# Introduction to Python - Part 1

## What is python?

## Using Python

### The python shell

The python shell, similar to the bash shell, allows us to use python in an interactive manner. You enter in one command at a time and the result is immediately returned. The python shell can be called directly from the terminal with the command `python`

This is a very convenient tool for testing python commands, and for getting started with the language, but it is not very useful for creating actual programs and or scripts to run. To make an actual program, you will need to put your code in a text file and save it with a `.py` extension.

### Python scripts

To create a python script, we can click on File > New > Text File in the top menu above. This will open the Jupyter Lab text editor in which we are going to create our very first python program. ["Hello World" demonstration]

### IPython Notebooks

Another way in which we can use python is how we are doing so here.

## Code Along

## Open a Python Notebook

**Launch JupyterLab and click to create a new Python3 notebook. To best understand the information here, you should enter the variables and commands you see in each coding cell and run it yourself (this document does not contain any of the command results). Remember you can run cells by pressing `Shift Enter`. The ouput of these commands will appear below the cell.**

## The Basics

If you ever get lost or need more information about how a function or object works in python, you can use the `help()` function.

In [ ]:
```
help('+')
```

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

# Variables

## Data Types

In python, like most programming languages 'data type' is an important concept. When you store data in variables, you need to be aware of *how* that data is being stored. This is referred to as a data type. Different data types can do different things and can interact with different 'operators' (e.g. '+, -, *, &, etc' in distinct ways. For example the `+` operator will sum the values of two variables that contain numbers:

In [ ]:
```
a = 2
b = 3
a+b
```

but it will concatenate two variables that contain strings:

In [ ]:
```
a = '2'
b = '3'
a+b
```

The following data types are 'built-in' by default in python:

- Text Type: `str` (abbreviation for string)
- Numeric Types: `int`, `float` (int is the abbreviation for integer)
- Sequence Types: `list`, `tuple`, `range`
- Mapping Type: `dict` (abbreviation for dictionary)
- Set Types: `set`

You can always check to see what 'type' a variable is by using the `type()` function

In [ ]:
```
x = 5
type(x)
```

Python will automatically set the data type when you create a variable under certain conditions. Listed here are examples of the different data types and their necessary punctuation (ex. `""`, `[]`, `{}`, or `()`)

In [ ]:
```
x = "myString" # str
x = 10         # int
x = 10.0       # float
x = ['pizza','apple','hotdog']  # list
x = ('pizza','apple','hotdog')  # tuple
x = {'pizza','apple','hotdog'}  # set
x = range(10)  # range
x = {"name": "Loyal", "age": 41, "department" : "neuroscience"}  # dict
x = True       # bool
```

You can also explictly set the data type that you want by using standard functions. This is known as 'casting' a force or change variable to be a specific type.

In [ ]:
```
x = str(5)
x
```

In [ ]:
```
x = float(4)
x
```

## Number types

`int` (integer) numbers are whole numbers (positive or negative) without decimals. There is no limit to the size of an integer in python.

In [ ]:
```
a = 3
b = 2939875239875757329237598273757 5
c = -42
```

The integer type should be used for numbers that will always be whole for their operations. Think of counting the number of bases in a DNA sequence, or counting the number of times something happens. You will almost never have a 'partial' quantity for these values. But be careful of how `int` types are handled when you do certain types of operations:

In [ ]:
```
a = 5
b = 10
c = 10/5
print(c)
type(c)
```

The `float` type is for 'floating point numbers'. These can be positive or negative and can contain one or more decimals. The `float` type can be specified by adding a decimal value to the end of a number when assigned to a variable:

In [ ]:
```
a = 5.0
b = 1.467283
c = -14.22
```

The 'float' type can also be scientific notation by adding an `e` to indicate the power of 10:

In [ ]:
```
x = 2e4
print(x)

y = -63.5e100
print(y)
type(y)
```

You can convert from one type to another with the `int()` and `float()` functions.

In [ ]:
```
int(y)
```

## Strings

Strings or string literals are generally denoted by enclosing them in quotes; either single ( ' ) or double ( " ) quotes:

In [ ]:
```
print("Welcome to BCMB!")
```

You can assign a string to a variable in the same way.

