Tweets from @raymondh

There is more to making a FlyingCar than simply subclassing from Airplane and Car

Composition generally beats inheritance, unless the composition pattern essentially reinvents what inheritance already does

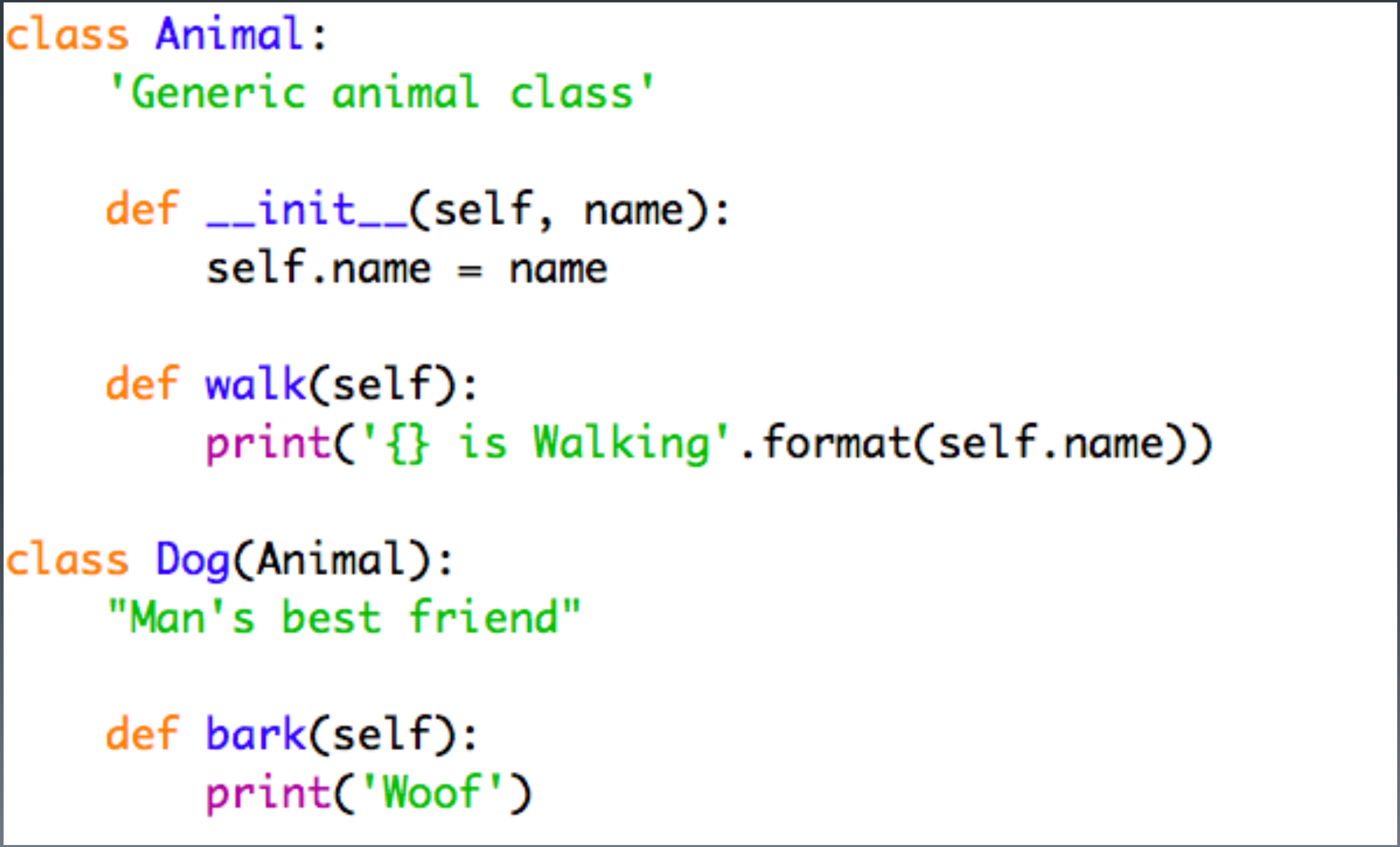
The Art of Subclassing

by Raymond Hettinger
@raymondh

Where we're headed

- Revisit the notion of a class and subclass with a view to deepening our core concepts of what it means to be a class, a subclass, or an instance.
- Discuss uses cases, principles, and design patterns
- Demonstrate how super() works and how to tame it
- Use examples from the standard library

Simple example of subclassing



```python
class Animal:
    'Generic animal class'

    def __init__(self, name):
        self.name = name

    def walk(self):
        print('{} is Walking'.format(self.name))

class Dog(Animal):
    "Man's best friend"

    def bark(self):
        print('Woof')
```

Terminology:

Adding:

Dog adds the bark() method to Animal which didn’t have a bark() method. This is a new capability.

