CIS192 Python Programming
Object Oriented Programming

Harry Smith

University of Pennsylvania

February 8, 2016
Outline

1 Decorators
   - Decorators for Independent Functions

2 Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods
   - Decorators for Classes
Decorators

- Weekly reminder: functions are first class
- Decorators are transformations on functions
  - A function that takes in a function and returns a modified function

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

- Is equivalent to

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

- A decorator can take arguments

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

- Is equivalent to

```python
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))
Outline

1 Decorators
   - Decorators for Independent Functions

2 Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods
   - Decorators for Classes
Instance vs. Class Attributes

- Java’s instance variables = Python’s instance attributes
  - `self` is necessary: `self.data`
- Java’s static variables = Python’s class attributes

```python
class Triple:
    count = 0  # class attribute

    def __init__(self, x, y, z):
        self.x = x  # data attribute
        self.y = y  # data attribute
        self.z = z  # data attribute
        Triple.count += 1
```
Instance vs. Class Attributes

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

```python
>>> Triple.count
0
>>> t = Triple(1,5,9)
>>> Triple.count
1
>>> t.count
1
```
The following are equivalent:

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

ggetattr 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:

```python
>>> 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
- **super()** provides a way to access superclass
  - super() → Same as super(<class name>, self)
- Make sure to call the `__init__` of the parent class

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

- All methods are inherited from parent class
Multiple Inheritance

- You can inherit from multiple super classes
  ```python
class Circle(Shape, Drawable):
    def __init__(self):
        super().__init__()
  ```

- Attributes are resolved via the Method Resolution Order (MRO)
  - In Python 3, default is the C3 order – complicated!
  - C3 Linearization on Wikipedia
"Private" attributes: _ and __

- A leading _ means, 'Seriously, don't use this.'
- `from my_module import *` will not import names with a leading _
- Two leading _ will trigger name mangling
  ```python
  __some_var → _classname__some_var
  ▶ classname is the name of the class which __some_var was defined in
  ```
You can still access any variable that you want!

- Unlike in Java: `private/protected/public`

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()` $\rightarrow$ `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.__del__()`
- `x(arg,...) $\rightarrow$ x.__call__(arg,...)"
**__str__, __repr__**

- `str(x) → x.__str__()`
  - Returns a human readable string describing object
- `repr(x) → x.__repr__()`
  - Returns a string describing object
  - `print(x)` prints `repr(x)`!
Comparisons

- $x < y \rightarrow x.__lt__(y)$
- $x > y \rightarrow x.__gt__(y)$
- $x <= y \rightarrow x.__le__(y)$
- $x >= y \rightarrow x.__ge__(y)$
- $x == y \rightarrow x.__eq__(y)$
- $x != y \rightarrow x.__ne__(y)$
__hash__ and __eq__

- 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)`
All the arithmetic operators have magic methods 
`__add__`, `__sub__`, `__mod__`, `__xor__`, ...

Additional methods for `+=` and others
@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
  - Just a normal function that lives inside a class!

- **@classmethod**
  - A class method gets the class object as `cls`
  - Calling `a.class_method()`
  - `class_method` has access to `A` (class of object `a`)
  - Respects subclassing
  - Class methods can use
    - Class attributes
    - The class itself! (to create new instances of `cls`)
    - other class methods
    - static methods
Creating Custom Decorator Classes

- Decorators can be defined as classes
- For decorators with no args
  - __init__(self, old_f)
  - __call__(self, *args, **kwargs)
    - Runs each time you call old_f
- For decorators with args
  - __init__(self, dec_args)
  - __call__(self, old_f)
    - Runs once as part of the decoration process!
    - Needs to return new_f