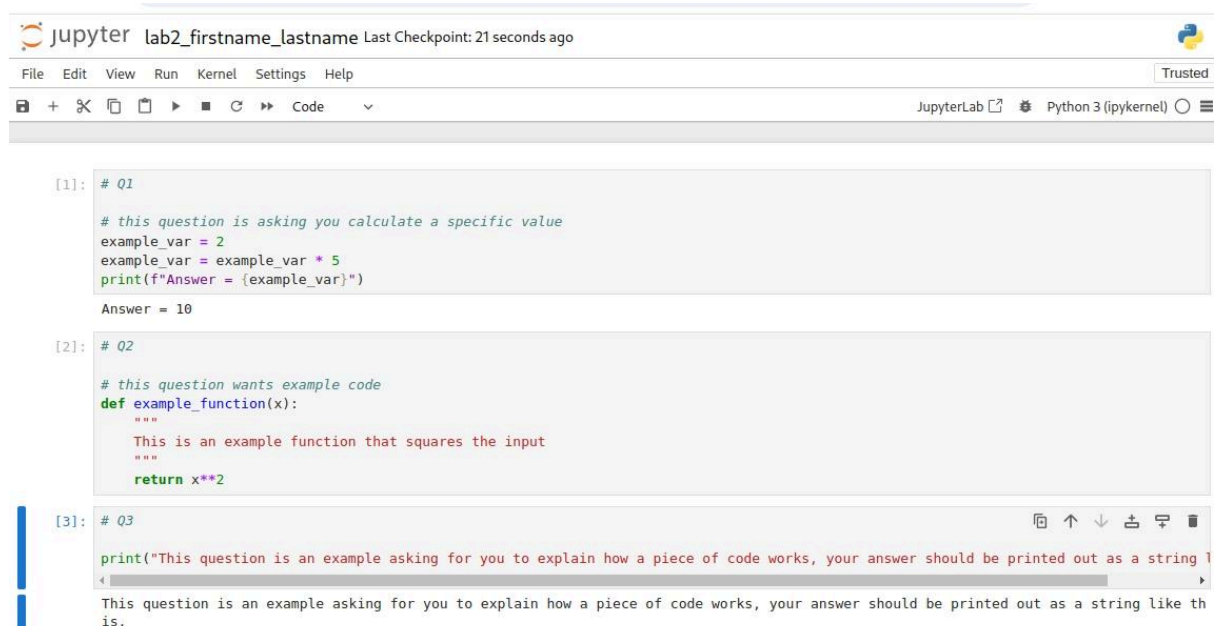


Lab 2: Solidifying the basics in Python

Part 0: Submitting

For this week's lab please submit your work to brightspace in this format:

- A single jupyter notebook containing all your answers in .ipynb format. You can do this by going to *file -> save as -> filename.ipynb -> enter*
- The answer to each question should be in its own code cell (either all the requested code, explicitly printing out the calculated answer, or printing out a string containing the text answer - examples below)
- Please indicate which question each cell is answering using a comment of the format “# Q1”, “# Q2” etc. We will talk about more features of jupyter notebook next week (but if you are already familiar with jupyter notebooks, you can use “markdown” cells before the code cell to indicate the question).



```
[1]: # Q1
# this question is asking you calculate a specific value
example_var = 2
example_var = example_var * 5
print(f"Answer = {example_var}")

Answer = 10

[2]: # Q2
# this question wants example code
def example_function(x):
    """
    This is an example function that squares the input
    """
    return x**2

[3]: # Q3
print("This question is an example asking for you to explain how a piece of code works, your answer should be printed out as a string like this.")

This question is an example asking for you to explain how a piece of code works, your answer should be printed out as a string like this.
```

As the course progresses these practical instructions will get a lot shorter. I'm providing a lot of detail to make sure you have a firm understanding of the basics of python before things start to get more complicated.

¹ Like last week's practical, this heavily draws on material from Zed Shaw's "Learn Python the Hard Way, 5th Edition" and Bust, Carr Markell, and Wirth's CS5 Course

Part 1: Booleans and Conditionals

In Python we have the following terms (characters and phrases) for determining if something is "True" or "False." Logic on a computer is all about seeing if some combination of these characters and some variables is True at that point in the program.