Overriding:

Snake replaces the walk() method of Animal with a walk() method that knows how to slither.

Extending:

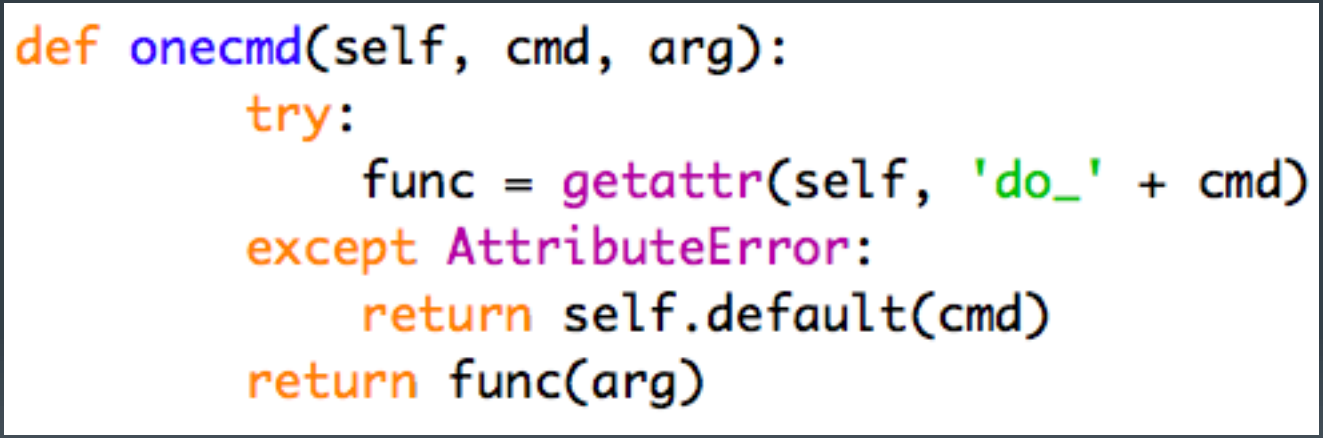
Cat modifies the walk() method to add tail swishing behavior while delegating the work of actual walking to its parent.

Pattern for Subclassing: Frameworks

- The parent class supplies all of the “controller” functionality and makes calls to pre-named stub methods.
- The subclass overrides stub methods of interest.
- For example, SimpleHTTPServer runs an event loop and dispatches HTTP requests to stub methods like do_HEAD() and do_GET()
- Someone writing an HTTP server would use a subclass to supply the desired actions in the event of a GET or HEAD request.

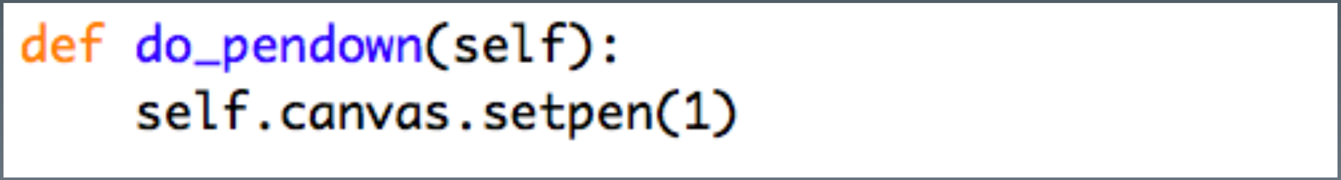
Pattern for Subclassing: Dynamic dispatch to subclass methods

- The parent class uses getattr() to dispatch to new functionality
- The child class implements appropriately named methods
- Example from cmd.py:



