Background Reading #1

Table of Contents

1. GitHub Workflow

1.1 Key Terms

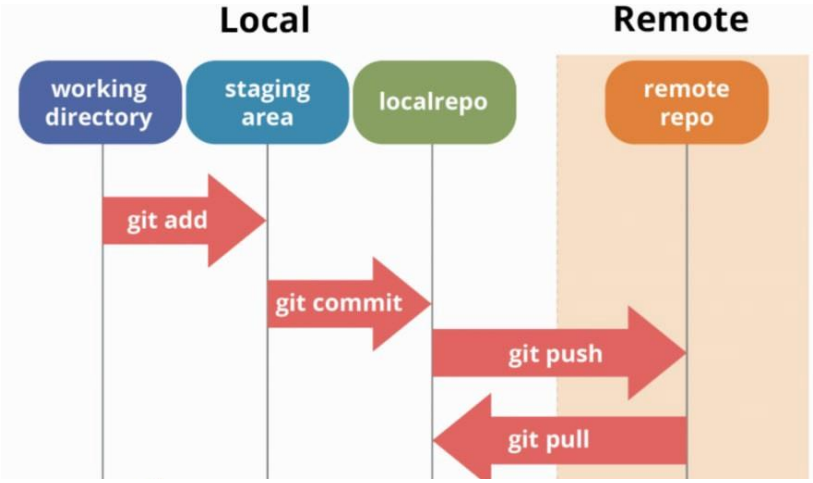
Term	Definition
Local	The files that live on your local PC (mainly .py code files, but could also be spreadsheets, image files, etc.)
Remote	The data files that live in the repository hosted on the GitHub website
Commit	Snapshot of all the files in the entire project
Push	Send commit to the online repository
Pull	Pull commits made by others (e.g., instructor/TA)

If you have ever versioned a file (E.g., click here), you have basically made a commit, though only for a single file. A Git commit is a global save of all project files. This is done for a version that is significant and you may want to inspect or revert to later. Here is a summary of how different saving modes work:

1. Changes are automatically saved locally in your Eclipse workspace as you work on your code file. While this is useful, it doesn't help you if something happens to your PC and you can't access Eclipse from that specific device.

2. Eclipse (the integrated development environment) will show you all the files that you have made changes to in the **Git Staging View**. **Commit** asks Eclipse to save all of these changes and store them in your local repository folder. This is still just a local save; however, the act of committing packages your changes up and makes them ready to be sent to the remote (cloud) storage on the GitHub website.

3. We will exclusively be using the **Commit and Push** option from the Git Staging View. Pushing = sending the saved bundle/commit to your remote GH repository. If something happens to your device and you did not **Commit and Push** the changes to GitHub, only the previous pushed commit will be available online. The figure below summarizes this.



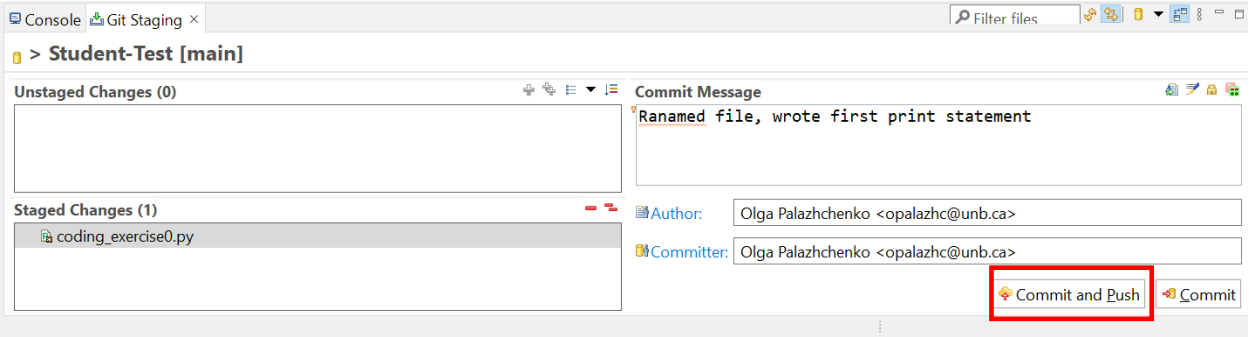
1.2 Commit and Push to Remote GitHub Repository

I recommend committing and pushing your changes to GH at least once a day (at the end of the day) or whenever you plan to step away from your code for a period of time.