- `and`
- `or`
- `not`
- `!=` (not equal)
- `==` (equal)
- `>=` (greater-than-equal)
- `<=` (less-than-equal)
- `True`
- `False`

Combinations of these values are really important in programming and form the entire basis of "flow-control" (e.g., conditionals and loops). You should be familiar with the inequality (<, >, ==, !=) from maths but here is a refresher on how "and" & "or" work:

A	B	A or B
True	True	True
False	True	True
True	False	True
False	False	False

A	B	A and B
True	True	True
False	True	False
True	False	False
False	False	False

Q1. Here is a series of Boolean statements. Using comments, manually evaluate each comparison into True/False within them and then write whether the entire statement evaluates to True or False. For example, if given `(1 == 1 and 2 == 1)`

```
# statement: (1 == 1 and 2 == 1)
# 1 == 1 is True so I can replace 1 == 1 with True
# (True and 2 == 1)
# 2 == 1 is False so I can replace 2 == 1 with False
# (True and False)
# (True and False) evaluates as False
print("(1 == 1 and 2 == 1) == False")
```

Here is another example for not (1 != 10 or 3 == 4)

```
# statement: (not (1 != 10 or 3 == 4))
# 1 != 10 is True so I can replace it with True
# 3 == 4 is False so I can replace it with False
# True or False is True
# not (True) is False
# print("(not (1 != 10 or 3 == 4)) == False")
```

Now do the same for each of the following:

1. ("test" == "test")
2. (1 == 1 or 2 != 1)
3. (True and 1 == 1)
4. (False and 0 != 0)
5. (True or 1 == 1)
6. ("test" == "testing")
7. (1 != 0 and 2 == 1)
8. ("test" != "testing")
9. ("test" == 1)
10. (not (True and False))
11. (not (1 == 1 and 0 != 1))
12. (not (10 == 1 or 1000 == 1000))
13. (not ("testing" == "testing" and "Zed" == "Cool Guy"))
14. (1 == 1 and (not ("testing" == 1 or 1 == 0)))
15. ("chunky" == "bacon" and (not (3 == 4 or 3 == 3)))
16. (3 != 3 and (not ("testing" == "testing" or "Python" == "Fun")))

Now we've had lots of practice with booleans, let's start using these to control how our code actually runs. To refresh from the lecture, the syntax of if, elif, and else statements are as follows and `condition` and `other_condition` represent boolean conditionals (e.g., `x == 5`, `x < 10`) like above.

```
if condition:
    print("condition is True")
elif other_condition:
    print("condition is False but other_condition is True")
else:
    print("condition and other_condition are False")
```

One thing that can be very useful with conditionals is “nesting” where we have another conditional within a conditional. This “nesting” is indicated based on how far the code is “indented” (i.e., how much whitespace there is before a bit of code”).

```
if x < 5:
    if x == 4:
        print("magic number")
    else:
        print("boring number")
```

While really powerful, this kind of nesting can get hard to read pretty quickly (which tends to lead to bugs)!

Q2. Go to this link

(https://maguire-lab.github.io/scientific_computing/static_files/practicals/lab2_conditional.txt) and copy the code into a cell. Write out the two possible sets of choices (e.g., 1 followed by “a bigger computer”) that lead to “Excellent Research!” being printed out.

Part 2: Introduction to Functions

Functions do three things:

1. They name pieces of code the way variables name strings and numbers.
2. They take arguments of variables (like you've been giving to the function “print”).
3. Using 1 and 2, they let you make your own "mini-scripts" or "tiny commands."

You can create an *empty* function by using the word `def` in Python like this:

```
def do_nothing():
    pass
```

This creates the function, but the `pass` keyword tells Python this function is empty. To make the function do something, you add the code for the function *under* the `def` line, but indent it 4 spaces:

```
def do_something():
    print("I did something!")
```

This effectively assigns the code `print("I did something!")` to the name `do_something` so you can then use it again later in your code, similar to other variables. Using a function you've defined is how you "run" it, or "call" it:

```
def do_something():
    print("I did something!")

# Now we can call it by its name
do_something()
```

When the `do_something()` at the bottom runs, Python does the following:

1. Finds the `do_something` function in Python's memory.
2. Sees you're calling it with `()`.
3. Jumps to where the `def do_something()` line is.
4. Runs the lines of code *under* the `def`, which in this case is one line `print("I did something!")`.
5. When the code under the `def` is finished, Python exits the function and jumps back to where you called it.
6. Then it continues, which in this case is the end of the code.

