CSC 108H: Introduction to Computer Programming

Summer 2012

Marek Janicki

Administration

- Midterms are being graded this Saturday and Monday.
  - If they will be finished by Monday office hours, this will be posted on Piazza, otherwise they will be returned in next week's lecture.
- The next Exercise will be posted this Saturday.
- Help Centre is in BA2270 2-4 M-R.

Review: None

- We have seen lots of types.
  - bool, str, int, etc.
- Each of these types can take on several possible values:
  - True, False
  - any integer
  - any str
- We have also seen `NoneType`
  - This type has one element: `None`

Review: None

- None is meant to represent no type.
  - So expressions that don't evaluate to any type evaluate to None.
  - So function calls and methods that don't return anything evaluate to None.

Which of these evaluate to None or NoneType?

- `def foo():`
  `    print 3`
- `foo()`
- `[1,3,5]`
- `[1,2,5].find(2)`
- `[1,5,3].sort()`
- `type(print 4)`
- `type([].extend([3,4]))`

Which of these evaluate to None or NoneType?

- `def foo():`
  `    print 3`
- `foo()`
- `[1,3,5]`
- `[1,2,5].find(2)`
- `[1,5,3].sort()`
- `type(print 4)`
- `type([].extend([3,4]))`

Review: Modules

- Modules are a way to group related bits of code.
- Each .py file is its own module.
  - The module name is the file name without the file extention.
- We use `import module_name` to load a module.
- And `module_name.module_element` to use any functions or variables defined in the module.

Which of these are potentially legal syntax?

- import foo
- from foo import goo
- from foo import go+
- import go+
- foo.__name__
- goo.__name__
- go+.goo()

Which of these are potentially legal syntax?

- import foo
- from foo import goo
- from foo import go+
- import go+
- foo.__name__
- goo.__name__
- go+.goo()

Code Correctness

- Given that we've written some code how do we know that it does what we want?
- Option 1: Prove it is correct.
  - Difficult and time consuming.
  - Needs a lot of background.
  - Often seems much stronger than required.
- Option 2:
  - Argue that the code works on a 'representative' set of inputs, therefore is correct.

Option 2

- This is doing Testing.
- What do we want from our Tests?
  - Easy to reuse.
    - So that if we rewrite our code, we don't need to rewrite the tests.
  - The should be separate code from the code that is being tested.
    - Again, so that we don't need to rewrite the tests.
  - The should cover enough inputs that they convince us that the code works.

Testing Reuse

- Ideally, we have tests that we don't need to change when we change the function that we are testing.
- The only time this should change is is we decide that we need to change what a function does.
  - That is, we've decided our overall program structure is a dead end, and we need to change big parts of it.
    - This is call refactoring, and it is undesirable.

Testing Reuse

- One way to ensure that tests can be reused is to keep them in a separate file, and have them only rely on the docstring of the things we test.
- This means that as long as each function does the same thing, we can keep running the same tests.

Test Selection

- What do we test?
  - For a given function, what inputs do we test?
  - Which functions do we test?

Testing Inputs

- What do we test?
- We can't test all inputs.
- So we need to choose a subset that is representative.
  - We can have 'typical inputs'.
  - We can test things where we might suspect programmer error.
  - We can test 'boundary conditions' that we suspect might have been overlooked.

Testing Inputs

- 'Typical Inputs'
  - Think about why you are writing a function, and how you think it will be used.
  - Then take some canonical examples of this.
  - Often times here it is useful to test things on several randomised inputs.
    - This will be covered after covering classes.

Testing Inputs

- It is useful to think 'adversarially' when picking test cases.
  - That is, try to picture yourself as an adversary trying to break a program.
  - But do so without cheating, so if the docstring specifies some kind of input, limit yourself to those inputs.
  - But within those inputs try and choose as bad inputs as you can.
  - These type of inputs are often called corner cases.

Break, the first.

Think of Test Cases for the following function stub.

```
def min_max(L):
    '''(list of ints) -> (int, int)
    Return a tuple of the minimum and
    maximum values in L in that order.
    Return None if L is empty.'''
```

Writing Tests

- So given that we know what our test cases are, how do we actually write tests to test them.
- Python has two built-in modules to help with testing.
  - nose
  - unittest
- We use nose in this course.
  - unittest is class-based.

Testing with Nose

- The context for testing with `nose` is that we have a module named `mod`.
- We want to test some or all of the functions in it.
- To do this we create a module called `test_mod`.
- In this module we `import nose` and we `import mod`.
- For each function `func` and behaviour we want to test, we have a `test_func_behav()` function.

Testing with Nose.

- So it's usually useful to keep 'typical' input tests in separate functions from 'adversarial' test.
- We have:

```
if __name__ == '__main__':
    nose.runmodule()
```

- This runs every test function in our test module.
- The output of this is tells us whether each test succeeded, failed, or generated an Error.

Test Functions.

- What do we put in the body of a test function?
  - Going in we know what test cases we want to test, and we know the outputs we expect.
  - We want to test if the actual output of the function is the same as what we want it to be.
- In the body of `test_func()` we have assert statements.
  - `assert (boolean condition)` will do nothing if the condition is true, but will throw an error if it's false.
  - So `test_func()` has a bunch of statements like:
  - assert func(input) == (expected_output)

Nose Output

- The first line of output tells us the result of the tests.
  - a dot means pass, an F means fail, an E means an error.
  - So, a failure is incorrect output, an error is an exception of some kind.
  - Each failure or error produces information about that failure or error.
  - The last bit tells us the number of tests passes, the number of tests failed, and the number of errors.