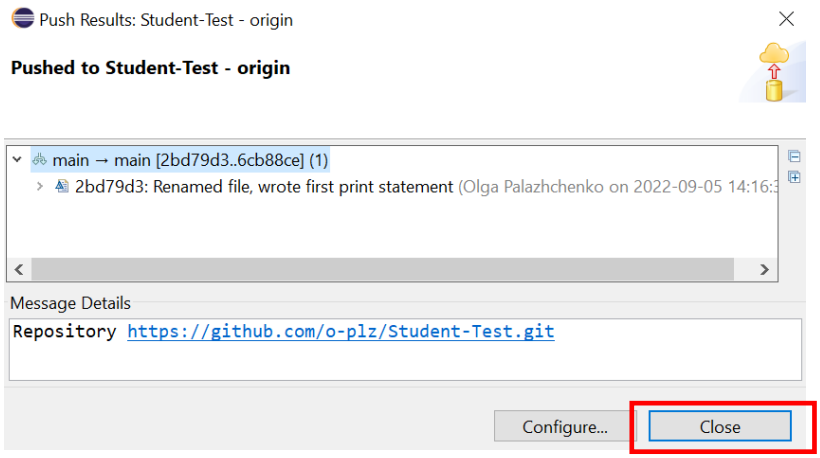
1. Make sure the **Git Staging** view is on (in Eclipse: Window > Show View > Other … > Git Staging). Drag all .py files you want to send to GH from the **Unstaged Changes** window to the **Staged Changes**. You'll notice that the name of your repository is shown at the top of the Staging View, beside the little yellow cylinder icon (indicating a repository)

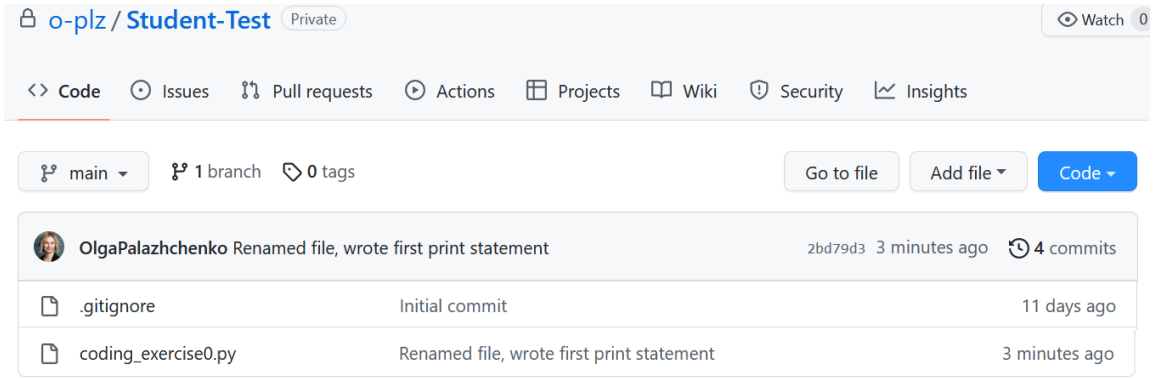
2. Write a short, but useful **Commit Message**. This will show up online on GitHub with the timestamp of the commit. Keep your message to one line. If you find yourself needing to write longer messages, you should be committing changes more frequently!