You can also pass "arguments" to functions:

```
def do_more_things(a, b):
    print("A IS", a, "B IS", b)

do_more_things("hello", 1)
```

In this case, I have two arguments (also called parameters) to the `do_more_things` function: `a` and `b`. When I call this function using `do_more_things("hello", 1)` Python *temporarily* assigns `a="hello"` and `b=1` and then calls the function. That means, inside the function `a` and `b` will have those values, and they'll disappear when the function exits. It's kind of like doing this:

```
def do_more_things(a, b):
    a = "hello"
    b = 1
    print("A IS", a, "B IS", b)
```

The name of the variable inside the function doesn't have to match the parameter when you call it. When you go inside a function it's like another world!!!. For example, the parameter inside the function is called `arg1` but you are passing in the variable `y`

```
def print_one(arg1):
    print(f"arg1: {arg1}")

y = "First!"
print_one(y)
```

Functions can also explicitly return values so you can assign them to new variables.

```
def add(a, b):
    print(f"ADDING {a} + {b}")
    return a + b

def subtract(a, b):
    print(f"SUBTRACTING {a} - {b}")
    return a - b

def multiply(a, b):
    print(f"MULTIPLYING {a} * {b}")
    return a * b

def divide(a, b):
    print(f"DIVIDING {a} / {b}")
    return a / b

print("Let's do some math with just functions!")

age = add(30, 5)
height = subtract(78, 4)
weight = multiply(90, 2)
iq = divide(100, 2)

print(f"Age: {age}, Height: {height}, Weight: {weight}, IQ: {iq}")
```

Q3. Evaluate what x will contain by hand

```
x = add(age, subtract(height, multiply(weight, divide(iq, 2))))
```

Here is a more complicated example that shows some of the different ways you can pass parameters to a function:

```
def calculate_kinetic_energy_and_momentum(mass, velocity):
    '''
    Using mass and velocity values calculate the kinetic energy and momentum
```

```

of an object
'''
kinetic_energy = 0.5 * mass * velocity**2
momentum = mass * velocity
print(f"An object with mass {mass}kg and velocity {velocity}m/s has:")
print(f"Kinetic Energy: {kinetic_energy:.2f} Joules")
print(f"Momentum: {momentum:.2f} kg·m/s")
print()

print("We can pass numbers directly to the function:")
calculate_kinetic_energy_and_momentum(2.0, 10.0)

print("OR, we can use variables from our script:")
object_mass = 5.0 # kg
object_velocity = 15.0 # m/s
calculate_kinetic_energy_and_momentum(object_mass, object_velocity)

print("We can even do calculations inside the function call:")
# Doubled mass and doubled g
calculate_kinetic_energy_and_momentum(1.5 * 2, 9.81 * 2)

print("And we can combine variables with calculations:")
calculate_kinetic_energy_and_momentum(object_mass + 10, object_velocity * 2)

```

Q4. What is the kinetic energy and momentum of an object of mass 50kg and a velocity of 100 m/s?

Q5. Write a function called `power(x)` that takes a number `x` as input and returns x^x . For example, `power(2)` returns 4 and `power(3)` returns 27. (As an experiment, try to find the smallest integer value for `x` such that `power(x)` returns a number greater than 1 million.)

Q6. Write a function that calculates the position of a ball thrown up in the air (like you did last week without a function). The vertical position of the ball y changes with time t according to the following equation (where v_0 is the initial velocity and g is the acceleration due to gravity):

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

Test this function by answering what is the position of the ball at $t = 0.9$ s when $v_0 = 5$ m/s, $g = 9.81$ m/s², using Python, what is the position of the ball at $t = 0.6$ s?

