Python Fundamentals: First Interactions with Python

Welcome to the first module of Python Programming of the SHARCNET Summer School!! This lesson deals with the first interaction of Python starting at the built-in data-structures such as variables, lists, and dictionaries. We will build up the knowledge you need, even if is your first time doing serious programming.

Learning Objectives

- Interact with the python interpreter.
- Understand what is a variable, how to set it up, and manipulate it.
- Use lists, tuples, and dictionaries in Python programs.
- Read and manipulate data in the interpreter and in files.
- Write loops and decision statements in Python.
- Understand and write pure python functions.

What are the names of your instructors? (Answer on instructors cue)

- Ivan (yes)
- Sergio (no)
- Tyler (go slower)
- Tyson (go faster)

Why Python?

- Easy for humans to read
- Fast to code with it
- Reasonably efficient
- Growing community and community support

Python is a general purpose scripting language, and therefore you can do most tasks with it, from GUI development to HPC applications. Python is a high level programming language, which means that has strong abstraction from the computer code (a.k.a humans can easily read it). Python syntax is even more akin to regular English than other programming languages such as C or Java, allowing the programmer to focus more on functionality. This is also why is said to be extremely readable, and readability is encourage among the "Pythonistas". One particularly useful property of Python is that is fully cross platform, meaning that, done correctly, the same program can be run in Windows, Mac, or Linux operating systems.

Getting Started with Python

There are many ways to interact with python:

- Python console: Just type `python` in the terminal
- Ipython and variations: Provides a rich toolkit to help you make the most of using Python interactively (used here). After install, just type `ipython` or check Jupyter
- Others

Variables in Python

Let's imagine that you want to store some numbers. In Python:

```
In [ ]: ...
```

Types of Variables

In Python you can query the type a variable belongs to using the `type` functions.

Native functions in python are invoked with the parenthesis sintax:

In [ ]:
```
type(x)
type(y)
```

Basic types in python include:

- Integers: The integer part of a number
- Floats or Floating-Point Numbers: Decimal numbers
- Complex Numbers: Numbers including imaginary ones
- Strings: A sequence of characters
- Logical or Boolean: Has only two states `True` or `False`

In [ ]:
```
i = 10 # Integers
print(type(i))
f = 5.5 # floats
print(type(f))
c = 3j # complex
print(type(c))
s = 'ABC' # String
print(type(s))
b = True # Boolean
print(type(b))
...
```

Integers

As the name classifies this data type is reserved for integer numbers. As you possibly know, integers cannot not have decimal places, as this type of data is discrete. The distinction that Python (and most other programming languages) do between integers and decimals (called floats in python, see below) is that they occupy different amount of memory, and therefore they are different class of objects. However, you can do any mathematical operation using floats and integers.

Floats

As you probably already know, float is a data type designated for numbers with decimal places. `1.5` is considered a float, but also `1.0` . Both integers and floats have to be designated **without quotation marks**, otherwise Python will interpret them as strings. To understand this better let's do another exercise.

What is the value of `a` after: `a = b = 10`

- 0 (yes)
- 10 (no)
- error (go slower)
- None (go faster)

Modifying the Type: Casting

Some transformations of variables are allowed:

In [ ]:
```
...
```

So to transform variables to one another, they need to be of the same kind. A string with numbers can be transformed into floats or integers, but any other character can't.

Another example, let's transform the float in variable `y` into a string and an integer using the functions `str()` and `int()`:

```
y1 = str(y) # Transform float into string
y2 = int(y) # Transform float into integer
```

To recap, using the `print` function and the `type` function we learned before, print the type and values of `y`, `y1` and `y2`:

In [ ]:
```
print('y is of type', type(y), 'with value', y)
print('y1 is of type', type(y1), 'with value', y1)
print('y2 is of type', type(y2), 'with value', y2)
```

Operations with variables

Within type operations are allowed, but cross-type operations will depend:

In [ ]:
```
...
```

In [ ]:
```
x = 10,
y = 'string'
print( ... )
```

Likewise, operations (such as concatenation/addition, subtraction, etc) can only been done in same types. For example, mathematical operations can only been done between numeric types (complex, floats and integers), and not with strings.

