CSC 108H: Introduction to Computer Programming

Summer 2012

Marek Janicki

Administration

- Assignment 3 is up.
  - Has two deadlines.
    - Wed. Aug 8, Fri. Aug 10
  - Will talk about it at the end of class.
- Final is Thurs. Aug 16, 7-10 in SF 3201
  - Material will be covered next week.
- Office hours next week will be
  - T2-4,F4-6.
- Exercise 4 will be optional.
  - No time to release it that's not concurrent with the assignment.

Class Review

- Classes are user-made types.
  - An instance of a class is called an object.
- A class has instance variables.
  - These can have distinct values for each object of the same class.
- A class also has class methods.
  - These work the same way as other type methods.

Create an instance of MyClass, and assign 10 to an instance variable num

```
class MyClass(object):
    pass
```

Create an instance of MyClass, and assign 10 to an instance variable num

```
class MyClass(object):
    pass
x = MyClass()
x.num = 10
```

Class Review

- Object Oriented Programming supports
  - Inheritance
  - Polymorphism
  - Encapsulation

Class Naming Conventions

- Classes are named using CamelCase
  - Not pothole_case.
- Objects are named using pothole_case.
- Class methods are named using pothole_case.
- Class variables are named using pothole_case.

Classes variables vs. Instance variables

- Each class can have class variables.
  - This is a variable that is associated with the class, rather than any specific object.
  - To create them, you use an assignment statement as follows:
  - `ClassName.variable_name = value`
- The variable can be evaluated with
  - `ClassName.variable_name`
  - `x.variable_name` if `x` is an instance of `ClassName`.

Class variables vs. Instance variables

- If you change the value of a class variable using ClassName.variable_name, the value changes for the ClassName objects.
  - `ClassName.variable_name = new_value`
- If you change the value of a class variable using x.variable_name, then it becomes an instance variable for that particular object.
  - `x.variable_name = new_value`
  - The value of ClassName.variable_name does not change, nor does the value of y.variable_name for any other ClassName instance y.

Class variables

- Class variables are generally used to denote constants.
  - Altering them via objects leads to complicated code.
  - Essentially this results in a higher level of aliasing problems.

What do these statments evaluate to?

```
class MyClass():
    z = 0
y = MyClass()
z = MyClass()
z.z

MyClass.z = 10
z.z
```

```
y.z

y.z = 5
y.z

MyClass.z

MyClass.z =3
y.z
```

What do these statements evaluate to?

```
class MyClass():
    z = 0
y = MyClass()
z = MyClass()
z.z
0
MyClass.z = 10
z.z
10
y.z
10
y.z = 5
y.z
5
MyClass.z
10
MyClass.z =3
y.z
5
```

Inheritance

- `ClassA` can inherit the methods and variables of `ClassB` by defining `ClassB` as follows:
  - `class ClassB(ClassA):`
- We call `ClassA` the superclass and `ClassB` the subclass.
  - Every instance of ClassB is also an instance of ClassA.
  - Not every instance of ClassA is an instance of ClassB.
  - So the set of instances of ClassA is a superset of the instances of ClassB.

Inheritance

- We saw that if we have the same method name in a subclass as in a superclass, and we call subclass_instance.method(), then we superclass' method is overwritten and we evaluate the subclass' method.
  - But sometimes we want to mostly reuse the superclass method code, and only modify it a little.
  - This comes up particularly commonly in constructors, where if your subclass is only a small change, you would not like to copy and paste the code from the constructor of the superclass.

Inheritance

- It would be really useful if we could call a superclass method inside of a subclass.
- Two ways of doing this, if x is an instance of SubClass.
- `SuperClass.method_name(x, ...)`
  - `x` goes in place of `self`.
  - No longer works in python 3.
- `super(SubClass, x).method_name(...)`
  - `super` returns `x`'s superclass object.
  - `self` implicitly passed here.

Inheritance

- Inheritance allows us to define new methods, and overwrite already existing ones.
- But even when we overwrite existing ones, we can still access them using super.
- `super(SubClass, x)` will return the SuperClass object associated with x.
  - Requires x to be an instance of SubClass.
- Recall that if x is an instance of a SubClass, it is also an instance of the SuperClass.

What do these evaluate to?

```
class ClassA(object):
    def foo():
        return 4
class ClassB(ClassA):
    pass
class ClassC(ClassB):
    def foo():
        return 5
```

```
x = ClassB()
y = ClassC()
x.foo()

super(ClassB, x).foo()

y.foo()

super(ClassC, y).foo()
```

What do these evaluate to?

```
class ClassA(object):
    def foo():
        return 4
class ClassB(ClassA):
    pass
class ClassC(ClassB):
    def foo():
        return 5
```

```
x = ClassB()
y = ClassC()
x.foo()
4
super(ClassB, x).foo()
4
y.foo()
5
super(ClassC, y).foo()
4
```

Break1

Exceptions

- Python often generates errors.
  - We can make our own functions, modules, types.
- We can also make our own errors, and generate our own errors.
- Errors in Python are objects.
  - All error are subclasses of `Exception`.
  - This means we can define our own errors by creating subclasses of `Exception`.

MyError

- `class MyError(Exception):`

  `pass`
- We can create instances of `MyError` by using `MyError()`.
- But these don't stop the code in the same way that python errors do.
- We can also create instances of python errors.
  - `TypeError(), NameError()`, etc.
  - Creating them in this way also doesn't stop the code.

Causing Code to crash

- Done using the keyword `raise`
- `raise TypeError()` will cause the code to crash with a `TypeError`.
- `raise MyError()` will cause the code to crash with a `MyError`.
- Passing the constructor a string will cause it to crash with that error massage.

Why do we want code to crash?

- It can be one way of enforcing sanity checks.
  - For example if you know that some list needs 10 elements, you can check the length and crash if the length is wrong.
  - Sometimes the program might run a very long time before an early error actually breaks the program.
    - The longer it runs, the harder the error is to source.
- Mostly crashing is undesirable.

Avoiding Crashes.

- Avoiding crashes in python involves two keywords:

```
try:
    block1
except:
    block2
```

- Block1 is executed until an exception is raised. Then block2 is executed.
- If no exection is raised, block2 is not executed.

Not catching some exceptions

- Often you only want to catch some exceptions.
  - It's common to design code to produce a specific kind of exception.
    - It's a common way to enforce parameter requirements.
  - But code may also have unplanned errors.
    - It is desirable for the code to crash in this case to indicate that something is wrong.
- `except SpecificException:`
  - This only catches instances of SpecificException or its subclasses.

Getting information from Exceptions

- As exceptions are objects, it is often useful to give them instance variables.
  - In particular, the things that actually went wrong should be added to the exception.
- For this to be useful, we need to be able to access the exception that was raised.
- `except MyError e:`
  - This creates a local variable e that refers to the instance of MyError that was raised.
  - This local variable can then be used in the exception block.

Break2

Assignment 3