Print your answer in the following format -> the position of the ball at $t = 0.6$ s is [your answer]

Q7: Write a function that converts temperatures from Celsius (C) degrees into corresponding Fahrenheit (F) degrees with the formula: $F = \frac{9}{5}C + 32$

Use this and print out the F corresponding to 5C.

Part 3: Strings

You embed variables inside a string by using a special `{}` sequence and then put the variable you want inside the `{}` characters. You also must start the string with the letter `f` for "format", as in `f"Hello {somevar}"`. This little `f` before the `"` (double-quote) and the `{}` characters tells Python 3, "Hey, this string needs to be formatted. Put these variables in there.". This is known as an f-string. For example, `types_of_people = 10` creates a variable named `types_of_people` and sets it = (equal) to `10`. You can put that in any string with `{types_of_people}`. You also see that I have to use a special type of string to "format"; it's called an "f-string" and looks like this:

```
types_of_people = 10
x = f"There are {types_of_people} types of people."

binary = "binary"
do_not = "don't"
y = f"Those who know {binary} and those who {do_not}."

print(x)
print(y)

print(f"I said: {x}")
print(f"I also said: '{y}'")

hilarious = False
joke_evaluation = "Isn't that joke so funny?! {}"

print(joke_evaluation.format(hilarious))

w = "This is the left side of..."
e = "a string with a right side."

print(w + e)
```

Q8 As you saw last week several numerical operators work on strings. The following code combines (also known as concatenating) strings and prints the chemical formula for glucose. Edit the strings to print the chemical formula for deoxyribose (C₅H₁₀O₄). Include a comment that explains what `end=' '` does.

```
part1 = "C"
part2 = "6"
part3 = "H"
part4 = "1"
part5 = "2"
```



```
part6 = "0"
part7 = "6"

print(part1 + part2 + part3 + part4 + part5, end=' ')
print(part6 + part7)
```

Q9. Write a function called `string_multiply(my_string, number)` that takes as input a string called `my_string` and a positive integer called `number` and returns a new string that is the concatenation of `number` copies of `my_string`. For example, `string_multiply('hi', 3)` returns `'hihihi'`.

Python also supports multi-line strings using 3 `"""`

```
print("""
There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
""")
```

There are also special bits of text that aren't necessarily printed but are used to tell python to do certain things to the string like spacing.

```
months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"
print("Here are the months: ", months)
```

Note the characters `\n` (backslash `n`) between the names of the months. These two characters put a **new line character** into the string at that point which creates a new line (like pressing enter in a normal text editor).

This `\` (backslash) character encodes these types of difficult-to-type characters into a string. This is sometimes known as an “escape” character and the special code like `“\n”` is an “escape sequence” because we are escaping from Python treating this as just character `\` followed by the character `n`.

An important escape sequence is to escape a single-quote `'` or double-quote `”`. Imagine you have a string that uses double-quotes and you want to put a double-quote inside the string. If you write `”I understand” joe.”` then Python will get confused because it will think the `”` around `”understand”` actually *ends* the string. You need a way to tell Python that the `”` inside the string isn't a *real* double-quote.

To solve this problem you *escape* double-quotes and single-quotes so Python knows to include them in the string. Here's an example:

```
"I am 6'2\" tall." # escape double-quote inside string
'I am 6\'2" tall.' # escape single-quote inside string
```

There are lots of other escape sequences in Python

```
tabby_cat = "\tI'm tabbed in."
persian_cat = "I'm split\non a line."
backslash_cat = "I'm \\ a \\ cat."

fat_cat = """
I'll make a list:
\t* Cat food
\t* Fishies
\t* Catnip\n\t* Grass
"""

print(tabby_cat)
print(persian_cat)
print(backslash_cat)
print(fat_cat)
```

Q10. If we use `\` to indicate escape characters how do we print a string that contains the actual character `\`. Write a print statement that prints a string that includes the character `\`.

