Welcome to Python!

Whether you are super-familiar with Jupyter Notebook or someone who has never written a line of code before, our goal here is to familiarize you with some computational tools that we often find useful in bioinformatics and genomics.

Each of these tutorials will contain some tools that will be central to completing the final Python Project.

No matter the level of experience you have,

reach out for help if you are stuck. The best way to learn is to learn from others! You are always welcome to ask the teaching staff for help. You are also welcome to work together with other participants. **SSP's policy on finding help to "de-bug" your programs is this:**

- Your first place to look for help is Google.
- If you do not find an answer within ~5 minutes, you can always ask a member of the teaching staff for help.
- You may ask your fellow participants for help. You may show them your computer screen and any error messages that arise. **However:** the person helping you cannot assume control of your keyboard and type anything.
- Likewise, **you are not allowed to copy-and-paste and/or send code to anyone else. You are also not allowed to copy-and-paste any code you find online, unless you give it proper citation and acknowledgement.**

If you have extensive experience with Python 3 and Jupyter Notebook:

...you may consider skipping ahead to Tutorial 3.

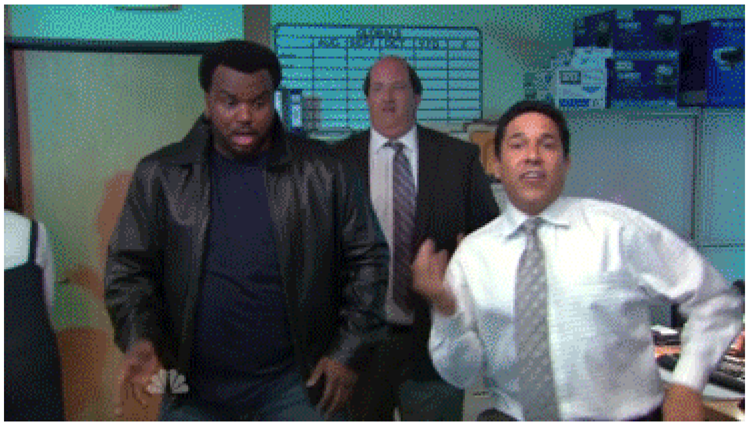
If you have previously done some computer coding and need to familiarize yourself with concepts:

...you may consider skimming over Tutorials 0 and 1 before skipping ahead to Tutorials 2 and 3.

If you have never done any computer coding before:

...welcome! Programming is challenging and might be intimidating, but it is an extremely powerful tool to have in your arsenal. There will be a lot of frustrating moments and problems to solve ahead, but you have everyone cheering for you!

Let's get started!



Before we begin

You are currently working on a platform called **Jupyter Notebook**. Essentially, the Notebook is a convenient place for us to show you code and write a lot of notes in. It is also a place where you can write your own code and interact with it! If you **click once on this text**, you will notice that a rectangle appears around this paragraph. The thick line on the left-hand side tells you that you have highlighted a **cell**. Each **cell** contains text that doesn't do anything, **or** computer code that you can run.

In [ ]:
```
# This cell has been changed to contain Python code!
# There are several ways to run Python code inside of a notebook. You can run
 the entire notebook by clicking the "Run" button
# in the toolbar at the top. You can also go to "Cell" > "Run Cells" to run an
y cells you have selected.
# You can also run a single cell at a time by pressing shift + enter.
# Try selecting this cell (click once) and pressing shift+enter.

import datetime
print("Hello, world!")
print("The current time is", datetime.datetime.now())
```

Congratulations! You've just executed your first Python code.

By the end of this Tutorial, you will be able to code your own rudimentary reverse translator.

Part 0: Variables

There are many building blocks we can use when it comes to coding. The first thing is **variables**. Variables are like nouns. We can name them and define them. To do, we use an equals (**=**) sign. Try running the code below.

In [ ]:
```python
my_variable = 5
print(my_variable)
```

In the previous two lines of code, we defined a variable called `my_variable` . We set it equal to the number 5. (Notice that the name that I wanted the variable to be called is on the **left side** of the equals sign, and the value that I wanted it to be is on the **right side** of the equals sign.) Just to check that `my_variable` is what we think it is, we told Python to `print` out the value of `my_variable`.
But, **be careful when you choose new variable names!** What if we made a new variable with the same name?

In [ ]:
```python
# Run this code!
my_variable = 9
print(my_variable)
```

What happened to the number `5` ?

