Class 5 Functions and Advanced I/O

I. Functions
II. Advanced I/O

I. Defining functions

```
def functionName(any parameters):
    what to do in function

    return something
```

A. Module

- A set of text files available to use that contains definitions of functions
- Does nothing demonstrable to us in the console when run by itself
- In order to use the module, we write code that imports modules e.g. `math` module has constants and functions → pi, e, etc…
  - A **function** always has () at end, whether or not they need the input
  - Another eg. `random.random()` → `random()` function within `random` module
  - **Constants** do not have input → no `()`

- Just to illustrate how `math` module is working, we can write our own modules and import them instead of math
  - important to have all files in the same folder
  - And make sure main file imports all modules needed → this 'main' file is called the **tester** (because it 'tests' the functions within a user-defined module)
  - In the **tester**, you can change a long module name to a short one by declaring "`import ___ as __`" when importing

e.g.

```
import circle as c

radius = 13
area = c.area(radius)
```

B. How should we define and use functions?: Either one of the two options is fine!

- One option: Writing a module(s) of useful functions and then writing a **tester** (as above)
- Second option: Writing one file with a list of functions you need and then writing a **main function** in the same file → see `pig.py` for an example
  - Remember you need to call the main function by `main()` in your code for this file to output something

C. Why do we use functions so much: encapsulation

1) Division of labor
2) Debugging
   - Easier to check where it went wrong
3) Readability
4) Design
5) Reusability

- Main function is the part that is not reusable, that's why we want it to be as small as possible

- Does not matter which order definitions of functions are written when functions call each other
- **encapsulation:** Fancy term for packing the details into different modules or functions
    - In the main function, you just know a certain part is being done by another function without knowing how; Details are in the other function or module

II. Advanced I/O

A. Input

```
variable = open('text_name','r') ## to open file object
var2 = variable.read() # store entire text of file in var2
var3 = variable.readline() #store next line in var3
```

e.g.

```
in_file = open('example.txt','r')
s=in_file.read()
print(type(in_file))
print(type(s))
```

→ Output: file
str

e.g. of using `in` operator and `for` loop on files to iterate through lines

```
for line in in_file:
    print(line)
```

B. Output

```
var = open('nameOfNewFile.txt','w')
var.write('stuff to write')
var.close() # file only begins to be written when you close it
            # before that, only stored in memory
```

for eg.

```
fred = open('out_file.txt', 'w')
fred.write('very cool')
fred.write('write more stuff')
fred.close()
in_file.close() # also close the read file to conserve memory
```

for e.g. to copy the entire text into another file

```
## this program copies the file example-txt

file_in = open ('example.txt', 'r')
file_out = open ('copy.txt', 'w')

#do the copying
for line in file_in:
    file_out.write(line) #line already has the newline character
```

```
file_in.close()
file_out.close()
```

Things to note:

- If a file with the same already exists in directory, newly written file overrides it
- Always remember to close the file after writing AND reading! Program doesn't print in the text file until you close it
- Consecutive write methods will lead to strings printed continuously on one line
- You need to add `\n` at the end of each line when using the `write` method to change lines
- '`a`' stands for append, or writing at the end of an existing file instead of replacing it as we did with '`w`'

C. Deleting and renaming files (os module)

```
import os
os.remove(textfileName)
os.rename(originalFile, newFileName)
```