python is index by 0. We will see this over and over again but for now if you call format like this:

In [ ]:

```
"{0} likes {1}".format("Albert O'Connor", 'Python')
```

We would call "Albert O'Connor" the 0th string passed into format and 'Python' the 1st. It is kind of weird, but roll with it. It will eventually make things eaiser.

Line Endings

Let's say we wanted to represent the string in one string variable?

```
200 University Ave.
Waterloo, ON
```

A line ending is one example of something which is part of text on the screen that we need to somehow represent in a string. The key is the backslash ("\") character. "\n" and "\r\n" represents two kind of line endings. Try adding "\n" in the right place to make it appear like the text at the beginning of this section.

In [ ]:

```
print("200 University Ave. Waterloo, ON")
```

Exercise

Alice is sending a short email message to Bob. She wants to format the message on screen so she knows what she is sending. She is using the string called template below. Change the value of the template string to make it better. You can run the block to get further instructions.

In [ ]:

```
# Edit this string
template ="{0} {1} {2} {3}"

# Leave this alone please
# check('p1', template.format("alice@domain.org", "bob@domain.org", "Alice's Subject",
 "This is my one line message!"))
```

Part 3

If Else

Has an application ever ask you a question? Maybe it asks you if you really want to quit because unsaved changed might lost, or if you want to leave a webpage. If you answer OK one thing happen, like your application closing, but if you answer No or Cancel something else happens. In all those cases there is a special piece of code that is being run somewhere, it is an *if* condition.

Like all languages, Python allows us to conditionally run code.

To have an if condition we need the idea of something being true and something being false. Remember, we call numbers "integers" and "floating point", and text "strings". We call true or false "boolean" values. True would represent OK where as false would represent No or Cancel in the example above.

The literal values in Python for true and false are "True" and "False"

In [ ]:

```
False is False
```

In [ ]:

```
True is True
```

In [ ]:

```
True is False
```

In [ ]:

```
true is False
```

We can write expressions with operations too.

In [ ]:

```
1 > 2
```

In [ ]:

```
"Cool".startswith("C")
```

In [ ]:

```
"Cool".endswith("C")
```

In [ ]:

```
"oo" in "Cool"
```

In [ ]:

```
42 == 1 # note the double equals sign for equality
```

In order to write an "if" statement we need code that spans multiple lines

```
if condition:
    print("Condition is True")
else:
    print("Condition is False")
```

Some things to notice. The if condition ends in a colon (":"). In Python blocks of code are indicated with a colon (":") and are grouped by white space. Notice the else also ends with a colon (":"), "else:". Let's try changing the condition and see what happens.

In [ ]:

```
condition = 1 > 2
if condition:
    print("Condition is True")
else:
    print("Condition is False")
```

About that white space, consider the following code:

```
    if condition:
        print("Condition is True")
    else:
        print("Condition is False")
    print("Condition is True or False, either way this is outputted")
```

Since the last print statement isn't indented it gets run after the if block or the else block.

You can play with this. Try indenting the last print statement below and see what happens.

In [ ]:

```
condition = True
if condition:
    print("Condition is True")
else:
    print("Condition is False")
print("Condition is True or False, either way this is outputted")
```

Exercise

You can also use "and" and "or" to combine conditions. Let's look at and.

```
    True and True is True
    True and False is False
    False and True is False
    False and False is False
```

With "and" both conditions have to be True to be True.

Below change the values of the three variables to make the entire "if condition" true.

In [ ]:

```
# Edit the values of these 3 variables
boolean_literal = False
number = 8
string_literal = "I like to count sheep before bed."

# Leave this code the same please
# Note that these parentheses are optional, but you should use them if they increase re
adability
if (number > 10) and boolean_literal and ("cows" in string_literal):
    print("Success!")
else:
    print("Try again!")

# Leave this alone, please
# check("p3", (number, boolean_literal, string_literal))
```

Part 4

Lists

So far we have numbers, strings, and conditional if statements. Now for our first container — a list.

A list in Python is just like a shopping list or a list of numbers. They have a defined order and you can add to it or remove from it.

Let's take a look at some simple lists.

In [ ]:

```
# The empty list
[]
```

In [ ]:

```
["Milk", "Eggs", "Bacon"]
```

In [ ]:

```
[1,2,3]
```

List literals are all about square brackets ("[ ]") and commas (","). You can create a list of literals by wrapping them in square brackets and separating them with commas.