In the previous chunk of code, we defined `my_variable` to be the number 9. This overwrote any instances of `my_variable` that existed before. In other words, we told Python: "Forget anything that was called `my_variable` before. This is `my_variable` now, and it is equal to `9` ."

What kinds of variables are there?

Just like there are different types of nouns (people, places, and things), there are different types of variables. Variables can be integers, decimal values, words/letters, lists, and so on! Let's see what different categories of variables there are. We call these different categories **data types.**
In the next few cells, I've defined some variables, and asked Python to tell me what data type each of them are. Run the cells below and look at the output to see the data types of each variable.

In [ ]:
```python
variable1 = 2020
type(variable1)
```

In [ ]:
```python
variable2 = 3.14159
type(variable2)
```

In [ ]:
```python
variable3 = 10.
type(variable3)
```

In [ ]:
```python
variable4 = "Hello!"
type(variable4)
```

In [ ]:
```python
variable5 = "asdfghj"
type(variable5)
```

In [ ]:
```python
variable6 = True
type(variable6)
```

```
In [ ]: variable7 = [1, 2, 3, 4]
        type(variable7)
```

```
In [ ]: variable8 = [[0, 1, 2], ['cat', 'dog'], ['PUR', 'IU', 'NMT', 'CUB']]
        type(variable8)
```

To recap, here are some different data types that we can use:

- **int**: a number without any decimal points
- **float**: a decimal with decimal points. If you add a decimal point after an int, it becomes a float!
- **str**: a string. Typically a bunch of characters/letters/numbers. They don't have to make sense! When defining string, you **must** type it in single quotes (') or double quotes (").
- **bool**: a boolean variable. These variables can only be **True** or **False**. You might also see them represented as **1** and **0**.
- **list**: stuff that is enclosed within square brackets ([ ]). Each **element** inside is separated by commas. Elements inside of a list can be ints, floats, strings--even other lists!

A quick detour into error messages

Sometimes, when Python sees code that it doesn't understand, it will spit out an angry, red-and-green error message at you. It will look nasty and scary, but don't worry!

At the end of the day, computers are not very smart. We need to tell computers what we want them to do. When you get an error message, it's because Python gave up on trying to understand what you told it to do.

Over time, you will become very good at deciphering these error messages. I've deliberately written the following cell to generate an error message. Run it and see what happens.

```
In [ ]: variable9 = nine
```

Let's break this error message down.

1. The first part of the error message will tell you the "location" of where the error message occured. This is more apparent when working in text editor softwares, where the "location" will usually be a line number.
2. The second part of the error message will display the actual piece of code that gave Python some trouble, with an arrow next to it.
3. The last line of the error message will be a short summary of what went wrong. This is usually the most important and useful piece of information.

Here, Python said that the name `nine` was not defined. Python thinks that `nine` is a variable, and it got frustrated because it hasn't been told about any variables named `nine` ! Let's change that in the next cell and see what happens.

In [ ]:
```
nine = 9
variable9 = nine
```

Much better! Error messages are not meant to look intimidating, but it can sure feel frustrating when Python doesn't do what you ask it to.

Part 1: Numbers and Strings

Once we have variables, we can start manipulating them! Let's do some math using our number variables (**int** and **float** data types).
What do you think the following cell does? Think about it, then run the cell to see if you are correct.

In [ ]:
```
a = 5
b = a + 1
print(b)
```

In the first line, we defined an int called `a`. In the second line, we defined a variable called `b` that was equal to the value of `a`, plus 1. Python realized that there was a variable called `a` that was equal to the value of `5`, so it realized that we wanted `b` to be equal to `5 + 1`.
In the third line, we told Python to `print` the value of the variable `b` so that we could see it.
What's the data type of `b`?

In [ ]:
```
type(b)
```

Looks like `b` is an **int**. Is this what you expected?
Try running the next cell of code.

In [ ]:
```
c = 1.
d = a + c
type(d)
```

Is this what you expected? Think about it, and explain your reasoning below:

Answer Here (double-click to type in cell)

Numbers are boring. What's next?

Strings are very important for genomics. How would you represent a codon sequence or series of amino acids? Turns out, one of the easiest ways is to use a jumble of characters.

In [ ]:
```
# Run this code
start_codon = "AUG"
Tyr = "UAU"
Leu = "UUA"
amber = "UAG"
```

We can **concatenate**, or "glue together", strings by using the plus (**+**) symbol. Think about what the following cell does, and try running it.