In [ ]:
```
a = "Welcome to BCMB!"
print(a)
```

Sometimes strings will span multiple lines. If this is the case, it is convention to enclose them in triple quotes:

In [ ]:
```
b = """Welcome to BCMB!
You've chosen the most exciting graduate program at JHU."""

print(b)
```

## Strings, substrings, and slicing

Strings in python are stored as 'arrays of bytes' representing each charcater, which bascially means that "Hello" is actually stored in python in something akin to a list: `["H","e","l","l","o"]` . So we can access, edit, and manipulate different parts of a string (or any other array) by 'slicing' with square brackets. Its important to recognize when doing this that python is a '0-indexed' language which means that any time you count in python, you always start with 0. Lets see how this works in practice:

In [ ]:
```
# Create a string literal and store in a variable
a = "Genomics is fun"

# To retrieve the second character in this string we will slice as so
print(a[1])

# To retrieve a range of positions, we will separate the start and end using ':'
print(a[2:6])

# You can use negative values to start the slice from the end of the string
print(a[-6:-1])

# You can leave either side of the ':' black to represent the beginning or end of a str
print(a[:8])

print(a[9:])
```

Slicing is a fundamental concept in python for accessing any set of elements in an array (not just strings). You will use this often.

Strings have several functions that are available for some common queries and manipulations:

In [ ]:
```
# Get the length of the string
print(len(a))
```

```
# Convert the string to lower case
print(a.lower())

# upper?

# You can strip off excess white space
b = "My favorite gene is Pantr2.    "
print(b.strip())

# You can replace portions of the string
print(a.replace("fun","hard"))

# or split a string into substrings based on a 'separator' which in this case in a sing
print(a.split(" "))
```

You can also test for instances of a substring within a string (note the use of a new syntax here that we will explore later in this document). Essentially, we are telling the variable *x* to identify "sea" in the string assigned to variable *a*. The result for `print(x)` is `True` meaning that the string 'sea' was identified in the string assigned to the variable *a*. Test out the other commands to see what they return.

In [ ]:
```
a = "She sells seashells by the seashore."
x = "sea" in a
print(x)

y = "sho" in a
print(y)

z = "sho" not in a
print(z)
print(type(z))
```

You can combine strings directly (concatenate) using the `+` operator as we discussed above:

In [ ]:
```
a = "Toad"
b = "the"
c = "wet"
d = "Sprocket"

print(a + b + c + d)
```

Whoops...we may want to format this a bit better to add a separator. Fortunately, python provides a convenient way to format strings called 'f-strings'. Simply add the variables in a new string and enclose them with curly braces `{}` .

In [ ]:
```
text = f"The best band ever is {a} {b} {c} {d}!"
print(text)
```

This is a *very* useful tool for formatting output strings containing useful pieces of information in your code/scripts

In [ ]:
```
gc = 56
```

```
name = 'Pantr2'
chromosome = 'chr4'

summary = f"The {name} gene is located on chromosome '{chromosome}' and has a GC conten

print(summary)
```

There are a number of methods for manipulating/searching/testing strings that are built in to python. Feel free to check them out and test them on your own.

# Boolean type

The Boolean type refers predominantly to logical tests, and ultimately, `type: bool` can only have two values: `True` or `False` (case sensitive). There are often times when you need to test a value or an expression. In python the value returned from these test is a `bool`:

In [ ]:
```
print(14 > 3)
print(14 == 3)
print(14 < 3)
```

You can use `bool` values and variables to help control the flow of your code. For example, we could print a message based on whether or not a condition is `True`.

In [ ]:
```
a = 50
b = 100

if a < b:
    print ("a is the smaller value")
else:
    print("b is the smaller value")
```

# Collection Data types

Collection data types store groups of `items`. Items can be named variables, or objects of other data types, including other collections (nested). There are four main collection data types, each with their own properties/assets:

1. A *List* is an *ordered* collection and is *mutable* (can be altered). It can also hold duplicate items.
2. A *Tuple* is an *ordered* collection and is *immutable* (cannot be altered). It also allows for duplicate items.
3. A *Set* is an *unordered* collection and *unindexed*. It does *not* allow for duplicate items.
4. A *Dictionary* is an *unordered* collection which is *mutable* and *indexed*. It does not allow for duplicate index keys.

## Lists

A list is instantiated using square brakets `[]`

In [ ]:
```
genes = ['Gapdh','Mef2c','Pax6','Cxcl1','Msi1']
```

You can access list items by referring to the index number (remember that python is zero-indexed).

