CIS192 Python Programming

Object Orientation

Eric Kutschera

University of Pennsylvania

February 6, 2015
1. Last Time (Functional Programming)
   - Partial Application
   - Decorators

2. Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods
   - Decorators for classes
Partial Application

- Partial application creates a new function by supplying an existing function with some of its arguments.
- **Say you have** `add(x, y): x + y`
- **You want** `add_3(y): 3 + y`
- `add_3 = add(3)` raises a `TypeError`
- `add_3 = functools.partial(add, 3)`
Outline

1. Last Time (Functional Programming)
   - Partial Application
   - Decorators

2. Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods
   - Decorators for classes
Decorators are transformations on functions.

- A function that takes in a function and returns a modified function

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

Is equivalent to

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

- A decorator can take arguments

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

- Is equivalent to

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

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

@dec1
@dec2
def func(arg1, arg2, ...):
    pass

Is equivalent to

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

1. Last Time (Functional Programming)
   - Partial Application
   - Decorators

2. Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods
   - Decorators for classes
Classes are ways to define new objects

class Foo:
    statement1
    statement2 ...

Creates a new class object named Foo

Class definitions create a new namespace (scope)

Variables defined in the class body are class attributes

Functions defined in the class body are instance methods
__init__

- __init__(self) constructs an instance of the class
- x = Foo()
  - Creates an object of type Foo (self)
  - Calls Foo.__init__(self)
  - Binds self to the name x
- Initialize an instance variable, x in __init__ with
  - self.x = default
Attribute Lookups

- `Foo.__dict__` is a dictionary storing class attributes
- `Foo.val` translates to `Foo.__dict__['val']`
- Given `x = Foo()` then `x.__dict__` is a dictionary storing instance attributes
- `x.val` translates to:
  - `x.__dict__['val']` if `val` is an instance attribute
  - `Foo.__dict__['val']` if there is no instance attribute named `val` but there is a class attribute named `val`
Instance methods

- Instance method definitions must use `self` as the first argument

```python
class Foo:
    def no_arg(self):
        pass
    def two_args(self, a, b):
        return (a, b)
```

Given `x = Foo()`

- `x.no_arg() → Foo.__dict__['no_arg'](x)`
- `x.two_args(1, 2) → Foo.__dict__['two_args'](x, 1, 2)`
- `Foo.no_arg(Foo) → Foo.__dict__['no_arg'](Foo)`
Outline

1. Last Time (Functional Programming)
   - Partial Application
   - Decorators

2. Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods
   - Decorators for classes
Single Inheritance and super

- Super classes are arguments to the `class` statement
- `object` is the default base class
- `class Foo: 
  class Foo(object)
- class Circle(Shape): inherits from Shape
- Make sure to call the `__init__` of the super class

```python
class Circle(Shape):

    def __init__(self):
        super().__init__(self)
        self.new_var = default
```
You can inherit from multiple super classes

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

Attributes will be resolved via the MRO (Method Resolution Order)
The default MRO is C3
    Probably don’t need to know how this works
**Super**

- `super()` when an instance method of a class C
  - `super(C, self)`
- `super(cls, obj)`
  - Class that precedes `cls` in the MRO of `obj`
  - It’s **bound** → `obj` gets inserted into method calls
- `super(cls, cls2)`
  - Class that precedes `cls` in the MRO of `cls2`
  - It’s **un-bound** → `self` yet to be supplied to method calls
1. Last Time (Functional Programming)
   - Partial Application
   - Decorators

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

1. Last Time (Functional Programming)
   - Partial Application
   - Decorators

2. Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods
   - Decorators for classes
Syntactic sugar is done with magic methods
Methods of the form `__method_name__` are “magic”
Things like `f()` and `seq[i]` are magic method calls
__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.__del__()`
- `x(arg,...)` → `x.__call__(arg,...) `

Eric Kutschera  (University of Pennsylvania)  
CIS 192  
February 6, 2015  
22 / 32
\texttt{\_\_str\_\_, \_\_repr\_\_, \_\_format\_\_}

- \texttt{str(x) \rightarrow x.__str__()}  
  Returns a human readable string
- \texttt{repr(x) \rightarrow x.__repr__()}  
  Returns a complete description of object
- \texttt{\'\{f\_str\}\'.format(x) \rightarrow x.__format__\_\_(f\_str)}  
  Formats \(x\) according to \(f\_str\)
Comparisons

- $x < y \rightarrow x.__lt__(y)$
- $x > y \rightarrow x.__gt__(y)$
- $x \leq y \rightarrow x.__le__(y)$
- $x \geq y \rightarrow x.__ge__(y)$
- $x == y \rightarrow x.__eq__(y)$
- $x != y \rightarrow x.__ne__(y)$
__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.
getattr

- `x.value ⇐⇒ getattr(x, 'value')`
- `getattr(self, name)` calls `__getattribute__(self, name)` which falls back on `__getattr__(self, name)`
- Defining `__getattr__` is useful to specify default values
- `getattr(x, 'value', default)` lets you give a default if everything else fails
Containers

- \texttt{len(x)} $\rightarrow$ \texttt{x.__len__()}
- \texttt{x[i]} $\rightarrow$ \texttt{x.__getitem__(i)}
- \texttt{x[i] = y} $\rightarrow$ \texttt{x.__setitem__(i, y)}
- \texttt{x[start:stop:step]} $\rightarrow$
  \texttt{x.__getitem__(slice(start, stop, step))}
- \texttt{k in x} $\rightarrow$ \texttt{x.__contains__(k)}
All the arithmetic operators have magic methods
`__add__`, `__sub__`, `__mod__`, `__xor__`, ...
Additional methods for `+=` and others
Outline

1. Last Time (Functional Programming)
   - Partial Application
   - Decorators

2. Object Orientation
   - Class Basics
   - Inheritance
   - "Private" attributes
   - Magic Methods
   - Decorators for classes
@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
Decorators can be defined as classes

For decorators with no args

- \texttt{\_\_init\_}(self, old\_f)
- \texttt{\_\_call\_}(self, *args, **kwargs)

For decorators with args

- \texttt{\_\_init\_}(self, dec\_args)
- \texttt{\_\_call\_}(self, old\_f)
- \texttt{\_\_call\_ \textbf{needs to return} new\_f}