To recap and a few points to note:

1. A float will be rounded down to the lowest integer when transformed to integer
2. An integer would be given a zero decimal when transformed to float
3. **Only** strings that contain numerical values can be transformed to either integer or float
4. *bonus*: `#` can be used in python to write comments in the code. I strongly recommend you thoroughly comment your code.
5. *bonus2*: When Python code fails, it output different kinds of error. We saw in points 8 and 9 that when the value is not correct we get a `ValueError`. Check the Python documentation for more errors!

Which code will correctlly print: "hello 1234"

- yes print("hello " + 1234)
- no print("hello " + string(1234))
- go slower print("hello " + str(1234))

Containers and Basic Data Structures

Variables are fine - but what if we want to "bundle" some of those variables?

How can I store them? For this, python has 3 basic containers: lists, dictionaries, and tuples.

Lists

A list is an ordered sequence of elements, and those elements are normally variables.

This data structure is mutable or changeable. Also you can iterate over it to retrieve your data or access one item at a time.

```
In [ ]: ...
```

So, lists are represented as comma separated variables between square brackets (`[var1, var2, var3]`). The variables can be of any type, including other containers, and objects.

Indexing: Retrieving elements from the list

To *index* the first element in the list we call the name of the list and the number of the element between square brackets

In [ ]:
```
...
```

To recap:

1. We can construct a list by typing variables or data between square brackets separated by comma:

```
# create a lits
alist = ['a', 'b', 3]
```

2. You can retrieve the items in the list by indexing the list:

```
# Retrieve the first element of the list
item1 = alist[0]
```

NOTE: Python indexing starts at 0

3. Now we can print the first element:

```
print('The first item is', item1)
```

4. What if I want say the first two? we can use a special kind of indexing called slicing:

```
print(alist[:2])
```

Note that you could write 0 or blank and in both cases it will retrieve the slice. Also note that the last index is not included on the result.

5. What if we want the last two? We can also have a slice if we use a negative number:

```
print(alist[:-2])
```

Iterating over a list

Oftentimes you will find yourself with very long lists. In these cases indexing and slicing the list one at a time is not very efficient.

In [ ]:
```
...
```

Imagine you want to go over a list, and apply and print a modification to each item. If you do it one by one, it will take a lot of code an a lot of time. Let's say that you have a list of numbers and you want to know what is the result of each item after elevating each number to the power of two and then subtracting two:

1. Let's create a list of numbers from 0 to 20. Python have a very convenient function called `range` to do this:

   ```
   numbers = range(10)
   ```

   However, the function `range` returns a special type of container also called so if we print this container we get:

   ```
   print(numbers)
   ```

2. Transform the range into a list. Luckily for you there is the function `list` which will give us the list of number in the range:

   ```
   numberlist = list(numbers)
   print(numberlist)
   ```

3. Now that we have the list we can go over it with a for loop. Let's first enumerate all the items using the `enumerate()` function and print each enumeration:

   ```
   # Iterate over the list enumerating the items
   for index, item in enumerate(numberlist):
    print('Item', index, 'is', item)
   ```

4. Now we can iterate over the list, compute the square of the number minus 2:

   ```
   # Iterate over the items and compute the squared minus 2
   for index, item in enumerate(numberlist):
    print('(item%d^2) - 2 =' % (index), (item**2)-2)
   ```

Recap we cover many new tricks!!:

- We can go over a list with a for loop
- Two new function: `range()` which creates a range of numbers, and `enumerate()` that enumerates every item in a container
- One subtle trick was displayed in step 4, did you see it? when we wanted to print the value of `index` inside of the string `(item%d^2) - 2 =`, we use a trick called string formatting, that allows you to include things in a string. You can use it to include strings by using `%s`, integers by using `%d`, and floats by using `%f`. The replacing variable have to be put outside of the string after a percentage sign (`%`). There are more advance usages and I will encourage you to check the Python documentation on string formatting, it is very useful!!

Now it is important to introduce mutability of containers. Containers such as *lists* and *dictionaries* are mutable, meaning that can be modified without re-writing them. *Tuples* on the other hand, cannot be modified. We will see more of *dictionaries* and *tuples* in their corresponding sections. For now, let's understand mutable lists through an exercise.