In [ ]:
```
print(genes[1])

print(genes[-2]) # negative indexing to select items from the end of the list. (-1 refe

print(genes[1:3]) # you select a range using ":" (returns a new list)
```

Since `list` items are mutable, you can change any specific item by refering to it's index number

In [ ]:
```
genes[2] = 'Sox10'
print(genes)
```

`list` collections (like all collection items) are *iterable*, meaning you can loop through elements.

In [ ]:
```
for x in genes:
    print(x.upper())
```

To add items to a list (at the end) you can use the `append()` function

In [ ]:
```
genes.append('Foxp1')

print(genes)
```

Conversely, you remove using several methods:

In [ ]:
```
genes.remove('Mef2c') # removes a 'specific' item
print(genes)

genes.pop() # removes a specified index position or the last item in the list if index
print(genes)

genes.clear() # empties the entire list
print(genes)
```

To join two lists, you can use the `+` operator

In [ ]:
```
fruit = ['apple', 'banana','pear']
veg = ['carrot','celery','potato']

food = fruit + veg

print(food)
```

## Tuples

Tuples operate very similar to lists, but once instantiated, the items in a tuple cannot be changed. Tuples are created with round brackets `()`. This is a useful data type to hold values associated with a single 'record'. For example if you wanted to record specific information about a single gene like its name, chromosome, and start position:

In [ ]:
```
a = ('Sox2','chr4',1589182)
b = ('Xist','chrX',23564335)
```

You access individual elements of a tuple in the same way as a list

In [ ]:
```
print(a[0])

print(b[1])
```

You can also loop through a tuple since it it iterable. We will go over loops more in depth in our first day of class, but give this one a try to see the output:

In [ ]:
```
for val in a:
    print(val)
```

Once you create a tuple, you cannot change the values, and you cannot add items to it.

# Dictionaries

Dictionaries are 'indexed' collections, meaning the *values* within the collection each must have a unique *key*. You can create a dictionary using curly braces `{}`.

In [ ]:
```
myGene = {
    'name': 'Sox2',
    'entrezID': 6657,
    'Ensembl': 'ENSG00000181449',
    'chromosome': 'chr3',
    'start': 181711925,
    'end': 181714436,
    'strand': '-'
}
print(myGene)
```

To access elements of a dictionary, you do so in a manner similar to other collection items ( `[]` ), but you must specify a 'key' value.

In [ ]:
```
print(myGene['chromosome'])
```

Dictionaries are mutable so you can change/assign values in the same way

In [ ]:
```
myGene['strand'] = '+'

print(myGene['strand'])
```

When you loop through a dictionary, the values returned are the key index values

In [ ]:
```
for key in myGene:
    print(key)
```

Or you can return dictionary keys in the dictionary key format:

In [ ]:
```
myGene.keys()
```

You can also iterate over the values, or key:value pairs

In [ ]:
```
for val in myGene.values():
    print(val)
```

Or if you wish to return the dictionary with the key and value pairs on individual lines you can loop through the dictionary as follows:

In [ ]:
```
for k, v in myGene.items():
    print(f'key: {k}  value:{v}')
```

Sometimes it may be useful to check if a key exists in a dictionary.

In [ ]:
```
lookup = "Ensembl"
if lookup in myGene:
    print(f"Found {lookup} in myGene dictionary keys.")
else:
    print("Key not found in myGene dictionary")
```

# Control Flow

An important aspect of programming is manipulating the flow of how the program executes commands/functions/operations/etc. This is how programs and scripts take some decisions and execute different things depending on different situations. The structure of most control flow elements in python is fairly similar: evaluate certain conditions/statements and follow this with a colon ( `:` ). The subsequent *code block* is below this statement and always indented. The block ends when the indentation ends. There are three types of control flow statements in python: `if` , `for` , and `while` . Each operates a bit differently.

## If...Else

You have already seen this type of control flow in the examples above, but let's dive a little bit deeper into how it works. The `if` statement first evaluates whether a given expression is `True` . If so, then the associated code block is then executed. Lets make sure we understand how it's organized.

In [ ]:
```
a = 5
if a < 10:
    print(f'{a} is less than 10.')

if a >= 10:
    print(f'{a} is greater than or equal to 10.')
```

Here we've constructed two `if` statements to test the variable `a` . Notice that the first statement (which evaluates to `True` ) is executed but the second ( `False` ) is not. We can also use the `else`

and `elif` (read: 'else,if') statement to further condition how python responds to our conditional test.