3. Select **Commit and Push** (shown in red box below). This sends changes to the online repo.

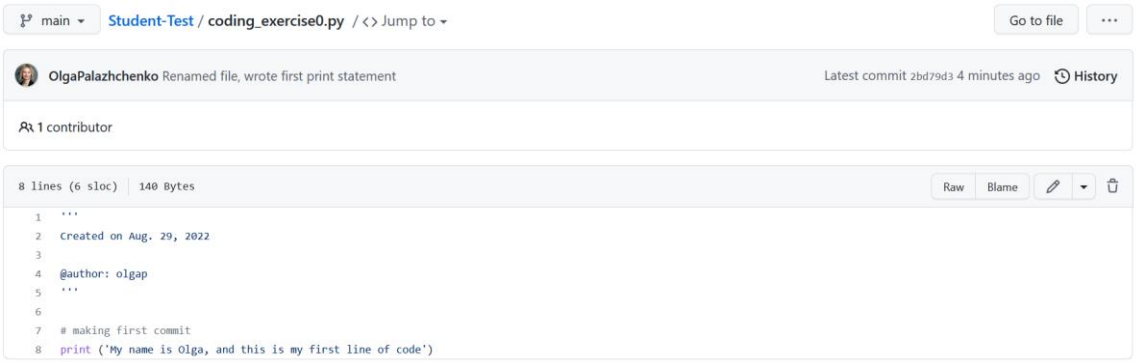


4. A summary message box will pop up. Click **Close**. Next, go to GitHub to see the commit.



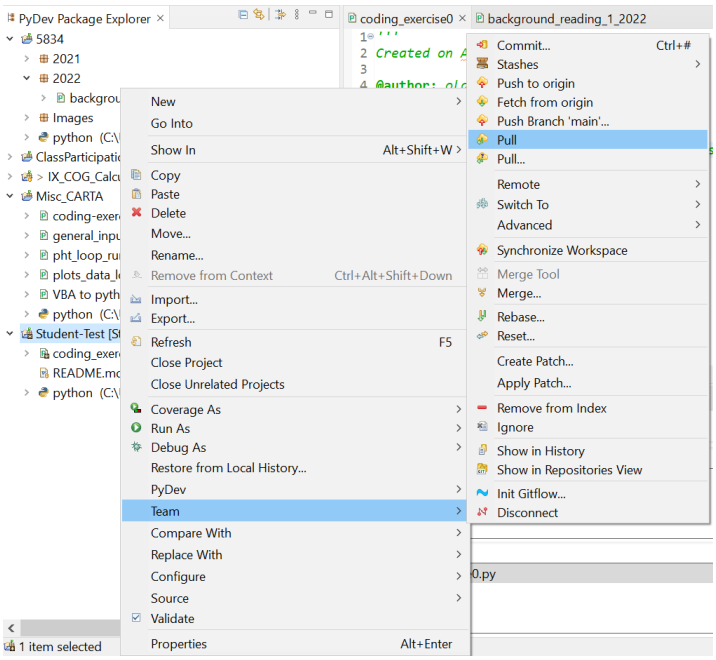


Clicking on the file name, (**coding_exercise0.py**) shows the source code. You can also click on the commit message itself, which specifies (with colour-coding), additions, deletions, etc.



1.3 Pulling from Remote Repository

From inside Eclipse, right click on the repo you wish to pull content from (Student-Test in this example), click on **Team**, then **Pull** (the "**Pull…**" is for custom settings, which we don't need). Any changed files (or new files) will now appear in that repo in your Eclipse workspace!



2. LET'S START CODING!

The source code from exercises below is available by "pulling" from the **.sample-code-solutions** class repository (located in the **fa-2022** team in our **o-plz** organization). Pulling the .py file allowing you to follow along and play around with the file we have created for this Background Reading.