In [ ]:
```
sequence = start_codon + Tyr + Leu + amber
print(sequence)
```

Is this what you expected? Explain your reasoning below:

Answer Here

Your Turn!

Write some code to print out the codon sequence that would encode the short amino acid sequence **SSP** (including start and stop codons). You'll have to do the following:

1. Define string variables for a start codon, serine codon, proline codon, and stop codon. (You'll have to look up what these codons are.) Although there are multiple possible codons for S, P, and "stop", just pick one.
2. Concatenate the strings together.
3. Print out your final codon sequence.

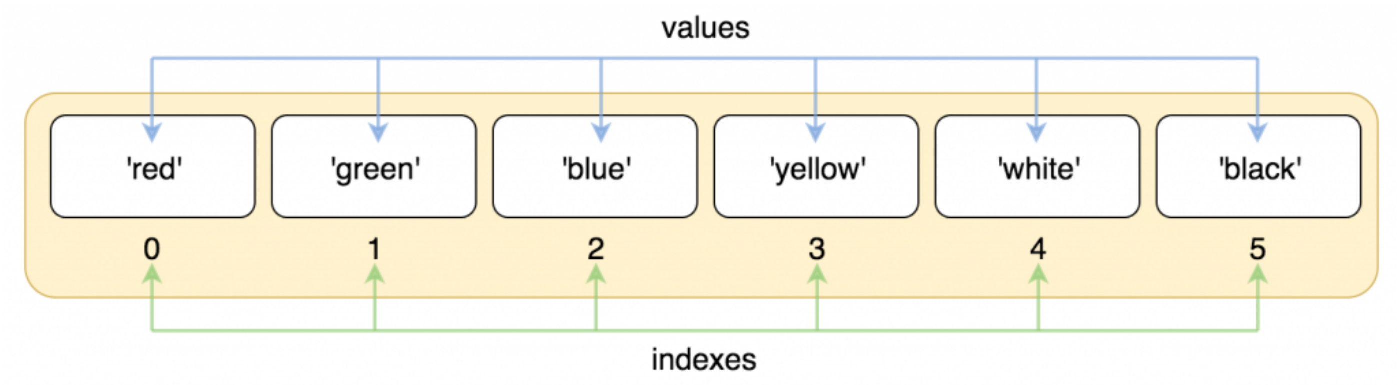
In [ ]:
```
# Type your code in this cell and run it!
```

Part 2: Lists

Great work so far! Variables are a tricky concept to learn, but you've all got the foundations down. You've already written your first lines of computer code! With some practice, you'll all be pros in no time.
Lists are a particularly tricky variable to work with. Recall: **lists** are enclosed in square brackets (**[ ]**) and contain **elements** that are separated by commas (**,**). We can access each of the **elements** by their position within the list, or **index**. Suppose we have the following list:

```
colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
```

The following graphic (courtesy of Railsware) shows each of the elements in the list and their respective indices. Note that **in Python, indexing starts from 0!** This means that the index of the first element is 0, not 1.



We can access specific elements within a list using the name of the list and the position of our desired element. For instance, what if we wanted to access the color "red" in our list of colors? We would need to know the name of the list it belongs in (`colors`), and its position within the list (`0`). To access the `0` th element, place the number `0` in square brackets after the list name.

```
In [ ]: # Run this code!
        colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
        colors[0]
```

List indexing is not just useful for *accessing* information. We can also use it to *replace* elements with our list. Once we access our desired element within a list, we can define it and use it just like a variable.
Think about what the following code does, and try running it.

```
In [ ]: # Run this code!
        colors[2] = 'magenta'
        print(colors)
```

Is this what you expected? Think about it, and explain your reasoning below:

Answer Here

Your Turn!

Your task is to print out another codon sequence that can encode **SSP**, along with start and stop codons. You can use the same start codon you used before. For our redundant codons, we're going to have to pick a specific codon from all of the possible codons that can encode an amino acid.
Below, I've typed out some lists representing some redundant codons. You'll have to use list indexing to access a specific codon of your choice. Then, concatenate your 5 codons together and print the sequence out.
Extra Challenge: How many lines of code does it take you to do this? Can you complete the entire task in *one* line of code?

In [ ]:
```
Ser_codons = ['UCU', 'UCC', 'UCA', 'UCG', 'AGU', 'AGC']
Pro_codons = ['CCU', 'CCC', 'CCA', 'CCG']
stop_codons = ['UAA', 'UAG', 'UGA']

# Your Code After This Line
```

In [ ]:
```
```