Welcome to Tutorial 2!

Keep up the good work! Today we'll be learning about functions, packages, and basic data visualization.

Part 0: Functions

Conditional statements and loops are quite powerful on their own. However, when programs get longer and longer, it's useful to have a way to organize code. This is where functions come in.

If variables are like nouns, then functions are like verbs. They do things to variables. Some functions are more complicated than others, and some functions modify more variables than others.

You've had a little but of experience with functions before. Remember our old friend, `print()` ? `print()` is a function that is automatically included in Python. We use it to display things as output. Quite useful!
In Tutorial 1, you also saw functions called `range()` and `len()` . What do they do?
When you're confused about a Python function, you can (almost) always refer to the Python documentation about it. You can search for it online, or use the `help()` function on it:

In [ ]:
```
# Run this code!
help(len)
```

Hmm, these are some 50-cent words, but here's the gist: the function `len` will tell you the number of items within an `object`. In order to work, this particular function needs an `obj` to take in. What happens if we try to use `len()` on nothing?

In [ ]:
```
# Run this code!
len()
```

An error message appears! Don't panic--the error message is telling us that we need to give `len` something to count. Before, we gave it a list of objects. Let's try counting some things:

In [ ]:
```
my_list = ['Dr. Hall', 'Dr. Das', 'Dr. Ice']

# How many items are in my_list?
print(len(my_list))

# How long is the first item in my_list ('Dr. Hall')?
print(len(my_list[0]))
```

What does `range()` do?

In [ ]:
```
# Run this code!
help(range)
```

Information overload! Don't worry: what we're really interested in is what `range` takes in as an **input** and what it will **return**. **Return** is a fancy word that means "give back to us as a result." In this case, `range` takes in one or two numbers. Then, it will **return** (or "give back to us") a list of integers between those two numbers.
Given your newfound expertise about the `range` function, what do you think the following code will print out?

In [ ]:
```
for i in range(1,5):
    print(i*2)
```

Is this what you expected? Explain your reasoning below.

Answer Here

Now that we've familiarized ourselves with the anatomy of functions, we can actually start writing our own. There are three things that every function needs:

1. The word `def`. The word `def` will alert Python that you are about to tell it about a new function.
2. A name. It's helpful to make it a descriptive name, when you start using your functions in other pieces of code.
3. A return statement. This is always the last part of a Python function. When Python sees the word `return`, it will say "I'm done!" and leave the function. Here's a pretty simple function that just has a name and a return statement:

In [ ]:
```
def myFcn(): # Everything in your function after this line must be indented!
    return(0)
```

Just by reading the code above, you can probably that it's a fairly useless function. It doesn't do anything except for tell us `0`! To make functions more useful, we need to give them inputs. In programming, we refer to these inputs as **arguments**. **Arguments** are things like variables that you give to **functions**. They are nouns that our verbs act on.

In [ ]:
```
# Run this code!
def cube(num):
    out = num ** 3 # The ** means that I am raising an exponent
    return(out)
```

Ahh, a more useful function! Note that it should take a single *numeric* argument: giving this function a string or a boolean variable wouldn't make any sense. It would probably also complain if we tried to give it a list, or two numeric arguments at once. Also note that it should `return` a single number as the result.
Let's take her for a spin!

In [ ]:
```
cube(3)
```

It looks like the function works! But, to build better programming habits, I'm going to add a block comment at the beginning of the function to explain a little bit more about what it does. Just like a regular comment (that you can write with a `#`), it will tell other human readers about your function. You can start and end a block comment with three double quotes (`"""`).

In [ ]:
```
def cube(num):
    """
    Takes in a single numeric variable and computes the cube of it.
    Returns the result as a single numeric value.
    """
    out = num ** 3
    return(out)
```

Now that Python knows more about what this function does, we can call the `help` function on it!

In [ ]:
```
help(cube)
```

But, our functions can take in more than one argument. It can also take in arguments of different data types! Python will expect to see arguments in the order in which they are given in the function definition. The order of arguments matters.
Here's an example of a function that takes in multiple arguments.

In [ ]:
```
def square_or_cube(numbers, squareBool):
    """
    Inputs: "numbers" will be a list of numerical values.
        "squareBool" will be a boolean variable indicating whether to square a
value or not.
    Output: A list of either squared or cubed values.
    """
    if squareBool:
        for i in range(len(numbers)):
            numbers[i] = numbers[i] ** 2
    else:
        for i in range(len(numbers)):
            numbers[i] = numbers[i] ** 3
    return(numbers)
```

What do you think the output of the following code is? Refer back to the function that we just defined before you run it.

In [ ]:
```
print(square_or_cube([1, 2, 3], True))
```

Is this what you expected? Explain your reasoning below.

Answer Here

It's also worth nothing that variables you use or define inside of functions have a small **scope**. Python has no idea what these variables mean outside of the functions they are defined in!

Your Turn!

Go back to the code you wrote in Tutorial 1 for a reverse transcriptase. Let's practice turning it into a function. You'll have to:

1. Define a function that takes in one argument: a string of characters.
2. Place your code from Tutorial 2 inside the body of the function. Instead of `print`-ing the result, make sure you `return` it instead.
3. Test your function on the RNA snipped I've included below. Be sure to `print` the results of your new function, so we can see the results.