2.1 Command and Shortcut Summary

Command/Symbol	Definition
print()	Prints whatever is inside brackets
float()	Converts object inside brackets to a float, e.g., 1.0
int()	Converts object inside brackets to an integer, e.g., 1
str()	Converts object inside brackets to a string (non-numeric data type)
len()	Calculates length of object. Typically used for lists to count how many items are in a list or number of characters in a string.
input()	Prompts user for input. Typically string inside brackets has instructions for input. User input is typed into the console
' ' or " "	Object surrounded by single or double quotes is a string data type
''' '''	Multi-line strings, without requiring a line break to continue the string
=	Assigns an object to a variable
[ ]	Indicates that a list is contained inside or placed at the end of the variable assigned to that list to locate a element of the list, e.g., Item_list = [1, 2, 3] or print(Item_list[0])
#	Indicates the start of a comment
\	Adds a line break without interrupting code sequence
*	Multiplication
**	Exponent/to the power of
for	Signals the start of a loop. Continues until a defined iteration. Used often with the range function.
while	Signals start of a loop. Continues until break command.
range(start, stop, [step])	Sequence that starts at start and counts up to stop - 1 by step.
zip()	Use in iterating over multiple lists at the same time, e.g., for item1, item2 in zip(list1, list2): [do something]
list.append()	Adds the item in the round brackets to the end of a list. The list name is written prior to the dot (here, this is a list called "list")
list.remove()	Removes the item in the round brackets from its location in the list. The list name is written prior to the dot (here, this is a list called "list")
Eclipse Shortcuts	**Function**
Ctrl + /	Highlighting section of code and using command converts section into a comment (clicking on single line only "comments out" that line)
Ctrl + \	Same as above, but turns commenting off (removes # from line start)
Ctrl + F	Opens search/replace window

2.2 Comments

A comment is an annotation in your code that is used to

- Inform the reader (often yourself), of what a particular piece of code is trying to do,
- Indicate the designed output, result, etc. of a part of the code, and most importantly,
- Make the code more readable.

Comments can also be useful to remind you to come back and clean up a part of your code that doesn't work (or can be made more efficient), or to explain to your future self why the code is written in such a way.

We use the pound (aka hashtag) symbol, #, to comment out the rest of a line: everything that follows the # is ignored by Python. Therefore, you can put little notes to yourself or others about what is going on in the code.

We can comment out any part of our code or un-comment it later. In Eclipse, we can also highlight the section of text we want commented out and use the shortcut **Ctrl + /** to toggle comments on (this will automatically place the # in front of each highlighted line) or **Ctrl + ** to toggle comments off. This is especially useful when going through the code file for this exercise, where you can comment out all the lines except the ones you are wanting to run/test.

```
#this code multiplies two numbers together
x = 9 * 9
```

2.3 Indentation

Python is slightly picky about how you lay out your code. It requires that code "blocks" are properly indented. Code blocks will come up again later, especially when we get to loops and if statements. For now, keep in mind that there is no need to indent or add additional spaces to any of your lines of code unless Your code does need to be lined up to work properly. Indentations or gaps before the code begins on a given line is also known as "whitespace". There shouldn't be any unnecessary whitespace.

```
#this will work because there is no weird spacing before the code starts
print('indentation is important')

#this has been indented/tabbed by one level
    print('indentation is important')
```

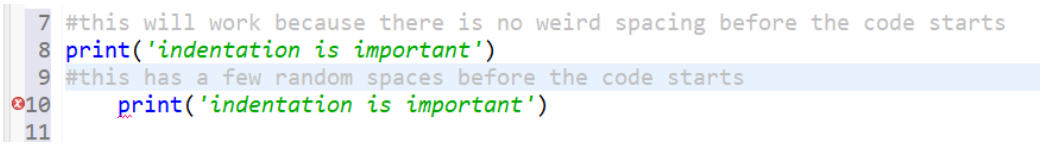
Console Output:

```
  File "C:\Users\olgap\eclipse-workspace\CHE5834 Exercises\Strings.py", line 10
    print('indentation is important')
IndentationError: unexpected indent
```

We note that the error tells us the line number where the problem was found, the command that caused it, and a decent explanation of what went wrong (unexpected indent). We will see later that this is a good error, in that our editor can easily call it out without letting the code proceed.