We have a similar problem with f-strings, because python tries to replace the text between `{` and `}` with the value of a variable, if we want to print the character `{` as well as a variable we need to tell python to treat `{` as a normal `{` not a special f-string `{`.

```
print(f"This is how we write an f-string containing {5+4} and the
literal characters {{ and }}")
```

We also talked about how you can index into specific parts of a string `my_string[position]` and use "slices" `my_string[start:stop]` (or even `my_string[start:stop:increment]`) to get a subsequence of a string. Remember python strings index at 0 and are not inclusive of the stop position (i.e., `my_string[start:stop]` gives you the string from the start position up to but NOT including the stop).

Q11. A DNA string is made up of A, G, C, and T characters (corresponding to each DNA nucleotide e.g., T = thymine). These are organised into codons which correspond to 3 consecutive characters. For the following DNA string create and print f-strings with just the following parts of the string (using indexes and slices).

```
dna_sequence = "ATGCTAGCTAGCTAGCTGATCGATGCTAGCTAGCTGATCG"
```

- First nucleotide in the string
- Last nucleotide in the string
- First codon (i.e., first 3 characters)
- The entire dna_string reversed
- The nucleotides at position 10, 11, 12, 13, 14, and 15
- Every other nucleotide from position 5 to 12.
- The first nucleotide of each codon (i.e., the 0th, 4th... etc)

Q12. Bacteria such as Salmonella are pathogenic because they have certain genes that allow them to make you sick. Many such genes can be found in clusters called *pathogenicity islands* in the genome. For researchers studying a pathogenic bacterium, finding these islands is an important step in understanding how the bacterium makes people sick.

Calculating GC content can be useful in helping find pathogenicity islands. The GC content is the proportion of bases that are G's or C's. It turns out that the GC content in pathogenicity islands frequently differs from other regions of the genome.

Using some of the built-in string functions discussed in the lecture, write a short function to calculate the proportion of a DNA string that is G's or C's

Here are some examples of `gc_content(dna_string)` at work:

```
>>> gc_content("ACCGC")
0.8
>>> gc_content("ATACTAAA")
0.125
```

Part 4: Lists

Most programming languages have some way to store data inside the computer. Some languages only have raw memory locations, but programmers easily make mistakes when that's the case. In modern languages, you're provided with some core ways to store data called "data structures". A data structure takes pieces of data (integers, strings, and even other data structures) and organizes them in some useful way. In this exercise we'll learn about the sequence style of data structures called a "list" or "Array" depending on the language.

Python's simplest sequence data structure is the `list` which is an ordered list of things. You can access the elements of a `list` randomly, in order, extend it, shrink it, and most anything else you could do to a sequence of things in real life.

You make a `list` like this:

```
fruit = ["apples", "oranges", "grapes"]
```

That's all. Just put [(left-square-bracket) and] (right-square-bracket) around the **list** of things and separate them with commas. You can also put anything you want into a **list**, even other **lists**:

```
inventory = [ ["Buick", 10], ["Corvette", 1], ["Toyota", 4]]
```

In this code, I have a **list**, and that **list** has 3 **lists** inside it. Each of those **lists** then has a name of a car type and the count of inventory. Study this and make sure you can take it apart when you read it. Storing **lists** inside **lists** inside other data structures is very common.

What if you want the 1st element of the **inventory list**? How about the number of Buick cars you have in inventory? You do this:

```
# get the buick record
buicks = inventory[0]
buick_count = buicks[1]
# or in one move
count_of_buicks = inventory[0][1]
```

In the first two lines of code (after the comment) I do a two-step process. I use the `inventory[0]` code to get the *first* element. If you're not familiar with programming languages most start at 0 not 1 as that makes math work better in most situations. The use of [] right after a variable name tells Python that this is a "container thing" and says we want to "index into this thing with this value", in this case, 0. In the next line, I take the `buicks[1]` element and get the count 10 from it.

You don't have to do that though as you can chain the uses of [] in a sequence so that you dive deeper into a **list** as you go. In the last line of code, I do that with `inventory[0][1]` which says "get the 0 element, and then get the 1 element of *that*".

Here's where you're going to make a mistake. The second [1] does not mean to get the entire ["Buick", 10]. It's not linear, it's "recursive", meaning it dives into the structure. You are getting 10 in ["Buick", 10]. It is more accurately just a combination of the first two lines of code.