You can even mix different types of things into the same list; numbers, strings, booleans.

In [ ]:

```
[True, 0, "Awesome"]
```

We can put variables into a list and set a variable to a list.

In [ ]:

```
your_name = "Albert O'Connor"
awesome_people = ["Eric Idle", your_name]
print(awesome_people)
```

Like strings lists have operations

In [ ]:

```
dir([])
```

"append" is an interesting one. "append" lets you add an item to the end of a list.

In [ ]:

```
your_name = "Albert O'Connor"
awesome_people = ["Eric Idle", your_name]
awesome_people.append("John Cleese")
print(awesome_people)
```

We can use square brackets ("[]") again with the variable of the list to access individual elements.

In [ ]:

```
awesome_people[0]
```

There is that 0 indexing again. The first element of the list is given index value 0.

In [ ]:

```
print("These people are awesome: {0}, {1}, {2}".format(awesome_people[0], awesome_peopl
e[1], awesome_people[2]))
```

Loops

Indexes are useful, but lists really shine when you start looping.

Loops let you do something for each item in a list. They are kind of like if statements because they have an indented block.

They look like this:

```
for item in list:
    print(item) # Do any action per item in the list
```

"for" and "in" are required. "list" can be any variable or literal which is like a list. "item" is the name you want to give each item of the list in the indented block as you iterate through. We call each step where item has a new value an iteration.

Let's see it in action with our list

In [ ]:

```
your_name = "Albert O'Connor"
awesome_people = ["Eric Idle", your_name]
awesome_people.append("John Cleese")

for person in awesome_people:
    print(person)
```

This is bascially the same as writing

In [ ]:

```
person = awesome_people[0]
print(person)
person = awesome_people[1]
print(person)
person = awesome_people[2]
print(person)
```

But that is a lot more code than:

```
for person in awesome_people:
    print(person)
```

Considering that our list of awesome people could be very long!

You can use the built-in function "range" to create lists of numbers easily

In [ ]:

```
range(0,10)
```

And then we can use that with a loop to print a list of squares.

In [ ]:

```
for number in range(0,10):
    print("{0} squared is {1}".format(number, number*number))
```

Exercise

Create a list of numbers where every item in the list is the same as its index, i.e. number_list[4] is 4. The list should contain 5 items.

In [ ]:

```
# Edit the contents of this list
number_list = []

# Leave this line alone please
# check("p4", number_list)
```

Part 5

Dictionaries

We have come a long way! Just one more section. Dictionaries are another container like lists, but instead of being index by a number like 0 or 1 it is indexed by a key which can be almost anything. The name comes from being able to use it to represent a dictionary.

List literals use square brackets ("[]") but dictionaries use braces ("{}"). Use "shift-[" to type "{".

```
{"Python": "An awesome programming language",
 "Monty Python": "A british comedy troupe"}
```

In a dictionary the key comes first followed by a colon (":") than the value then a comma (",") then another key and so on. This is one situation where a colon doesn't start a block.

Let's see what running the literal dictionary looks like.

In [ ]:

```
{"Python": "An awesome programming language",
 "Monty Python": "A british comedy troupe"}
```

We can assign a dictionary to a variable and we can index it by keys to get the values (definitions) out.

In [ ]:

```
our_dictionary = {
   "Python": "An awesome programming language",
   "Monty Python": "A british comedy troupe"
}
our_dictionary["Python"]
```

We can loop over the keys in a dictionary to list all of our definitions...

In [ ]:

```
for key in our_dictionary:
    print('The Key is "{0}" and the value is  "{1}"'.format(key, our_dictionary[key]))
```

Mail Merge

Our project this weekend is to write a mail merge program. Given a template for the message and a list of dictionaries of information including email addresses, we can create several messages each personalized depending on the information we have.

The data will start as a spreadsheet but it will end up in a dictionary. Each message will have its own dictionary. We can use the string formating you have learned about with a little twist to get Python to format the dictionaries as emails for us.

We will look at the basics now, but don't worry will be working on this today and tomorrow!

Format with Dictionaries

So far our formatting strings have looked like this:

```
"Hello, {0}"
```

Where "0" in "{0}" refers to the index of the arguments we pass into the format function. We can also put argument names inbetween the braces "{}".