Eclipse will also red underline the spacing error (see screenshot below) much like when the spelling of a word is not recognized in Microsoft Office. You will also see a red x at the line where Eclipse has identified an error, letting you know that something is wrong before you even run the code and see the error in the console.

To run the code without the error, either remove the incorrect indentation/whitespace or "comment out" (convert the line of code to a comment) that line by manually typing a pound symbol, #, in front, or clicking on the line with the incorrect indentation and using the ctrl + / keyboard shortcut.



You'll also notice that Python is case-sensitive where built-in functions (or classes, but more on this later) are used. For example, the command is print, not Print.

2.4 Printing, Data Types, and Variables

A string is a data type that is a collection of characters. These can be letters, numbers, or special characters. We tell Python that we are using strings by using quotes (either single or double quotes can be used).

The first command we will learn in Python is **print**. The print command takes a comma separated list of objects to print to the screen (to the console in Eclipse). We'll start by first printing strings of characters, but anything we wish from our codes can be printed. In Python 3 (this is different for Python 2, but we don't need to worry about this), we enclose the things we are printing by round brackets.

Things in the double or single quotes are taken as is by Python, meaning a print statement will output the literal characters. When printing, each object is separated by a space by default, meaning that both of the following commands result in the same output.

```
#this will show the difference between printing one versus multiple strings
print('I love nuclear engineering')
#default spaces between different strings printed
print('I', 'love', 'nuclear', 'engineering')
```

Console Output:

```
I love nuclear engineering
```

Going forward, we will use print statement a lot to test (or debug) parts of our code. It is often convenient to comment these out temporarily while testing other parts of the code. We can then return and uncomment some of our commands if we need to use them again. This saves a lot of time on retyping commands.

2.4.1 Variables

An identifier or name for information that we want to store in our code for later use is called a variable. We assign this information to a variable using the **=** sign. Once we begin to develop more sophisticated codes, the importance of naming our variables such that they tell us as much information as possible while still being concise will become clear. For now, we will use very simple variable names.

We can store different data types using variables, including strings. Putting code in quotes will results in converting that code to a string (collection of characters).

```
x = 'Nuclear Engineering is a cooler course than Nuclear Reactor Physics'
print(x)
#printing the character x
print('x')
```

Console Output:

```
Nuclear Engineering is a cooler course than Nuclear Reactor Physics
x
```

2.4.2 Subsets

With strings (and other data types) we can also subset. This means access only some of the characters in the string. We do this using square brackets. Putting a single number in the brackets gives you the character in that position. We will use this idea a lot going forward, as it also applies to obtaining an object from a list or array of objects.

Note: Python starts numbering at 0 so that 0 is the first character in a string. This is different from other programming languages, so be careful!

You can also get a range of characters in a string using the colon. The colon operator is non-intuitive in that [a : b] accesses the elements from position a (some integer) to position b (some integer -1).

```
#prints the first letter in the string
print(x[0])
#prints the last letter in the string
print(x[-1])
#determines the length of the string
y = len(x)
print (y)
#prints the entire string
print(x[0:67])
# prints the entire string
print(x[0::])
```

2.4.3 Overloading Strings

Performing mathematical operations on strings is referred to as overloading. Since these are not numbers, they cannot be added, subtracted, multiplied, etc. as numbers are. For example, the + operator concatenates (or smushes together) the strings/characters it operates on, while the * or multiplication operator creates the number of multiples by which the string (or list) is being multiplied.

```
x = 'checking how '
y = 'math operators work'
z = x + y
print(z)

z = x * 3
print (z)

z = x * 4.1
```

Console Output:

```
checking how math operators work
checking how checking how checking how
Traceback (most recent call last):
  File "C:\Users\olgap\eclipse-workspace\CHE5834
Exercises\CodingExercise1 Practice.py", line 56, in <module>
    z = x * 4.1
```

In the last example, multiplying a string by a non-integer number gives an error, as this doesn't make sense from an overloading perspective.

