# Homework 5

**Due Saturday October 19 by 11:59pm on Gradescope**

Note that the directory `hw05/` contains `data.zip`. Run the following command to unzip the files and directories in `data.zip`.

```
In [ ]: !unzip data.zip
```

You should have a directory `data/` containing file `planets.txt` and directory `university/`. Please import the following package.

```
In [ ]: import numpy as np
```

## 1. Lists vs Arrays

While lists and arrays have some similarities, lists and arrays are different data types supporting separate operations. In this problem, we want to go about coding a function in two ways -- one way with lists and the other way with arrays. For each approach, we can take advantage of the operations that apply either to lists or to arrays.

### *Question 1.1*

Write a function that take as input two lists, `x_list` and `y_list`, and computes $x^2 + y^3$ for corresponding each x, y in the two lists. Assume both lists have the same length, and the output should also be the same length. Try using **zip**.

```
In [ ]: def list_func(x_list, y_list):
            # BEGIN SOLUTION

            # END SOLUTION
```

```
In [ ]: # Test cases
        assert tuple(list_func([1, 2], [3, 4])) == (28, 68)
        assert tuple(list_func([1, 2, 3, 4], [2, 3, 4, 5])) == (9, 31, 73, 141)
```

***Question 1.2***

Write a function that take as input two lists, `x_list` and `y_list`, and computes $x^2 + y^3$ for every possible pair of x, y in the two lists. You should iterate over the $x$s in the outer loop and $y$s in the inner loop. Do not assume both lists have the same length. Try using **list comprehension** (see Lutz *Learning Python* Chapter 20)

In [ ]:
```
def list_combination_func(x_list, y_list):
    # BEGIN SOLUTION

    # END SOLUTION
```

In [ ]:
```
# Test cases
assert tuple(list_combination_func([1, 2], [1, 2])) == (2, 9, 5, 12)
assert tuple(list_combination_func([1, 2, 3, 4], [2, 3, 4])) == (9, 28, 65, 12
, 31, 68, 17, 36, 73, 24, 43, 80)
assert tuple(list_combination_func([], [1, 2])) == tuple()
```

***Question 1.3***

Write a function that inputs two lists, `x_list` and `y_list`, converts them to arrays, and computes $x^2 + y^3$ for corresponding elements x, y in the two arrays. Assume both lists have the same length. Do not use a loop. Instead take a sum of the arrays. The output should be an array.

In [ ]:
```
def arr_func(x_list, y_list):
    # BEGIN SOLUTION

    # END SOLUTION
```

In [ ]:
```
# Test cases
assert (arr_func([1, 2], [3, 4]) == [28, 68]).all()
assert (arr_func([1, 2, 3, 4], [2, 3, 4, 5]) == [9, 31, 73, 141]).all()
```

***Question 1.4***

For ten numbers, `list_sum` and `array_sum` both take a similar amount of time.

In [ ]:
```
sample_list_1 = list(range(10))
sample_array_1 = np.arange(10)
```

In [ ]:
```
%%time
list_func(sample_list_1, sample_list_1)
```

In [ ]:
```
%%time
arr_func(sample_array_1, sample_array_1)
```

The time difference seems negligible for a list/array of size 10; depending on your setup, you may even observe that `list_sum` executes faster than `array_sum`! However, we will commonly be working with much larger datasets:

In [ ]:
```
sample_list_2 = list(range(100000))
sample_array_2 = np.arange(100000)
```

In [ ]:
```
%%time
temp = list_func(sample_list_2, sample_list_2)
```

In [ ]:
```
%%time
temp = arr_func(sample_array_2, sample_array_2)
```

With the larger dataset, we see that using NumPy results in code that executes many times faster (might vary depending on your machine)! Throughout this course (and in the real world), you will find that writing efficient code will be important; arrays and vectorized operations are the most common way of making Python programs run quickly.