Lists are simple enough, but you need to practice accessing different parts of very complicated **lists**. It's important that you can *correctly* understand how an index into a nested **list** will work.

Q13. Copy the following lists into your notebooks. For each of the following **list** names and pieces of data in the **list** write down how to get that information. For example, if I tell you **fruit** 'AAA' then your answer is `fruit[0][2]`. You should attempt to do this in your head by looking at the code, then test your guess in the Jupyter

```
fruit = [
    ['Apples', 12, 'AAA'], ['Oranges', 1, 'B'],
    ['Pears', 2, 'A'], ['Grapes', 14, 'UR']]
```

```
languages = [
    ['Python', ['Slow', ['Terrible', 'Mush']]],
    ['JavaScript', ['Moderate', ['Alright', 'Bizarre']]],
    ['Perl6', ['Moderate', ['Fun', 'Weird']]],
    ['C', ['Fast', ['Annoying', 'Dangerous']]],
    ['Forth', ['Fast', ['Fun', 'Difficult']]],
]
```

You need to get all of these elements out of the `fruit` variable:

- 12
- 'AAA'
- 2
- 'Oranges'
- 'Grapes'
- 14
- 'Apples'

You need to get all of these elements out of the `languages` variable:

- 'Slow'
- 'Alright'
- 'Dangerous'
- 'Fast'
- 'Difficult'
- 'Fun'
- 'Annoying'
- 'Weird'
- 'Moderate'

Q14. What do the following indexing operations spell out:

```
languages[0][1][1][1]
fruit[2][1]
languages[3][1][0]
```

Part 5: For Loops

The nice thing about python is that it makes it very easy to loop over strings or lists (or any other object that implements some special code we'll talk about later in the course called either an iterator or a generator). Here are some examples of the basics of loop including the range function.

```
the_count = [1, 2, 3, 4, 5]
fruits = ['apples', 'oranges', 'pears', 'apricots']
change = [1, 'pennies', 2, 'dimes', 3, 'quarters']

# This first kind of for-loop goes through a list
```

```

for number in the_count:
    print(f"This is count {number}")

# same as above
for fruit in fruits:
    print(f"A fruit of type: {fruit}")

# Also we can go through mixed lists too
for i in change:
    print(f"I got {i}")

# we can also build lists, first start with an empty one
elements = []

# Then use the range function to do 0 to 5 counts
for i in range(0, 6):
    print(f"Adding {i} to the list.")
    # append is a function that lists understand
    elements.append(i)

# Now we can print them out too
for i in elements:
    print(f"Element was: {i}")

```

Q15. Imagine that we have a list of DNA sequences and we wish to know how many of them are of a specific length. Write a function called `count_list(dna_list, length)` that takes a list of DNA strings called `dna_list` and a positive integer `length` as input and returns the number of strings in the list that have the specified `length`. Use the built-in `len` function and `for` loops. Here's an example of the function in action:

```

>>> count_length(["ATA", "ATCG", "TTT", "A"], 3)
2
>>> count_length(["AACC", "A", "T"], 2)
0

```

Part 6: Bringing it all together

Q16. The pattern of these characters encodes the instructions of how to make a protein in a cell. Like your own Python code, there is a syntax to this and only specially structured strings of DNA called *open reading frames (ORF)* are used to create proteins. A DNA string is only an ORF if it begins with 'ATG', ends with 'TGA', 'TAG', or 'TAA', and has a length that is a multiple of 3. Write a function that takes a string called DNA as input and works as follows:

- The function returns the string 'This is an ORF.' if the input string satisfies all three of the conditions required of ORFs.
- Otherwise, if the first three symbols are not 'ATG', the function returns the string 'The first three bases are not ATG.'
- Otherwise, if the string does not end with 'TGA', 'TAG', or 'TAA', the function returns the string 'The last three bases are not a stop codon.'
- Otherwise, if the string length is NOT a multiple of 3, the function returns the string 'The string is not of the correct length.'

Show the output for this function on the following strings

```
"AGGGGCCGGCTGCGCATAGCGA "  
"ATGAAACCCGTGCACAAA "  
"ATGGGGCCCATGTAA "  
"ATGGCCCGTGGCACGATAG "
```