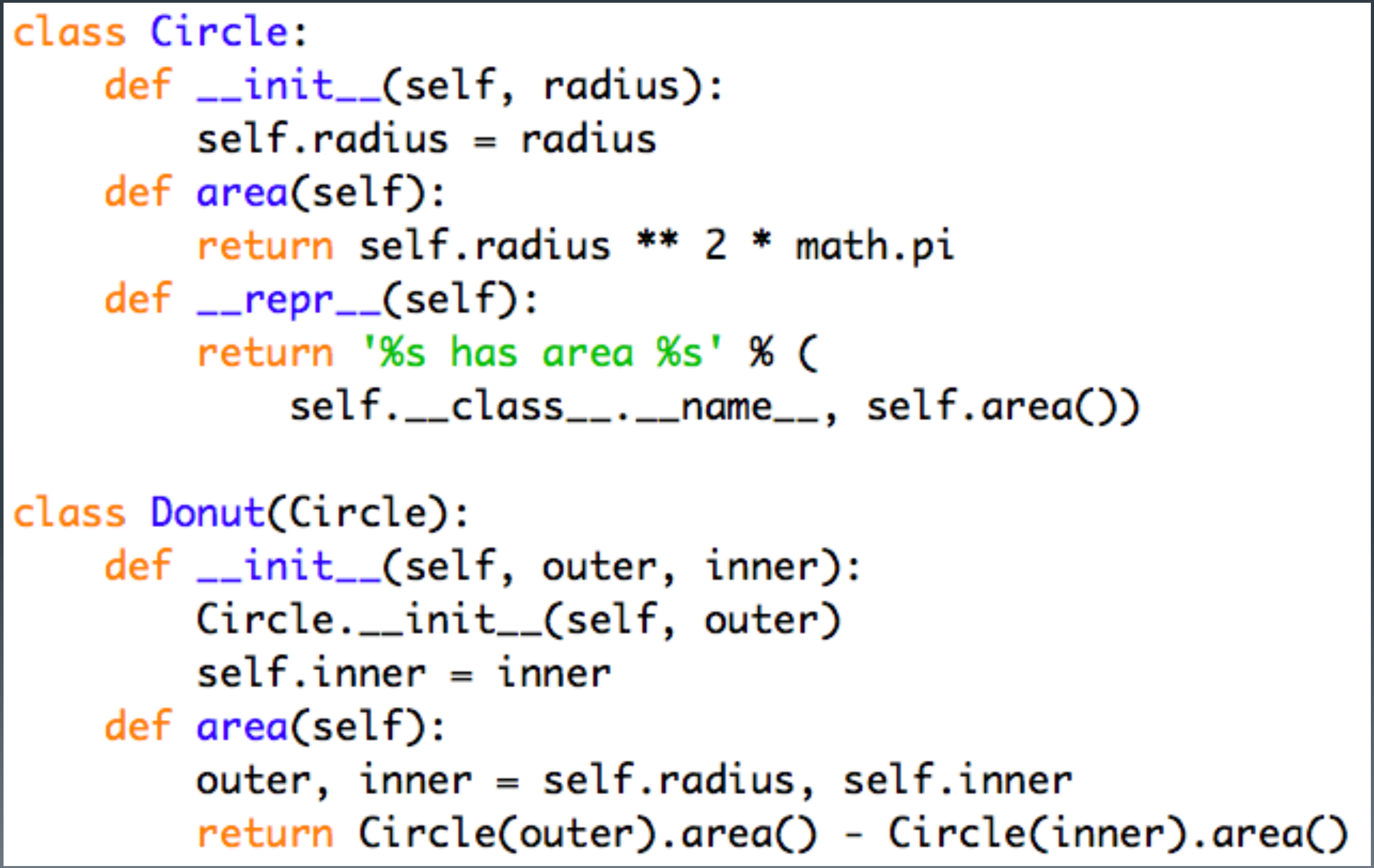
```python
def onecmd(self, cmd, arg):
        try:
            func = getattr(self, 'do_' + cmd)
        except AttributeError:
            return self.default(cmd)
        return func(arg)
```

- How it might be used in a subclass:



```python
def do_pendown(self):
    self.canvas.setpen(1)
```

Call Patterns for Subclassing:



```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius ** 2 * math.pi
    def __repr__(self):
        return '%s has area %s' % (
            self.__class__.__name__, self.area())

class Donut(Circle):
    def __init__(self, outer, inner):
        Circle.__init__(self, outer)
        self.inner = inner
    def area(self):
        outer, inner = self.radius, self.inner
        return Circle(outer).area() - Circle(inner).area()
```