2.5 Numerical Data Types

Integers are whole numbers, including the negatives. They never have a fractional, or decimal, part and should only be used for a count, e.g., number of times we iterate through a calculation, the number of elements in an array, etc. Floating point numbers are numbers that do have a fractional part. Most numbers in engineering calculations are floating point type.

2.5.1 Converting Between Data Types

Sometimes it is useful to convert a string to an integer or vice versa. We simply use the commands float(), int(), or str(), with the object you are converting placed inside of the round brackets.

```
#assigning the string 2 to the variable y
y = '2'
print(y)
#converts y from a string to a float.
# Reassigning the variable y changes its identity after this point
y = float(y)
print(y)
```

Console Output:

```
2
2.0
```

Note: one of the handy things in Eclipse is clicking on a any object highlights all other locations of that object in your code. We can also use **Ctrl + F** to call up a traditional search box, but the highlighting gives a quick glance at nearby mentions of that object (e.g., the word “float” is highlighted in the block of code above).

2.6 Basics of Lists

A list is a Python object that contains several other objects. A list is defined by enclosing the list members in square brackets. These objects can be strings, numbers, or other objects. We can access items in a list using square brackets:

```
listofstrings = ['A', 'B', 'C']
lengthlist = len(listofstrings)

print(listofstrings[1])
print(listofstrings[lengthlist-1])

listintegers = [1, 5, 9]
print(listintegers[0])
```

Try running these on your own!

2.7 For Loops and Range() Function

There are often situations when we need to iterate a fixed number of times. It is useful to have a The **for** loop is built for such a situation. One way we often use for loops is with the **range()** function.

This function takes 3 input parameters: range(start, stop, [step]). The stop parameter tells range to go to the number right *before* stop. The parameter step is in brackets because it is optional. If

you do not define it, Python assumes you want to count by 1 (i.e., step by 1). What range returns is a sequence that starts at start and counts up to stop - 1 by step. If you don't indicate the start, Python starts counting at 0.

```
# for loop examples
for i in range(1, 5):
    # starts counting at 1 because start specified
    print (i)

# line break that shows up as empty space in console
print ('\n')

for i in range(5):
    # by default, starts count from 0 because start not specified
    # combining with if statements
    if i <= 2:
        None
    else:
        print (i)
```

Console Output:

```
1
2
3
4


3
4
```

We can also iterate over a list. Later, we'll see that this can be a list of anything…variables, strings, numbers, functions, plots, etc. This is the beauty of object-oriented programming, which you may have heard that Python uses. For now, we'll start with a list of numbers.

```
FavNumbers = [1, 5, 1988]
for i in FavNumbers:
    x = i * 4
    print (x)
```

Console Output:

```
4
20
7952
```

2.8 Useful List Operations

2.8.1 Adding and Removing Items from Lists

To add or remove items from a list (e.g., let's say we have a list called list_var) use the **append** and **remove** commands with syntax of the form

list_var.append(item_add)

list_var.remove(item_remove)

where item_add and item_remove are items to add or remove from the list “list_var”, respectively. Appending to a list adds the element to the end of the list. We can also start with an empty list by assigning a variable to an empty set of square brackets.

```
new_list = []

for i in range(5):
    new_list.append(i)

print (new_list)

new_list.append('random string')
new_list.remove(2)

print (new_list, new_list.index(3))
```

Console Output:

```
[0, 1, 2, 3, 4]
[0, 1, 3, 4, 'random string'] 2
```

We can tell the remove command exactly which item to remove if we know for sure that it is in a list, or we can ask for the location (index) of a certain item in a list (if we know it’s there).