## 2. Glob-ing

Remember that the phrase "glob" means search through directories and files. The abbreviation stands for global meaning that we search globally throughout the filesystem. We want to copy certain files from `data/university` to `data/`. Note that `data/` is the parent directory of `data/university`.

**Question 2.1** Remember that `|` (pipe) takes the output of one command and feeds it as the input of the next. Describe what the follow command does:

In [ ]:
```
!ls data/ | wc -l
```

**Answer**: (Type answer here)

**Question 2.2**

We want to copy some files over to a new folder. The files we want have file names following a pattern:

1. are stored in `data/university`
2. starts with `nyu` then has 3 digits, then
3. ends with three characters preceded by period (or "has a 3-character extension")

For example `nyu547.txt` or `nyu537.pdb`, but not `nyu762uae.txt`.To be safe, we check to see what files match that criteria.

Use `ls`, with a wildcard pattern, to find the list of files that match the criteria. Next, pipe it through `wc -l` as above to count how many there are. There should be 93 files found.

```
In [ ]: # !ls <your pattern here>
```

```
In [ ]: # !ls <your pattern here> | wc -l
```

**Question 2.3**

Make a directiory `data/university2` and copy the matching files over to the new directiory. You should use the commands `mkdir` and `cp`.

```
In [ ]: # BEGIN SOLUTION

        # END SOLUTION
```

```
In [ ]: !ls data/university2
```

## 3. Bash Scripts

Instead of entering commands line by line, we can add commands to a script. With a script we can reuse code. In this problem, we want to use scripts for common tasks such as renaming files and viewing lines of files.

### ***3.1 Working with Strings***

Bash has a way to replace a portion of a string with different characters. The command resembles `${string/${old_characters}/${new_characters}}`. Here `string`, `old_characters`, and `new_characters` are variables. For example suppose we want to replace `Hello` in `Hello World` with `Hi`. We could use the following command...

In [ ]:
```
%%bash
string="Hello World"
old_characters="Hello"
new_characters="Hi"
echo ${string/${old_characters}/${new_characters}};
```

Write a script called `rename_files.sh` to rename the files in the current working directory according to `old_characters` and `new_characters`. For example, suppose we want to replace each occurence of `nyu` with `CDS` for the files in `data/university/`. We would run `bash rename_files.sh nyu CDS` in the terminal.

Try taking the following steps to write `rename_files.sh`:

1. Use `mv` to rename the files. Replace `old_characters` with `new_characters`
2. Obtain `old_characters` `new_characters` as the two arguments passed to the script. Remember we access the arguments in order as `${1}` and `${2}`.
3. Use a `for` loop that begins with `for string in $(ls ${1}*)`. Here the wildcard will indicate all files beginning with `old_characters`. Note that `$(...)` means subprocess -- run the command as a separate process to give us the output for use in the script.

In [ ]:
```
# COPY THE CONTENTS OF YOUR SCRIPT HERE
```

### *3.2 Working with Numbers*

Remember the script `middle.sh` from Lecture 06. It allowed us to view the middle portion of a file by chaining together `head` and `tail` with a pipe. Let's try to rewrite it a little...we want to write a script called `show_lines.sh` that indicates a file, the ending line, and the starting line.

For example, if we wanted to view line 5 up to and including line 10 of `data/planets.txt`, then we would run `bash show_lines.sh data/planets.txt 10 5` in the terminal from the current working directory.

Following the code in `middle.sh` we need to specify the number of lines for `head` and `tail`. If we indicate `10` for `head` through use of `${2}`, then we need to specify `6` to tail. Note that `6=10-5+1`. We can do arithmetic in bash using `$(())`. Note that we have two sets of parentheses instead of one set for subprocess (see Question 3.1). So we would need `$((${2} - ${3} + 1))` for `6`.

Using this observation write the code for `show_lines.sh`.

In [ ]:
```
# COPY THE CONTENS OF YOUR SCRIPT HERE
```