Mutating lists

Suppose that you have a list of three colors `Red`, `Blue`, and `Green`. Let's imagine that you actually want the primary colors. In this case you have to replace `Green` by `Yellow`.

In [ ]:
```
colorlist = ["Red", "Blue", "Green"]
for index, color in enumerate(colorlist):
    print('Color {} is in index {}'.format(i, color))
...
```

Let's also imagine that you want to add `Purple` and `Grey` at the end of the list:

In [ ]:
```
...
```

Now you get how to get items from a list through indexing. For completion sake, let's say we want to spell the last color in the list. Of course we can just write it down in a print statement if we know what it is, but in cases where you don't know you can do something like this:

```
# Get the last element with negative indexing
last = colorlist[-1]
# convert the string Gray into a list of characters using the function list
chars = list(last)
# print the spelling of the last item in colorlist
print('The color %s is spelled:' % last)
for character in chars:
    print(character)
```

So far we have seen that:

1. Any element in a list can be mutated (replaced) by something else without redefining the rest of the elements
2. Lists have a method called `append` to append elements at the end of the list
3. The function `list()` can be used to turn string into a list of characters

I encourage you to check the python documentation on lists and other methods that are very useful such as `extend()`, `pop()`, `index()`, etc.

Lists are very useful data structures, however, it can become cumbersome when you don't know the exact index and you want to access a given value. Looping over a very big list can also be very slow, if you want just a given set of values. For this, we can use another very useful data structure called dictionaries.

Dictionaries

A dictionary is basically a map between pairs of objects. It is also a mutable object, meaning that you can change its values at any time. Dictionaries work in a key-and-lock fashion, where with a particular key you point to a value:

In [ ]:
```
adict = {"Red":3, "Blue":4, "Green":5, "Yellow":6, "Purple":6, "Grey":4}
...
```

NOTE: Dictionaries are not sorted

To recap, different than lists, dictionaries are built either by having variables in between curly brackets (`{key1:var1, key2:var2, key3:var3}`), or by using the built in function `dict` (`dict(key1=var1, key2=var2, key3=var3`). As you can see, you need to provide a key and a value. You can access the value with that particular key. Dictionaries, like lists, also have useful methods that allow you to access the keys, the values, or both. This methods are called `keys()`, `values()`, `items()`. So if your dictionary is called `d`, and you want only the keys, you will call the method `d.keys()`. Likewise if you only want the values you can use `d.values()` to access them. If you want to get pairs, you can use `d.items()`.

Creating and looping over dictionaries

Let's assume that you are creating a phone book and want to retrieve and modify its contents:

In [ ]:
```
phonebook = {'Bob': 5146012356, 'Stacy': 9024896778, 'John': 9099788563}
# Retrieve Stacy's number, and modify it from 9024896778 to 9024896779
...
#print(phonebook['Kelly'])
```

For completion, let's imagine you would like to include other friend after the phone book is created without having to re-type all the entries.

```
# Add Kelly's number
phonebook['Kelly'] = 901526956
# Print Kelly's number
print("The added Kelly's number is", phonebook['Kelly'])
# Iterate over the key-value pair
for name, number in phonebook.items():
    print("%s's number is %d" % (name, number))
```

In this example we can see that:

1. We can access a dictionary's value with a key
2. Dictionaries have the attribute items, which returns the key-value pair
3. Dictionaries have the attribute keys which returns an iterable with the keys
4. Dictionaries can be modified by calling the key and assigning a new value
5. You can add a new key value pair by calling the dictionary with the new key pointing to a value
6. *Bonus*: Did you see that the last print have a string formatting with two variables? did you see that you can pass a tuple of arguments to that string formatting? We will get to tuples later, but keep an eye on it.

It is important to notice that dictionaries, unlike lists, do not keep any particular order in their keys.

Tuples

Tuples are a container just like lists, but are immutable.