In [ ]:
```
a = 50
if a < 10:
    print(f'{a} is less than 10.')
elif a == 10:
    print(f'{a} is equal to 10.')
else:
    print(f'{a} is greater than 10.')
```

There are a few things to note here. First, the `elif` statement, we are providing *another* conditional test for the variable `a`. If the first `if` returns `False`, then the next `elif` in the program will then be tested. If all of the specific `if` and `elif` statements return `False`, then the remaining code block under the `else` statement is evaluated. In this way, `else` acts as a 'catch all' if none of the other statements are `True`.

*Only one* of the statements above will be executed; the first statement to evaluate to `True`. Once this happens, python steps out of the `if...else` statement and then continues on wth the rest of the program.

The second point to make from the above is the use of the double `=` in the `elif`. This is the 'comparison operator'. A single `=` is used as the 'assignment operator' as we have been using to assign values to variables. If we want to test that two values are in fact equal, then we *must* use `==`.

## Operators

The construction of boolean logical tests is an important part of how you control the flow of your python program. Often we want to test whether a variable has a certain value, or even exists at all. Or perform some mathematical transformation on a value. To do this, we use different 'operators'. Operators are the constructs which can manipulate the value of individual items (or operands). You are familiar with many of these, for example 5 + 3 = 8. In this expression 5 and 3 are operands and '+' is the operator. Python has several types of operators, here we will distinguish between a few types

### Arithmetic operators

These you should be inherently familar with for the most part. Assume that a = 10 and b = 20:

- `+` Addition: Adds values on either side of the operator. a + b = 30
- `-` Subtraction: Subtracts right hand operand from left hand operand. a – b = -10
- `*` Multiplication: Multiplies values on either side of the operator a * b = 200
- `/` Division: Divides left hand operand by right hand operand b / a = 2
- `%` Modulus: Divides left hand operand by right hand operand and returns remainder b % a = 0
- `**` Exponent: Performs exponential (power) calculation on operators a**b =10 to the power 20

### Comparison operators

These are the operators that allow you to compare two items/variables. Each of these operators returns a `bool` value ( `True` or `False` )

In [ ]:
```
a = 10
b = 20

print(a == b) # evaluates whether two operands are equal
print(a != b) # evaluates whether two operands are _not_ equal
print(a < b) # less than
print(a > b) # greater than
print(a <= b) # less than or equal to
print(a >= b) # you can probably guess
```

## Assignment operators

These *assign* values to a variable:

In [ ]:
```
a = 10 # assigns the value 10 to the variable a
a += 5 # Adds the value on the right to the value in a and then assigns the new value t
print(a)

a -= 10 # subtracts the right value from the value in a and then assigns the new value
print(a)

a *= 5 # multiplies the right value with the value of a and then assigns to a
print(a)

a /= 5 # divides the value of a by the right value and then assigns to a
print(a)
```

## Logical and Membership operators

Logical operators help you compare different expressions

In [ ]:
```
a = True
b = True
c = False

# and: if both values are True then the condition is True
print((a and b))

# or: if _either_ value is True then the condition is True
print((a or b))

# not: reverses the logical state of the condition
print(not(a or b))
```

Membership operators test whether a value (operand) is a member of a collection (as in a string, list or tuple).

In [ ]:
```
fruits = ['apple','banana','pineapple']
a = "orange"

print(a in fruits)
```

```
print(a not in fruits)

print("o" in a)
```

These are the basic operators that you will need to know to create conditional expressions to guide your control flow.

## Looping/Iteration

## While loops

The while statement executes commands as long as an evaluated conditional expression remains true. This will loop through the code block until such time as the statement is no longer `True`.

In [ ]:
```
i = 1

while i < 10:
    print(i)
    i += 1
```

## For loops

The `for..in` statement also performs loops. In this case however, the loop *iterates* over a sequence or collection, and in doing so it assigns each element of the collection to a specific variable. Any collection that is *iterable* can be used to construct a for loop. In the example below `range(0,10)` creates an iterable collection of numbers from 0-9. Each instance of the loop places one of these values (in order) into the newly created variable `i`, and then executes the code block associated with this loop:

In [ ]:
```
for i in range(0,10):
    print(i)
```

An optional `else` statement can be used to execute a code block after the iterations have completed.

In [ ]:
```
genes = ['Gapdh','Mef2c','Pax6','Cxcl1','Msi1']

for gene in genes:
    print(gene)
else:
    print("No more genes!")
```

## You made it through this intro! Congrats!

You know now basic variable and data types as well as ways to extract or iterate over that information!

Let the TAs know if you have any questions and we will expand upon these lessons in class.