Nose Output

- The information about the errors so far is just the error information that python gives back to us.
- If we fail a test we can an 'AssertionError'.
- If we want to add some information to this, we can put in a string after a comma in the assert statement.

```
assert (condition), "Some String."
```

Designing Nose Test Files

- It is useful to test every function you write seperately.
  - Called unittesting.
  - Nose makes this easy.
- Writing one big test for a function that calls other functions is a terrible idea.
  - It makes tracking down bugs really hard.
- If you change the implementation of a function, the nose test file doesn't need to be changed.
  - Regression testing is the idea of testing different versions of software to ensure no new bugs exist.

Break, the second.

Writing Code

- In the first lecture, we talked about the design, code, verify paradigm.
- Design a program to solve the problem.
- Code the design you have settled on.
- Verify that the code satisfies the design.

Design, Code, Verify and Testing

- This does not mean that you shouldn't think about testing until the end.
- Verify is essentially running test cases, but testing is part of all three steps.

Design

- The first step is to design your code. This means designing the modules, figuring out helper functions and things like that.
- It can be very tempting to go to code as soon as you have a design that seems plausible.
  - But a bad design means massive hours of code investment.
- Already at the design phase you should be trying to think adversarially about your design.
  - Try and break it.

Design

- Designing Code isn't all whiteboard/scrap paper stuff.
- A good step when writing out a big project is to create function stubs for all the things you will write.
  - A function stub is a function definition and docstring with no body (i.e., the body is pass).
- As soon as you have a docstring, you can write test cases for the function.

Transitioning from Design to Code.

- As soon as you have a docstring, you can write test cases for the function.
- Test cases should be the first real code you write.
  - This because we want them to be implementation independent.
- Once you've written your test cases, you should start writing the actual code.

Writing Code

- Try to write code 'bottom up'.
  - That is, start with functions that don't call other functions that you've written.
- As soon as you've finished writing a function, test it immediately.
  - The easiest time to fix/find errors is when the function is fresh in your mind.

Writing Code

- If you're writing, that doesn't mean you don't need to worry about design anymore.
  - You may find that something is difficult, and that a helper function you haven't designed would in fact be really useful.
  - When this happens, you should take a step back, ensure that your overall design is still good, and then think about and write test cases for this helper function.

Verify

- When you're done writing everything, and everything passes the tests, it's time to step back and ensure that you have good test coverage.
- Now that you've written the code, you should have a better intuition for your design, and should be better able to think or corner cases that can break it.

Design Code Verify

- This is a useful overall strategy.
- But it is also useful as a sub-strategy for every function that you write.
- In some sense at every meta-level of programming you should be trying to implement this.

Testing Summary

- Want individual Unit tests.
  - These should be independent of eachother.
  - There should be some generic ones, and some chosen 'adversarially'.
- Want to design tests before writing code.
  - Makes for more robust code and more robust tests.
- Want to rerun tests when we change code.
- How does Nose do this?

Nose and Testing

- Unit Tests.
  - Each test in nose is its own function, so we can write a function for each unit test we want.
- Designing Tests Early.
  - All we need to write test in nose is the docstrings for the function.
  - The tests treat functions as a black box.
- Regression Testing.
  - Nose makes it quite easy to run all the tests we have whenever we want.

So you have an Error,

- If you find an error, you need to debug it, a process that is often painful.
- There are a few ways to mitigate this pain.
  - Test early! Test Often.
  - Read the error information, and use it to see if the code is correct at the point of the error.
  - Backtrack to the first point that the code differs from what you think it would be.
  - Run through the code in your head to make sure that if everything goes the way you think, the code will work.

Dictionary Review

- Unsorted sets of `(key, value)` pairs.
- Keys in a dictionary are unique.
- Values can be accessed with `dict_name[key]`
- `{}` is an empty dictionary.
- `dict_name[key] = value` adds a `(key, value)` pair to the dictionary.
  - If the key already exists, the value associated with it is overwritten.

What do these expressions evaluate to?

```
a = {}
a[0] = '0'
a[0]

a['0'] = 0
a[0]
```

```
a[0] = 0
a[0]


a[[0]] = [0]
a[[0]]
```

What do these expressions evaluate to?

```
a = {}
a[0] = '0'
a[0]
'0'
a['0'] = 0
a[0]
'0'
```

```
a[0] = 0
a[0]
0
a[[0]] = [0]
a[[0]]
crashes
```

Dictionary Review

- `key in dict_name` is `True` iff there is a `value` associated with that `key` in `dict_name`.
- `dict_name.keys()` returns a list of keys.
- `dict_name.values()` returns a list of values
- `dict_name.pop(key)` removes a key value pair from the dictionary.
- `dict_name.copy()` generates a copy of the dictionary.
- `d1.update(d2)` adds all the key value pairs in `d2` to `d1`.

Complete the function according to its docstring.

```
def minus(d1, d2):
    '''(dict, dict) -> NoneType
    Remove every key in d1 that is
    also in d2 from d1'''
```

Complete the function according to its docstring.

```
def minus(d1, d2):
    '''(dict, dict) -> NoneType
    Remove every key in d1 that is
    also in d2 from d1'''

    for key in d2.keys():
        if key in d1:
            d1.pop(key)
```