THE UNIVERSITY OF SYDNEY

INFO1110 / COMP9001 Week 7 Tutorial

Testing and exceptions

Testing and coverage

We will be focusing on different types of testing and how to write testing programs and frameworks.

- Unit Testing

Unit testing involves testing components of your software and examining their functionality. Checks individual functionality of functions and objects and confirms that the method is correct.

- Black Box (input/output) Testing

Black box testing involves not knowing the internals of the software and testing it. This may reduce how you can test the system but can help understand problems from a user's perspective. A simple program to compare the out of a program to the expected output.

- White box testing

White box testing involves using the structures within the software itself and testing them. These can help expose and narrow down issues related to the software and expose where the problem lies.

- Regression Testing

A regression test involves re-running the already established test cases after a problem is fixed. This is to ensure that as part of your fix you have not introduced a new probelm in your system.

- Code Coverage

Code coverage is used in conjunction with test cases. It examines how much of the code is executed from the test cases and provides a report. The report provides insight into which components are tested and what still requires testing.

Input and output

Input/Output test cases are very easy to produce and are reflective of what the user will see when executing the program. Initially when writing test cases for your program you will identify behaviour about the program and be able to see what the expected output should be.

A great unix utility is to use the `diff` command and pipe the output to a file and compare it to the expected results.

Example:

```
python my_program.py > actual.out
diff actual.out expected.out
```

or

Example:

```
python my_program.py | diff - expected.out
```

Question 1: Debugging a vending machine

You will be given a vending machine program to test. You can access this program from:

`git clone https://github.com/info1110/tutorial7.git` or from the Ed resources section.

- Loads a file from command line argu which contains stock in the format:

  <beverage> <cost> <quantity>

- If the format is incorrect (qty is missing for example) Our program should show an error message `Unable to read stock file`

- Allow the user to select an option from [0 - n-1] in the vending machine menu The vending machine menu should be displayed in the form: [number] <beverage>

  The screen should also ask the user if they want to quit by pressing q, followed by enter to quit.

  Example:

  ```
  [0] espresso
  [1] latte
  [2] cappuccino
  [3] ice_coffee
  Select an option between 0 and 3 or press 'q' to quit:
  ```

- Handle checking out the item and for the user to pay for it. This should ask for the user to insert $1, $2, $5, $10 or $20 until the amount of change is greater than or equal to the cost of the beverage.

If the user cancelled the item then they are able to press c, followed by enter to cancel their order. The program should then output `Your order has been cancelled` and then close. Once paid, the program will state `Item has been paid for, Thank you!` and then close.

Example:

```
[0] espresso
[1] latte
[2] cappuccino
[3] ice_coffee
Select an option between 0 and 3 or press 'q' to quit:
Insert $1, $2, $5, $10 or $20 or press 'c' to cancel:
```

- The program should not allow the user to pay unless the stock is greater than 0. If a user has selected an item with no stock, then the program should output `No more items left` or otherwise proceed to allow the user to pay for the item.

- Why do we test code in the first place?
- Would we create test cases first or after have implemented the code?

We recommend you write these unit test files (you may also use Ed to -test- your test cases)

- `display_menu_and_stock_1.out`, Go through the case where system displays the menu selection and user quits. It will use the example.stock file
- `display_menu_and_stock_2.out`, Go through the case where system displays the menu selection and user quits. It will use the ultimo.stock file
- `checkout_and_pay_1.out`, Go through the case where the user selects item 0, is charged and pays for it with $1 and $2.
- `checkout_and_pay_2.out`, Go through the case where the user selects item 2, is charged and pays for it with $1 and $2 and $5.
- `checkout_and_pay_cancel_1.out`, Go through the case where user selects item 1 and cancels

- What other test cases could you add to this program?

Unit Testing

Unit testing is where we test individuals components of the source code. We will be going through how to write our own unit testing pattern.

When writing your own unit tests you will have an expected result and actual result. If the actual result does not meet the expected result then you can conclude there is a problem with your code. This helps catch errors that you have made within your program. To write a unit test you will need to use the `assert` keyword that will evaluate a condition.

Example:

```
assert 2+2 == 4
```

This will evaluate correctly and no error will be thrown by the python interpreter. However, in the following example, we can clearly see that the condition is not true.

```
assert 2+1 == 4
```

And will throw an `AssertionError`.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

When writing your own test cases you will, it is best to create a separate python file that will utilise your application program.

The convention is to specify `test_` at the start of your test function to communicate clearly to other programmers that it is a test function.

```
def test_add():
  return 2 == simplecalculator.add(1, 1)
```

Once you have written all your unit test functions you can execute them in your script. Note: The following tests relate to a calculator program.

```
test_add()
test_sub()
test_mul()
test_div()
```

Any error that you encounter will invoke an AssertionError and you will be able to retrieve the line number it occurred on from the traceback statement.

Alternatively, you may add your test cases to a list and execute them one by one in a loop. However, this is a slightly tricky pattern to setup.

```
test_fns = []
test_fns.append(test_add)
```

Assuming that none of the functions take any arguments you can execute them in a loop and record their results

```
passed = 0
for fn in test_fns:
  try:
    fn()
    passed += 1
    print(fn.__name__ + ' passed')
  except AssertionError as e:
    print(fn.__name__ + ' failed')
    print(e)
  except Exception as e:
    print(fn.__name__ + ' error')
    print(e)

print(passed + '/' + str(len(test_fns)) + ' passed')
```

- What is the difference between black box and white box testing?

Question 2: Simple calculator

You are to write a simple calculator program that will implement basic arithmetic functions. You can retrieve the scaffold from: `git clone https://github.com/info1110/tutorial7.git`.

