Week 4 - Factoring & Debugging

What is programming?

- Problem solving
  - Using a specific toolkit (computer code)
  - Combined with logic

We write a series of logical steps that can be taken (given assumed inputs) in order to realize a desired outcome

How do we solve a problem?

- In programming, we utilize a method called **Functional Decomposition**
  - Also called **Factoring**
  - Break a problem down (decompose it) into its smallest functional elements
  - Construct those elements
  - Combine elements to achieve the end goal

Factoring Recent Assignments

1. `StudentRecord` and `Course` classes
2. Recursive Functions

Let's walk through factoring these problems

NOTE: DON'T PROCEEED WITH THIS VIDEO UNLESS YOU HAVE COMPLETED ASSIGNMENT 3

Advantages of Factoring

- Your code will be easier to read
- You will know what you need to do
- It is clear what the next step is
- Your code will be **reusable** to a greater extent
    - Other programmers will have an easier time following your code
- It will be easier to **debug** and run **unit tests**

What is Debugging?

Debugging is, like the name suggests, the process of removing bugs from a program or script.

- Why do we get the error that we get?
- How is data moving through our code?
- What needs to be fixed?

Note: Anecdotally, "debugging" has its origins in the physical removal of bugs from giant vaccuum-tube computers in the early/mid 20th century, and is attributed to Grace Hopper

What is Unit Testing?

Unit Testing is the process of feeding many different (and possibly wrong) types of information to our code in order to determine how the code will work under less-than-ideal circumstances.

- What happens if our input is incorrectly formatted?
- What if the data is the wrong **type**?
- What if ...

Why Should I Debug and Unit Test?

- **Debugging** is critical, since our code will not work if it contains bugs. At the very least, it will not work as we want/expect it to
- **Unit Testing** is how we understand where our code fails to prepare for any possible case that could occur
    - We need this if we want to prevent "Garbage In, Garbage Out" problems in the future

Debugging in IPython

From inside an IPython console, we can run our script using

```
%run my_script_name.py
```

This will run the script inside our console, and all variables will then be available to us for exploration afterwards

Debugging in IPython

We can also run our script as

```
%run -d -bX my_script_name.py
```

`X` should be an integer. This will open the debugger at line `X` in our script after running all previous lines

Debugging in IPython

Use `%debug` to enter debug mode in IPython after an error

- Allows you to explore around the error!

Debugging in IPython

Use `_ih` to access a list of recently run commands

```
_ih[-5:] # Access the last 5 commands that have been run
```

This is useful to make sure your code was run in order and that you are processing data correctly

Debugging in Scripts

You can also set debugging traces in your code:

```
from IPython.core.debugger import set_trace

# Chunk of your code goes here
set_trace()
# Rest of your code here
```

The trace will kick your program to a debugger at the point in which the trace is inserted into your code

Doing Unit Tests

```
import unittest

class TestComNum(unittest.TestCase):

    def test_ne(self):
        self.assertNotEqual(ComplexNumber(4,3), ComplexNumber(4,-3))


unittest.main(argv=[''], verbosity=2, exit=False)
```

Let's look at the unittests used for grading a past assignment to learn more.

Using Try, Except

```
try:
  myCode()
except:
  raise RuntimeError("This is what went wrong...")
  # We could also use any other kind of error
  # TypeError, KeyError, etc.
```

For more types of errors, see this list

This kind of code block allows us to create code that **might actually fail**, but that we want to run wherever possible, while being notified when it does not succeed.

Lab Time!