In the above example, we ask for the index of the number 3, which is in the list “new_list”. We are told that its location (after we have done some appending and removing of items) is 2, meaning it is the third entry when we count from 0.

We can also combine this idea with an if statement to check if something is **in** a list.

```
if 3 in new_list:
    print('this item is in the list')
```

Console Output:

```
this item is in the list
```

2.8.2 Iterating Over Multiple Lists Simultaneously

Let’s make two new lists:

```
new_list2 = [1, 3, 5, 6]
new_list3 = ['cat', 'dog', 'goose']
```

The **zip**([list1, list2, ….list_n]) function takes lists as input and makes them ready to be simultaneously iterated over. This allows us to loop through multiple lists at once, where the same index is accessed from each of the multiple lists as we march through them. Simply “zipping” together multiple lists will combine them into something called an “iterator”. For example:

```
zipped_list = zip(new_list, new_list2)
print (zipped_list)
print (list(zipped_list))
```

Console Output:

```
<zip object at 0x0000026019288580>
[(0, 1), (1, 3), (3, 5), (4, 6)]
```

The first print statement gives a unique object code, meaning what Python has assigned to this new box that holds stuff (our lists). This is not the most useful here, but also doesn't mean the code is doing anything wrong. It means the zipped object only really wants to be iterated over, not have anything else done to it. We can print the contents of it by using the list() function. Note that zip pairs together the first elements of both lists, then the second elements, then the third, etc. This sets us up to be able to automatically march through multiple lists, element by element.

The zip function stops when the last list index (list item number) of the smallest list has been reached, meaning it runs until the smallest of all the lists has reached its end. If all your lists are the same length, this does not matter. If your lists are different lengths and you don't know it, this could result in a silent error - only some of the list items had an operation performed on them.

Let's print all three lists and inspect them. It is easy to spot the differences by visual inspection here, as our lists are short, but simply printing may not be useful if your lists are many 100s or 1000s of entries long. We can also check the length of the lists. We can combine our diagnostics a little more interestingly using something known as a tuple, or list of lists.

```
title_lists = ['list 1:', 'list 2:', 'list 3:']
all_lists = [new_list, new_list2, new_list3]

for i, j in zip(title_lists, all_lists):
    print (i, j, len(j))
```

Console Output:

```
list 1: [0, 1, 3, 4, 'random string'] 5
list 2: [1, 3, 5, 6] 4
list 3: ['cat', 'dog', 'goose'] 3
```

We can immediately see that we have three different lengths of lists. When we try to print all the lists elements, we will expect that only the first three will be printed, as the smallest list (**new_list3**) only consists of 3 list elements.

```
for x, y, z in zip(new_list, new_list2, new_list3):
    print (x, y, z)
```

Console Output:

```
0 1 cat
1 3 dog
3 5 goose
```

2.9 Built-In Python Functions

We've already seen some build-in function that were included with the Python download (**zip()**, **list()**, **range()**, etc.): https://docs.python.org/3.3/library/functions.html.

There are also some functions that are the Python standard library (came with Python) but need to be imported into the specific code file we want to use them.

We can access many more functions from package libraries that we can ask Eclipse to download from the internet, which we will do in the next Coding Exercise.

We access the functions in a library by using the **import** command. For instance, let's try "**import random**." It is good practice to have all imports at the very start of the file (right below the auto-generated, multi-line string of date the file was generated and the author).

```
'''
Created on Sep. 3, 2022

@author: olgap
'''

import random
```

Back to where we were in our code (following previous example), **random** is a type of object called a class (like a function, but fancier) that can generate a random number. This will have a few of its own functions. For example:

random.randrange(stop) Return a random integer N such that N < stop

random.randint(a, b) Return a random integer N such that a <= N <= b.

```
print (random.randrange(10))
print (random.randint(0,9))
```

Console

```
1
9
```

3. REFERENCES

"Computational Nuclear Engineering and Radiological Science Using Python," Ryan G. McClarren, Elsevier, 2017.