Each of these arithmetic functions should be localised to its own method so that you can easily test your code in isolation.

Operations to implement:

- add -> 2 variables
- sub -> 2 variables
- mul -> 2 variables
- div -> 2 variables
- lsh -> 2 variables
- rsh -> 2 variables
- mod -> 2 variables
- abs -> 1 variable

Add Usage:

```
add 1 2
3
```

```
def add(a, b):
    return a + b
```

Mul Usage:

```
mul 10 5
50
```

Once you have constructed your code you can move onto constructing unit test for your program.

Question 3: Unit Tests for calculator

After your have implemented your calculator program, now implement some unit test cases by writing your file that tests your code. It is advised you create your own test file called `simplecalculator_test.py`. To get access to the program you have written from the previous exercise, you will need to use the `import` keyword.

```
import simplecalculator
```

You can refer to this module through its name and execute the functions attached to it.

```
def test_add():
        assert simplecalculator.add(1, 1) == 2
```

You should test all of your functions as well as any except case that you can think of. Hint: What id you were to execute `sub` with 1 0 as input?

Exception handling

Handling

Unfortunately we cannot make every bit of input correct and other processes may interfere with files we are using and logic in our code may throw an exception because it is in an invalid state. These are what are considered to be exceptions, states in our program that will need to handled and we can do this by using a `try-except` block.

```
x = int(input())
d = int(input())
y = x/d
```

- What would happen if we were to input 0 for d?

You can consider `try-except` blocks in a similar way as conditional statements. The different being is that the kernel or interpreter has signaled your program that it has made an exception will need to jump to the except branch of the program.

```
x = int(input())
d = int(input())

try:
  y = x/d
except:
  print("You attempted to divide by 0!")
```

- What other errors could we encounter besides providing 0 for input?

In the event we want to handle the case where we multiple exceptions we can specify the branch of the exception. Similar to if statements.

```
try:
  x = int(input())
  d = int(input())
  y = x/d
except ZeroDivisionError:
  print("You attempted to divide by 0!")
except ValueError:
  print('Input could not be converted to int')
```

Raising your own Exceptions

Although you have encountered errors and exceptions from the interpreter, we can create our exceptions when we have determined an error in our own system. Python does not force the programmer to handle exceptions unlike other languages, however this more to the detriment to itself since it can hide that an error can occur.

To raise our own exceptions we can use the `Exception` type and the `raise` keyword. The `Exception` type allows the user to specify a message an argument when the exception is raised.

Raising an exception:

```
raise Exception("This is an exception buddy!")
```

However, we can define our own exceptions by creating a class (a little ahead of schedule).

```
class MyOwnException(Exception):
  pass

#We can now raise this exception

raise MyOwnException("Oh hey! It's my own exception!")
```

Question 4: Video Rental Store

Source: `git clone https://github.com/info1110/tutorial7.git` or from the Ed resources section.

You have been given code to a video rental store program. This program does not raise exceptions when there is:

- No DVD available (`NoDVDError`)

  The DVD will be in the list but if the user attempts to rent the DVD and it is not available it will not be available and therefore this exception should be generated.

- The user has made an invalid selection (`InvalidSelectionError`)

  With the list of DVD available, it will provide an index of the DVDs and if a user specifies an index outside of range displayed it should raise an exception "InvalidSelectionError"

To ensure you have made these changes, you can use the test script `rent_test.py` provided with the source code.

Question 5: New feature tests and regressions

With the Video Rental Store code, you will have access to unit test cases. After changing the code from the previous question we want you to write new unit tests for that code that shows that the exception is being executed correctly.

You can access the unit test file for this program via `rental_store_tests.py` file.

```
def test_no_dvd_error():
        pass

def test_invalid_selection_error():
        pass
```

After implementing these test cases, you can run the code and check to see if not only that features you have implemented are working but other test cases are also working.

- Why do we have regression testing?
- Is regression testing just an extension on unit testing?

PDB - Python Debugger

PDB models itself after many other debuggers such as GDB. PDB allows the programmer to set breakpoints which allows you to halt the execution of hte program at a certain point and inspect variables and their values that have been set up to that point of execution.

To run pdb, you can import the module directly either using the interactive mode and import your module or run it separately as a module.

```
python3 -m pdb <your program>.py
```

In this case, pdb will immediately invoke its trace and allow you to inspect your program

Execute until return

By default by typing `r` into the pdb console, pdb will execute the script until a return statement, in a simple script, this is the end of the program.

```
r
```

Breakpoint

As discussed before a breakpoint allows pdb to halt execution of the program for the programmer to interact with the script, after this point is reached the programmer can use commands such as `display` to inspect variables that have been set to determine if they have been calculated correctly.

You can use the shorthand `b` to set a breakpoint or `break`, which is usually followed by a line number Example:

```
b 4
```

This will set a breakpoint on line 4, if no statement exist the command will return an error.

Display variable contents

Once the program has reached the end or is at a breakpoint you can display the value of a variable. If it does not exist (yet) then nothing will be displayed.

- What value does PDB have over just printing variables and state information?

```
display <variable name>
```

The python foundation offers very good documentation on how to use pdb: PDB

Question 6: Writing your own unit testing framework (Extension)

When we import a module we will have access to a property called `__dict__` which is a dictionary of all objects (including functions and variables) which are accessible within the module.

Using this property, you use it to iterate and find the methods which are used for testing. You can make the assumption that any method that starts with `test_` is a unittest.

Hint: Use `callable`

Resources

Errors and Exceptions

Python Debugger