This means that once it is created you cannot modify its contents without re-writing the object. To define `tuples` you can enclose a set of variables between parenthesis `( () )`

In [ ]:
```
# Define red fruits
red = ('Apple', 'Strawberry', 'Raspberry')
# Define green fruits
green = ('Pear', 'Avocado')
# Print them
print(red)
print(green)
```

Tuples are very useful when you need to guarantee that a given grouping will not be modified by the code. Tuples are also used in string formatting, and to pack and unpack variables (this is to advanced for this lesson, but you can go ahead and look it up!). Items stored in tuples can be accessed just like we did with lists, but unlike lists we cannot assign new content in the place of another.

In [ ]:
```
...
```

We can access the items by index, just like lists. Let's access the second element on the red fruits (Strawberry), and then let just enumerate both tuples:

```
# print the second element in red
print(red[1])
# Enumerate all fruits in red
print('Fruits in container red are:')
for index, fruit in enumerate(red):
 print('Fruit %d in red is %s' % (index, fruit))
# Enumerate all fruits in green
print('Fruits in container green are:')
for index, fruit in enumerate(green):
 print('Fruit %d in green is %s' % (index, fruit))
```

But tuples cannot be modified, but is it true? let's try to replace Apple with cherry:

In [ ]:
```
red[0] = 'Cherry'
```

Booleans

A boolean expression (or logical expression) evaluates to one of two states true or false

In [ ]:
```
...
```

Every object has a boolean value. The following elements are false:

- None
- False
- 0 (whatever type from integer, float to complex)
- Empty collections: “”, (), [], {}
- Objects from classes that have the special method **nonzero**
- Objects from classes that implements **len** to return False or zero

Evaluating booleans: Conditional statements

The importance of booleans is that you can evaluate variables and execute conditional code:

In [ ]:
```
...
```

Several operations also evaluate logically: Comparison, membership, and identity.

In [ ]:
```
...
```

Comparison operators The <, <=, >, >=, ==, != operators compare the values of 2 objects and returns True or False. Comparison depends on the type of the objects. See the Classes to see how to refedine the comparison operator of a type.

```
10 == 10
10 <= 10
```

Chaining comparison operators Comparison operators can be chained. Consider the following examples:

```
>>> x = 2
>>> 1 < x < 3
True
>>> 10 < x < 20
False
>>> 3 > x <= 2
True
>>> 2 == x < 4
True
```

The comparison is performed between each pair of terms to be evaluated. For instance in the first example, 1<x is evaluated to True AND x<2 is evaluated. It is not as if 1<x is evaluated to True and then True<3 is evaluated to True !!! Each term is evaluated once.

Evaluation of logical and comparison operators and membership operators The evaluation using the and and or operators follow these rules:

and and or evaluates expression from left to right. with and, if all values are True, returns the last evaluated value. If any value is false, returns the first one. or returns the first True value. If all are False, returns the last value

Membership operators in evaluates to True if it finds a variable in a specified sequence and false otherwise. not in evaluates to False if it finds a variable in a sequence, True otherwise.

```
>>> 'good' in 'this is a great example'
False
>>> 'good' not in 'this is a great example'
True
```

Identity operators is evaluates to True if the variables on either side of the operator point to the same object and False otherwise is not evaluates to False if the variables on either side of the operator point to the same object and True otherwise

```
>>> p = 'hello'
>>> ps = p
>>> ps is p
True
```

What is printed in the following code snippet?

In [ ]:
```
a = 10
b = 'hello'
c = 'hello world'
if a > 20:
    print("YES")
else:
    if "cat" in c:
        print("NO")
    elif b in c:
        print("Go slower")
    else:
        print("Go faster")
```

Converting Between Data Structures

You can convert a tuple to a list, a list to a tuple, and you can convert a list of tuples into a dictionary. You can also convert a dictionary to a list, but only the keys will be displayed.

