CIS192 Python Programming

Object-Oriented Programming

Robert Rand

University of Pennsylvania

February 10, 2016

Outline

1. Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods
2. Decorators
   - Decorators for independent functions
   - Decorators for classes

Data vs. Class Attributes

- Java's instance variables = Python's data attributes
- Java's static variables = Python's class attributes

```
class Triple:

    count = 0 # class attribute

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.z = z
        Point.count += 1
```

Data vs. Class Attributes

Class attributes can be accessed directly through the class, rather than through an instance (though that works too):

```
>>> Triple.count
0
>>> t = Triple(1,5,9)
>>> Triple.count
1
>>> t.count
1
```

getattr

The following are equivalent:

```
>>> t = Triple(1,5,9)
>>> t.x
1
>>> getattr(t, 'x')
1
```

`getattr` takes as input 1) either an instance or a class and 2) the *string name* of an attribute or method

getattr

Difference between passing a class vs. instance:

```
>>> f = getattr(Triple, 'show')
>>> f(p)
<1, 5, 9 >
>>> g = getattr(p, 'show')
>>> g()
<1, 5, 9 >
```

Single Inheritance

- `class Circle(Shape):` inherits from Shape
- Make sure to call the `__init__` of the parent class

```
class Circle(Shape):
  def __init__(self):
    Shape.__init__(self)
    self.new_var = default
```

- All methods are inherited from parent class

Super

- If `Shape` inherits from `object` :

```
class Circle(Shape):
  def __init__(self):
    super(Circle, self).__init__()
    self.new_var = default
```

Multiple Inheritence

- You can inherit from multiple super classes

```
class Circle(Shape, Drawable):
  def __init__(self):
    super(Circle, self).__init__()
```

- The resolution order depends on the class
  - Most classes use depth-first-search up the parent graph, starting with the first parent.
  - If the class descends from object, it uses the *C3* order which is somewhat more complex (standard in Python 3)

_ and __

- A leading _ means use at your own risk
- `from mod import *` will not import names with a leading _
- Two leading _ will trigger name mangling
- `__some_var` → `_classname__some_var`
  - `classname` is the name of the class which `__some_var` was defined in

We're all adults here

- You can still access any variable that you want
- If you know the classname and variable you can do the mangling yourself
- The purpose is to prevent subclasses from accidentally overwriting stuff

Magic Methods

- Syntactic sugar is done with magic methods
- Methods of the form `__method_name__` are "magic"
- Things like `len()` and `seq[i]` are magic method calls
- Check out Rafe Ketter's tutorial: http://www.rafekettler.com/magicmethods.html

__new__, __init__, __del__, __call__

- `x = C()` → `x = C.__init__(C.__new__())`
- `__new__` creates a new object
- `__init__` initializes it
- `del x` removes the binding of x in the current scope
  - If x was the last reference to an object, obj `obj.__del__()`
- `x(arg,...)` → `x.__call__(arg,...`

__str__, __repr__, __format__

- `str(x)` → `x.__str__()`
  - Returns a human readable string
- `repr(x)` → `x.__repr__()`
  - Returns a complete description of object
- `'{f_str}'.format(x)` → `x.__format__(f_str)`
  - Formats x according to f_str

Comparisons

- `x < y` → `x.__lt__(y)`
- `x > y` → `x.__gt__(y)`
- `x <= y` → `x.__le__(y)`
- `x >= y` → `x.__ge__(y)`
- `x == y` → `x.__eq__(y)`
- `x != y` → `x.__ne__(y)`

__hash__ and __eg__

- hashing is used in dictionaries and sets
- User defined objects default to reference equality
- If you define `__eq__` but not `__hash__` the object is unhashable
- Defining equality and hashing for subclasses is tricky

Containers

- `len(x)` → `x.__len__()`
- `x[i]` → `x.__getitem__(i)`
- `x[i] = y` → `x.__setitem__(i, y)`
- `x[start:stop:step]` → `x.__getitem__(slice(start, stop, step))`
- `k in x` → `x.__contains__(k)`

Numeric Types

- All the arithmetic operators have magic methods
- `__add__`, `__sub__`, `__mod__`, `__xor__`, ...
- Additional methods for `+=` and others

Outline

1. Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods

2. Decorators
   - Decorators for independent functions
   - Decorators for classes

Decorators

- Decorators are transformations on functions
  - A function that takes in a function and returns a modified function

```
@dec
def func(arg1, arg2, ...):
  ...
```

- Is equivalent to

```
def func(arg1, arg2, ...):
  ...
func = dec(func)
```

Decorator Arguments

- A decorator can take arguments
- `@decmaker(argA, argB, ...)`
  `def func(arg1, arg2, ...):`
  `  ...`
- Is equivalent to

```
def func(arg1, arg2, ...):
  ...
func = decmaker(argA, argB, ...)(func)
```

- `decmaker(argA, argB, ...)` returns a regular decorator

Multiple Decorators

- ```
  @dec1
  @dec2
  def func(arg1, arg2, ...):
    ...
  ```
- Is equivalent to

```
def func(arg1, arg2, ...):
  ...
func = dec1(dec2(func))
```


```

@property and @setter

- Decorate an instance method with `@property` to use `C.attr`
- Decorate with `@attr.setter` to define a setter method
  - Gets called in `C.attr = val`
- Decorate with `@attr.deleter` to define a deleter method
  - Gets called in `del C.attr`
- All decorated functions for a property must have same name

@classmethod and @staticmethod

- `@staticmethod`
  - A static method doesn't receive a `self` argument
  - Static methods should not depend on class attributes
- `@classmethod`
  - A class method gets the class object as `self`
  - Call the first argument `cls`
  - Class methods use
    - Class variables
    - other classmethods
    - staticmethods

Making a decorator

- Decorators can be defined as classes
- For decorators with no args
  - `__init__(self, old_f)`
  - `__call__(self, *args, **kwargs)`
- For decorators with args
  - `__init__(self, dec_args)`
  - `__call__(self, old_f)`
  - `__call__` needs to return `new_f`