In [ ]:

```
print('Hello, {name}! Your favorite color is {favorite_color}.'.format(name="Albert O'C
onnor",
                                                                favorite_color=
'green'))
```

This is handy if you ever need to reuse a value and you don't want to list it twice.

But we can do more, instead of passing keyword arguments into the format function like we did above, we can pass in a dictionary. Note the dictionaries keys have to be strings, but that is exactly what we will be doing.

In [ ]:

```
info = {'name': "Albert O'Connor",
        'favorite color': 'green'}
print('Hello, {name}! Your favorite color is {favorite color}.'.format(**info))
```

The "**" before data is a bit of stranger Python syntax. It means instead of passing info as the first agrument, pass all the key value pairs inside the info dictionary as keyword arguments. Let's look at a few varation on the syntax one at a time.

In [ ]:

```
data = {'name': "Albert O'Connor",
        'favorite_color': 'green'}
print('{0}'.format(data)) # This prints the dictionary as text
```

In [ ]:

```
data = {'name': "Albert O'Connor",
        'favorite_color': 'green'}
print('{0}'.format(**data)) # This produces an error, there are no indexable arguments,
just keyword
```

In [ ]:

```
data = {'name': "Albert O'Connor",
        'favorite_color': 'green'}
print('{0}'.format('Eric!', **data)) # Eric is the 0th argument, all the keywords are i
gnored!
```

In [ ]:

```
data = {'name': "Albert O'Connor",
        'favorite_color': 'green'}
# data is passed as indexiable so there is no keyword arguments.
print('Hello, {name}! Your favorite color is {favorite_color}.'.format(data))
```

In [ ]:

```
data = {'name': "Albert O'Connor",
        'favorite_color': 'green'}
# There we go, name and favorite_color are passed as keywords.
print('Hello, {name}! Your favorite color is {favorite_color}.'.format(**data))
```

In [ ]:

```
data = {'name': "Albert O'Connor",
        'favorite_color': 'green'}
# data is passed as keyword, but doesn't have the key 'pet_name'.
print('{pet_name}'.format(**data))
```

The excersises in this section will combine the above idea with lists and loops.

Exercises

First we need some data, in this case we want to create two dictionaries, each with a name and email key: Alice (alice@nyu.edu) and Bob (bob@nyu.edu)

Question 1

In [10]:

```
# Edit with the values for Alice.
entry_1 = {'name': "",
           'email': ""}

# Add the keys and values to this entry like above for Bob
entry_2 = {}


### BEGIN SOLUTION
entry_1 = {'name': "Alice",
           'email': "alice@nyu.edu"}
entry_2 = {'name': "Bob",
           'email': "bob@nyu.edu"}
### END SOLUTION
```

In [11]:

```
# TEST

assert entry_1['name'] == "Alice"
```

In [12]:

```
### BEGIN HIDDEN TESTS
assert entry_2['email'] == 'bob@nyu.edu'
### END HIDDEN TESTS
```

Now we want to write our own message which uses both the 'name' and 'email' values. We are going to use another way to write Python strings, inside triple quote string you can just press enter for a newline, no "\n" required. Just write as you would normally in an editor.

In [13]:

```
message = """To: {email}

Hey {name},

How is the weather?
"""

print(message)
```

```
To: {email}

Hey {name},

How is the weather?
```

Question 2

Final step, let's format the message with each of the entries.

In [14]:

```
# Fill in the missing pieces for Alice (entry_1) and Bob (entry_2) respectively

formatted_message_Alice = message.format(name = ... , email = ...)
formatted_message_Bob = message.format(name = ... , email = ...)

### BEGIN SOLUTION
formatted_message_Alice = message.format(name = entry_1['name'] , email = entry_1['emai
l'])
formatted_message_Bob = message.format(name = entry_2['name'] , email = entry_2['email'
])
### END SOLUTION
```

In [15]:

```
# TEST

assert formatted_message_Alice == """To: alice@nyu.edu

Hey Alice,

How is the weather?
"""
```

In [16]:

```
### BEGIN HIDDEN TESTS

assert formatted_message_Bob == """To: bob@nyu.edu

Hey Bob,

How is the weather?
"""
### END HIDDEN TESTS
```

Lab1 Complete! Congratulations.

Ensure you follow the submission instructions. Please submit this file on JupyterHub as mentioned above.