In [ ]:
```
fruits = ('Apple', 'Banana', 'Pear')
fruits = list(fruits)
...
```

In [ ]:
```
# create a tuple
fruits = ('Apple', 'Banana', 'Pear')
#loop over it and print the fruits
print('Fruits available in tuple:')
for fruit in fruits:
    print(fruit)
...
```

You can also create a list of tuples and transform it into a dictionary using the `dict()` function. Also, let's use the `list()` function to make a list of the keys in the newly created dictionary:

```
# Create a list of tuple pairs and convert it to dictionary
fruitsncolor = [('Apple', 'Red'), ('Banana', 'Yellow'), ('Pear', 'Green')]
print(fruitsncolor)
# Convert the list of tuples into a dictionary and print it
d = dict(fruitsncolor)
print(d)
# Print the list of keys
print('Fruits in dictionary', list(d))
```

Here we just showed you that if you provide the key-value pair in a list, you can generate a dictionary with the `dict` function. I also showed you that you can go back and forth from tuples to lists with the functions `list` and `tuple` .

Recap

In this section we also saw that lists are represented with square brackets (`[]`), dictionaries with curly brackets (`{}`) and tuples with parenthesis (`()`). All these data structures (including strings) have very useful functions that we will not cover here, but I encourage you to explore before continuing. You can visit the Python documentation (https://docs.python.org/3/tutorial/datastructures.html) for more information.

Loops

When you have to repeat an action a number of times, you need to *loop* over the task. We inderectly have seen one kind of looping before called *for loop*

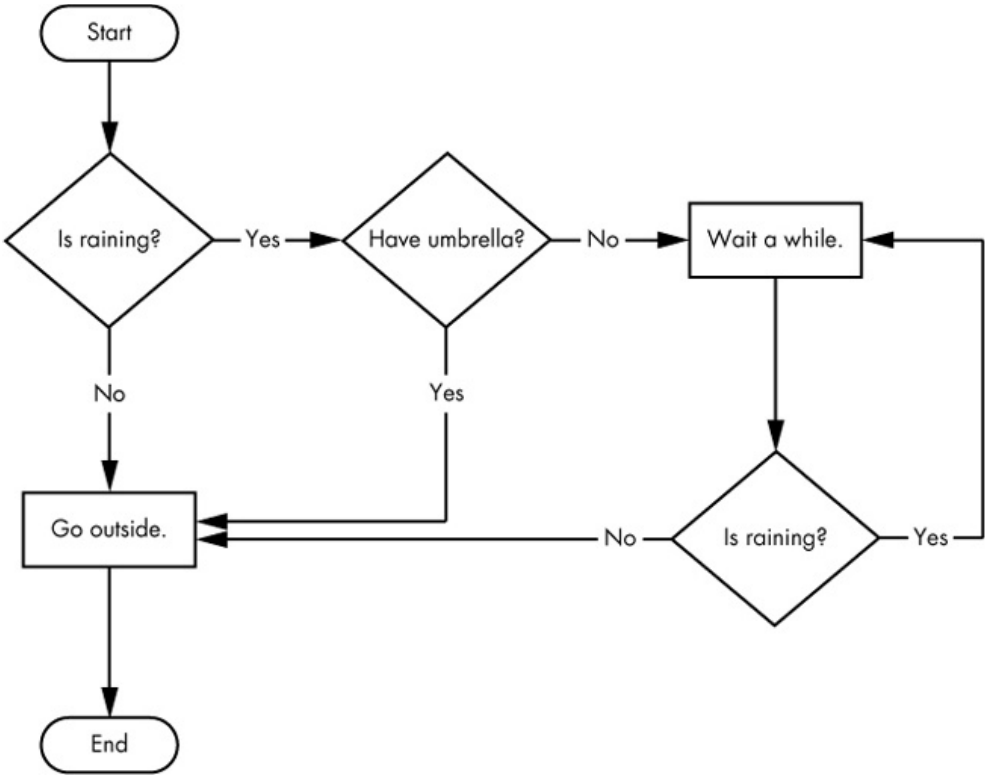
In [ ]:
```
n_simulations = 10
for simulation in range(n_simulations):
    print('Performing simulation', simulation)
```

What about if you do not have a set number of items you want to loop? what if you want to execute lines of code until some condition becomes False?

while loop

In [ ]:
```
current_number = 0
while current_number < 10:
    print("Current number is", current_number)
    current_number += 1
```

Flow control



In [ ]:
```
...
```

BEWARE of while true or while number, it will create infinite loops, unless you break it.