In [ ]:
```
rna_seq = 'GGAUCGCCUA'
# Your Code After This Line
```

Part 1: Packages

Writing our own functions can be nice and convenient. But sometimes, there's no need to write functions when we know that other people have done the work for us already. In Python (and many other programming languages), other computer scientists have collected useful functions into **packages.**
Packages are typically open-access, and are full of useful functions that coders before us have written for other people to use.

We've installed a few important packages on all of your distributions of Python. The first one you'll be introduced to is `numpy` (**num**erical **Py**thon). `numpy` is a *huge* library. If you continue to work with Python, you will encounter it time and time again.
It's time to get introduced to `numpy`! Her nickname will be `np`. It's shorter than using her full name.

In [ ]:
```
# Run this code!
import numpy as np
```

Great! Once you have a package installed, that one-line `import` statement is all you need to start accessing all of the tools inside of it. Here's are some examples of some of `numpy`'s useful functions:

In [ ]:
```
# Get the base-10 logarithm of a number
print(np.log10(1000))

# Get the determinant of a matrix
print(np.linalg.det([[1, 0], [3, 4]]))

# Raise each element in the first argument to the powers listed in the second
 argument
print(np.power([1, 2, 3], [3, 2, 1]))
```

And we're just scratching the surface! In the project, you will be using other useful Python libraries in addition to `numpy`.

Part 2: Data Visualization

A widely used library of functions for visualizing and plotting information is called `matplotlib.pyplot`. By convention, the nickname for this library is `plt`.

In [ ]:
```
import matplotlib.pyplot as plt
```

`pyplot` can make all sorts of useful graphs, including line plots, scatter plots, histograms, and lots more.

In [ ]:
```
# Get some random numbers to play around with
data_to_plot = np.random.rand(50)
# Plot these values
plt.figure()
plt.plot(data_to_plot)
plt.show()
```

In the first line of the previous cell, we just generated some random numbers.
In the second line of code, we `called` a new figure. This `call` alerts Python that we want to show the user a set of graphs.
The third line of code uses `plt.plot` to create a line graph on the `figure`.
The fourth line tells Python to put this line graph on the figure we wanted to show the user.
When using `pyplot`, you generally always need:

- a call using `plt.figure()`
- some function to plot your data
- showing your plot using `plt.show()`

The plot we just made is nice, but it needs to be cleaned up a little bit. Let's give it some titles, and play around with the markers.

In [ ]:
```
plt.figure()
plt.plot(data_to_plot, 'go')
plt.xlabel('Index of each data point')
plt.ylabel('Value')
plt.show()
```

Instead of a line plot, we used green (' g ') circular (' o ') markers to denote each data point. Python knows a lot of colors, such as `r` (red), `b` (blue), and `k` (black). There are also different marker symbols, including `^` (triangle), `s` (square), and `x` (cross). There are lots of variations on this, and they can all be found in the Python documentation!

To count the frequency of each value occurring, maybe it's best if we use a histogram. Instead of using `plt.plot`, we should use `plt.hist`.

In [ ]:
```
plt.figure()
plt.hist(data_to_plot, bins=20)
plt.xlabel('Index of each data point')
plt.ylabel('Value')
plt.show()
```

Note that I passed an argument called `bins` to the `plt.hist` function. This will change the number of bars that are displayed on the histogram, and as a result will change how finely your data is displayed. Feel free to play around with the number of `bins` to see how it changes.
Run `help(plt.hist)` to find out some more options you can change.

A short detour and a cool trick

Remember one of our favorite data types, strings? It turns out that you can easily break up a string into a `list` of characters, just by using the function `list()`. Try out the following example:

In [ ]:
```
example_letters = list("biochemistry")
print(example_letters)
```

Your turn!

Using this cool new trick, let's try to visualize the amino acid composition of a short sequence. In the following cell, I've copied over one of the shortest complete polypeptide sequences: one of extracellular hemoglobin of *Tylorrhynchus heterochaetus*, a type of polychaete worm. Your task is to:

1. Split up this sequence into a list of its residues.
2. Plot the number of occurrences each residue has using a histogram.
3. Label the axes of your figure and give it a pertinent title.

You can check your work against the original paper (uploaded on Canvas under the name **polychaete.pdf**).

Extra challenge: Can you use some color schemes to color nonpolar, basic, and acidic residues in different colors? Can you label each bar on the histogram with the number?
There are many, many creative ways you can represent data using `pyplot` and other Python libraries. I encourage you to explore!

In [ ]:
```
complete_sequence = "TDCGILZRIKVKQQWAGVYSVGESRTDFAIDVFNFFRTNPDRSLFNRVNGDNVYSPE
FKAHMVRVFAGFDILISVLDDKPVLDQALAHYAAFHKQFGTIPFKAFGQTMFQTIAEHIHGADIGAWRACYAEQIVTG
ITA"
# Your Code After This Line
```

In [ ]:
```

```