Section B1 Debugging

How do run the Python Debugger?

The Python Debugger (pdb) is the standard debugger for Python. There are two ways you can have your Python code run through this debugger.

1. Incorporate the debugger directly into your code through a module and call the debugger's **set_trace** To do this, simply add the lines **import pdb** and **pdb.set_trace()** to your code. Any code above the line pdb.set_trace() will run as normal, but all code after it will run through the debugger.
2. You can call the debugger as a script through the terminal. But you can also use it inside any Shell scripts you have written. Instead of making changes inside your code, you will run your Python program using **python3 -m pdb [file name]**.

Using commands to aid your debugging

The first set of commands allow you manage the program's execution.

- **s**(tep) – Execute the current line. Stops at the first possible occasion (either the next line or a function call).
- **n**(ext) – Execute the current line. Stops only at the next line.
- **r**(eturn) – Execute the code until the current function returns.
- **c**(ont(inue)) – Execute the code until a breakpoint is hit.
- **b**(reak) – Set a breakpoint. A breakpoint is a line of code that has been marked by the debugger to stop at.

The next few commands allow you to gain information about where your code is and what it is doing.

- **l**(ist) – Displays 11 lines around the current line. If you enter this command again, without continuing code execution, it will continue the previous listing.
- **p** – Runs Python's print. You can use p [variable name] to easily check the current state of the variables in your program. Because this is running Python's print function, print is an invalid command, and if you want to write it out in full, you will have to format it as proper Python code. There's also **pp** to display using pprint instead.
- **w**(here) – Displays the Stack Trace. This is not normally necessary, as proceeding to a new line will automatically print the Stack Trace, but it is useful if you have run a few commands and want to remind yourself where the program currently is.

Finally, here are some other useful commands.

- **h**(elp) – Prints the list of commands. If you use this command followed by the name for a second command, it will give you more information on using that second command.
- **q**(uit) – Quit the debugger.

IPython Debugger
The IPython Debugger (ipdb) is an improved debugger for Python that uses the same interface as pdb.To run ipdb inside Jupyter Notebook you will need to add the ipdb module by adding the line **from IPython.core.debugger import set_trace** the call the **set_trace()** function where you want the first breakpoint.

exceptions

Working with exceptions in Python: The *try* statement
A try statement is comprised of at least two components. Firstly, the **try** block. When the program encounters a try, it will *attempt* to run the code inside the block. If the program encounters an exception, the program will stop running code inside the try block and will begin executing the **except** block. If the program did not encounter an exception, the program will skip the except block after the try block has completed.

There are two other types of blocks that can be used with a try statement. The **else** block will run if there are no errors, that is, it runs if the except block does not run. The **finally** block executes regardless of the result of the try.

```
try:
    usr_input = input("Enter an integer: ")
    inverse = 1/int(usr_input)
    print(inverse)
except:
    print("An error occurred.")
else:
    print("No error occurred.")
finally:
    print("block completed.")
```

Raising your own exceptions
When you are developing more complex code, you might want to define your own exceptions so that any programmer using your functions or methods can get more relevant information. In Python you can do this using the **raise** keyword

```
a = 0

if a < 1 or a > 10:
    raise Exception("variable a must be between 1 and 10")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
/tmp/ipykernel_2553/1300123875.py in
      2
      3 if a < 1 or a > 10:
----> 4     raise Exception("variable a must be between 1 and 10")

Exception: variable a must be between 1 and 10
```

You can also replace the Exception with the specific exception types, such as TypeError to further customise the exception.

Linting

Linting is a subset of static analysis that specifically looks for stylistic issues and mistakes in code. Some can also flag possible memory leaks and insecure code design. Stylistic linters such as **Pylint** and **Pycodestyle** (formally named pep8) can both be used to ensure good programming practices are being followed.
The code and description are a pair, with the description acting as a brief overview of the error and the code being usable to check the documentation for more information. The code will begin with one of 3 letters.

- **C** – your line does not meet the standard coding conventions
- **W** – your line has a potential problem that requires your attention
- **E** – your line has a severe problem that requires your attention.

Pylint also rates your code out of 10.

Testing

Testing using assertion

```
def sum(num1, num2):
    assert (type(num1) == int or type(num1) == float or type(num1) == complex), \
        "num1 was not a number (int, float or complex)"
    assert (type(num2) == int or type(num2) == float or type(num2) == complex), \
        "num2 was not a number (int, float or complex)"
    sum = num1 + num2
    return sum

x = sum(12, 24)
print('The sum of your two numbers is', x)
```

Unit testing

With unit testing, we test each individual component, called a **unit**, independently of the other units. What defines a unit depends on context of what you are working on. Typically a unit is an individual algorithm, function, or method, but it could be any block of code.

Black box Testing
In computer science, a **block box** a system that is only perceived through its input and output. That is you have no knowledge of how it works internally. Here are some examples of black-box testing against the aforementioned abs() function. Here it makes sense to use assert for the first three tests, as we expect these to be true. The latter, we do not know if it will work or not, so you might consider using a try except.

```
assert abs(16) == 16
assert abs(-4) == 4
assert abs(3j) == 3
try:
    abs('hello')
    print('test: abs(\'hello\') passed')
except:
    print('test: abs(\'hello\') failed')
```

Unit testing
There are three broad categories to test when unit testing.

1. **Normal usage**: These tests are designed to test if the program performs as expected under normal usage.
2. **Edge cases**: These are the extremes or boundaries that your code should operate under. They may also be special values that you need to manually handle. If your code works at all edges, we can assume it works on all the cases between them. With practice you will become more comfortable with identifying edge cases.
    - When two or more edge cases are tested in a single test, it is called a **corner case**.
3. **Unintended usage**: These tests are designed to try and break your code. For example, if you had some code or a function that attempted to sum two numbers together, does it behave in the way you intend if you pass it two strings?

Test Driven Development
Test driven development (TDD) This involves you building all the tests that your eventual code will need to pass, before any of your code's functionality is implemented.