What does the final_list look like after the following code snippet?

In [ ]:
```
some_list = []
for i in range(10):
    some_list.append(i*2)
final_list = []
for i,ele in enumerate(some_list):
    if i < 4:
        final_list.append(ele)
```

yes [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

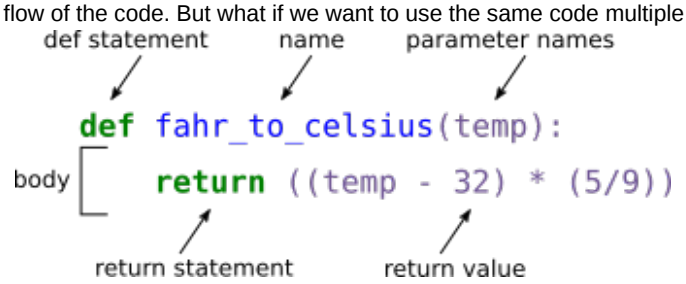
no [0, 2, 4, 6]

go slower []

go faster Error

Functions

We have seen so far how to store data into variables and containers. We also have seen how to go over lots of data using loops, and how to control the flow of the code. But what if we want to use the same code multiple times?



In [ ]:
```
def fahr_to_celsius(temp):
    return ((temp - 32) * (5/9))
...
```

Scope and Lifetime of variables

Scope is where the variable is recognized and the lifetime of a variable is how long you can access it. We can have local and global scopes and lifetimes:

In [ ]:
```
def my_func():
    x = 10
    print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
...
```

So, if you define a variable within a function (Local variable), it will be only recognized within that function and will destroyed once you exit the function.

Which inputs in the following function header have default values?

```
def my_func(some, thing, goes=100, right=200, here=300)
```

- yes — All
- no — goes, right, here
- go slower — It's complicated...
- go faster — Why is Python like this...?

Base problem

Scenario:

You have a list of 10 genes names, a second list with the number of copies of each gene in species *x*, and a new gene and copy count. You want to create a container that would allow you to access the number of gene copies by calling the gene name. You also received a third list matching the same gene names but with counts for species *y*.

Aim:

1. Create a container that would allow you to access the number of gene copies by calling the gene name by species name.
2. Filter for genes that have more than 100 counts on species x.

HINT: You will be creating a dictionary of dictionaries

Steps

1. Create a variable called `genes` containing the following gene names: `COI`, `Cytb`, `12S`, `18S`, `IgA`, `A1BG`, `ZZZ3`, `GDF6`, `CNN1`, `MKNK1`.

In [ ]:
```
genes = ['COI', 'Cytb', '12S', '18S', 'IgA', 'A1BG', 'ZZZ3', 'GDF6', 'CNN1', 'MKNK1']
```

1. Create a second list called `spx_counts` with the following counts (**NOTE: counts are made up**): `1000`, `1000`, `5000`, `500`, `54`, `35`, `564`, `6`, `45`, `68`.

In [ ]:
```
spx_counts = [1000, 1000, 5000, 500, 54, 35, 564, 6, 45, 68]
```

1. Create a third list called `spy_counts` with the counts of gene copies for species *y*: `1050`, `1050`, `3452`, `315`, `45`, `45`, `345`, `5`, `78`, `15`.

In [ ]:
```
spy_counts = [1050, 1050, 3452, 315, 45, 45, 345, 5, 78, 15]
```

1. Create a tuple with a new pair of gene name and its count for species *x* called `new_entry`: `('ALPI', 789)`

In [ ]:
```
new_entry = ('ALPI', 789)
```

1. The new entry has a mistake, it was not the `ALPI` gene but instead it was the `ALPG` name. Correct it.

In [ ]:
```
new_entry = list(new_entry)
new_entry[0] = 'ALPG'
new_entry = tuple(new_entry)
```

1. Create a dictionary for each species

In [ ]:
```
dx = {} # Empty dictionaries to start
dy = dict() # This is also a empty dictionary but with the constructor
for index, gene in enumerate(genes):
    # we could filter here ...
    dx[genes[index]] = spx_counts[index]
    dy[genes[index]] = spy_counts[index]
```

1. Add the new entry of species *x* in the appropriate dictionary

In [ ]:
```
dx[new_entry[0]] = new_entry[1]
```

1. Create a dictionary where the keys are the species names and the values are the dictionaries with the counts

In [ ]:
```
big_dict = {'x': dx, 'y': dy}
```

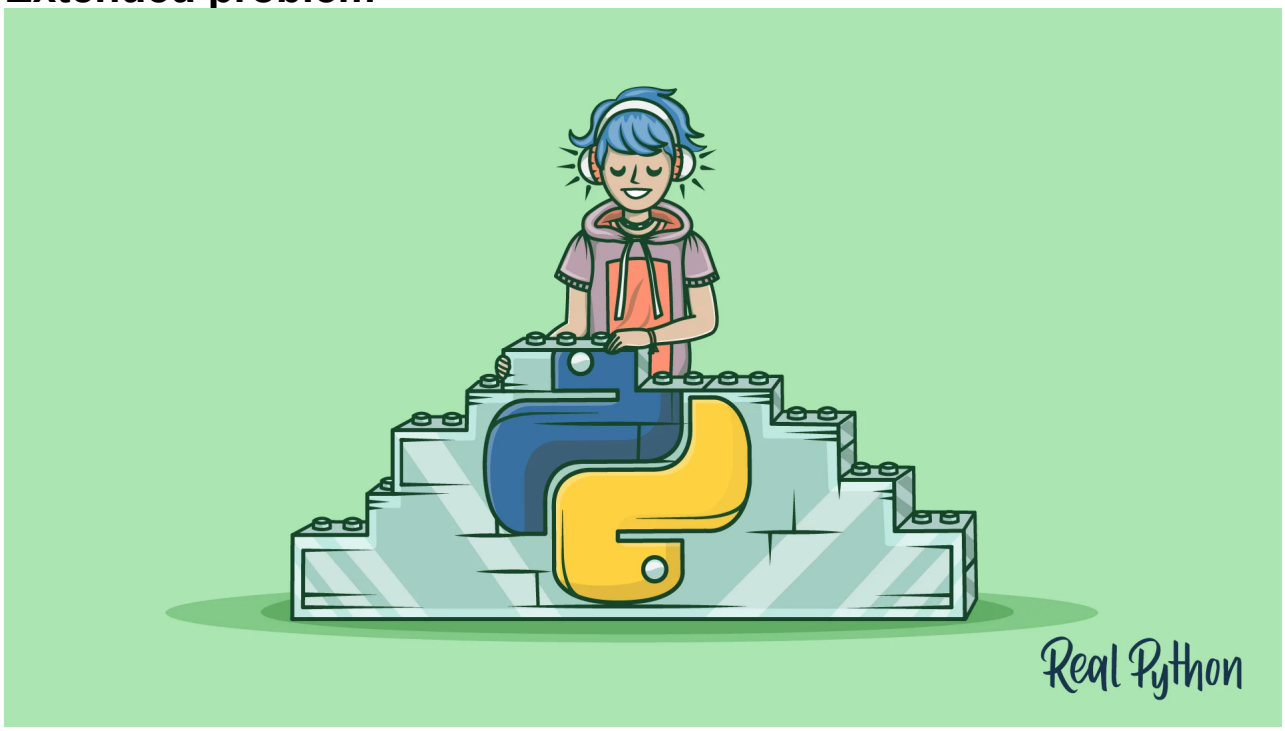
1. Print the counts in both species for the gene `CNN1`

In [ ]:
```
for sp, dictionary in big_dict.items():
    print('The count of CNN1 in species %s is %d' % (sp, dictionary['CNN1']))
```

1. If you did not filter in step 6, let's try filtering the big dictionary:

In [ ]:
```
filtered = dict(x={}, y=dy)
for gene, count in big_dict['x'].items():
    if count >= 100:
        filtered['x'].update({'gene':count})
...
```

Extended problem



Now, based on the big dictionary we have created, print only genes that start with a `C` **AND** that have more than 500 counts in species x and more than 500 counts on species y.

If you have fallen behind, or need to start fresh, the moodle page will explain how to begin the extended problem.