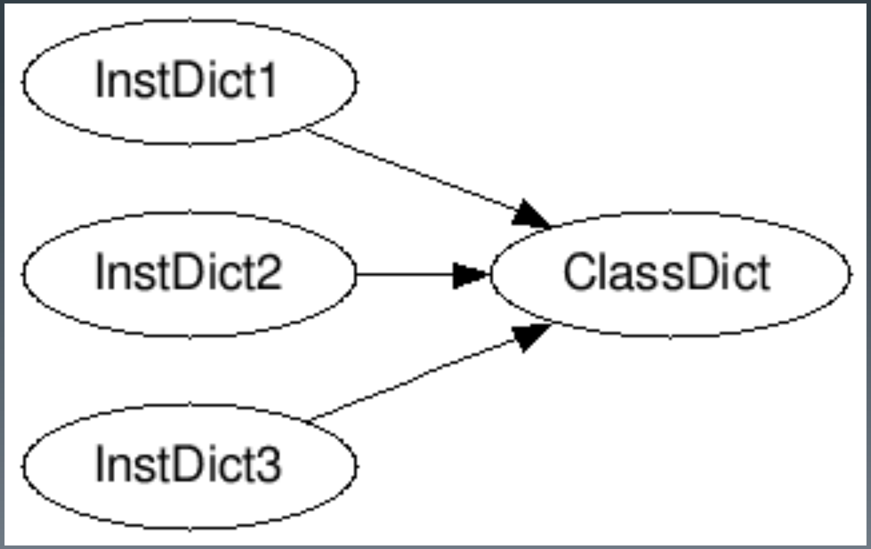
Now, let’s retool our thinking about subclasses

What does it mean to be an object or class?

One common definition an object: An entity that encapsulates data together with functions (methods) for manipulating that data.

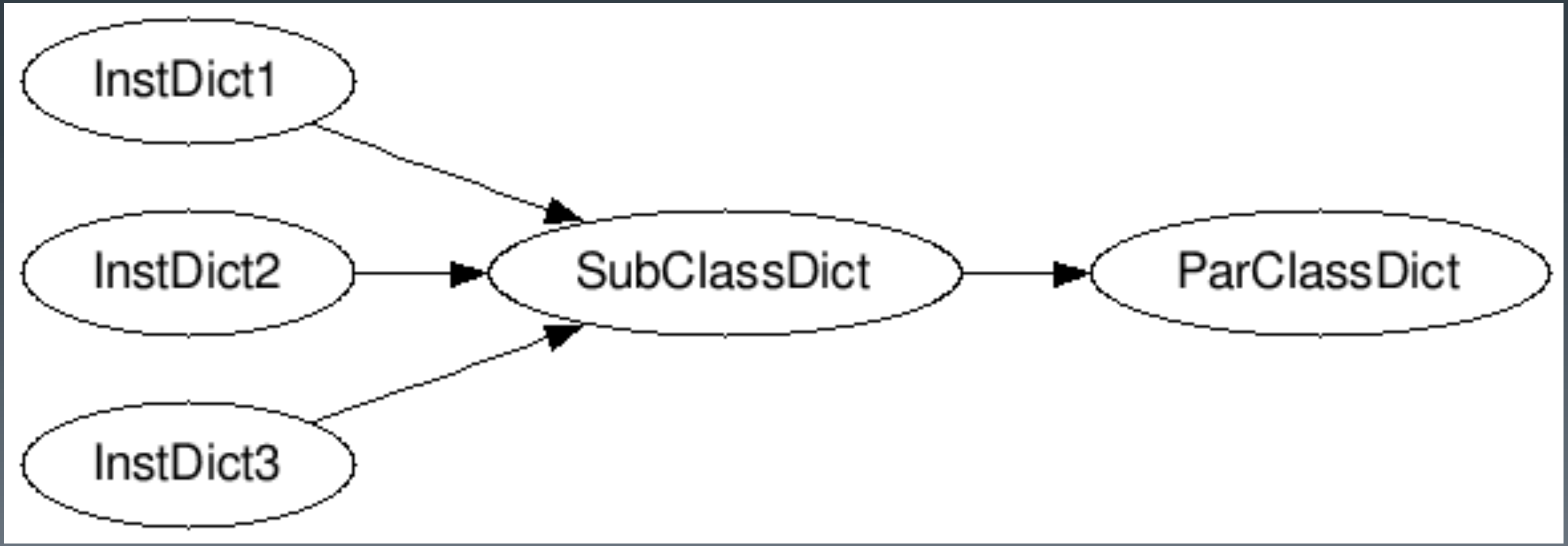
Operationally, we implement that with dictionaries:

- Instance dictionaries hold state and point to their class
- Class dictionaries hold the functions (methods)



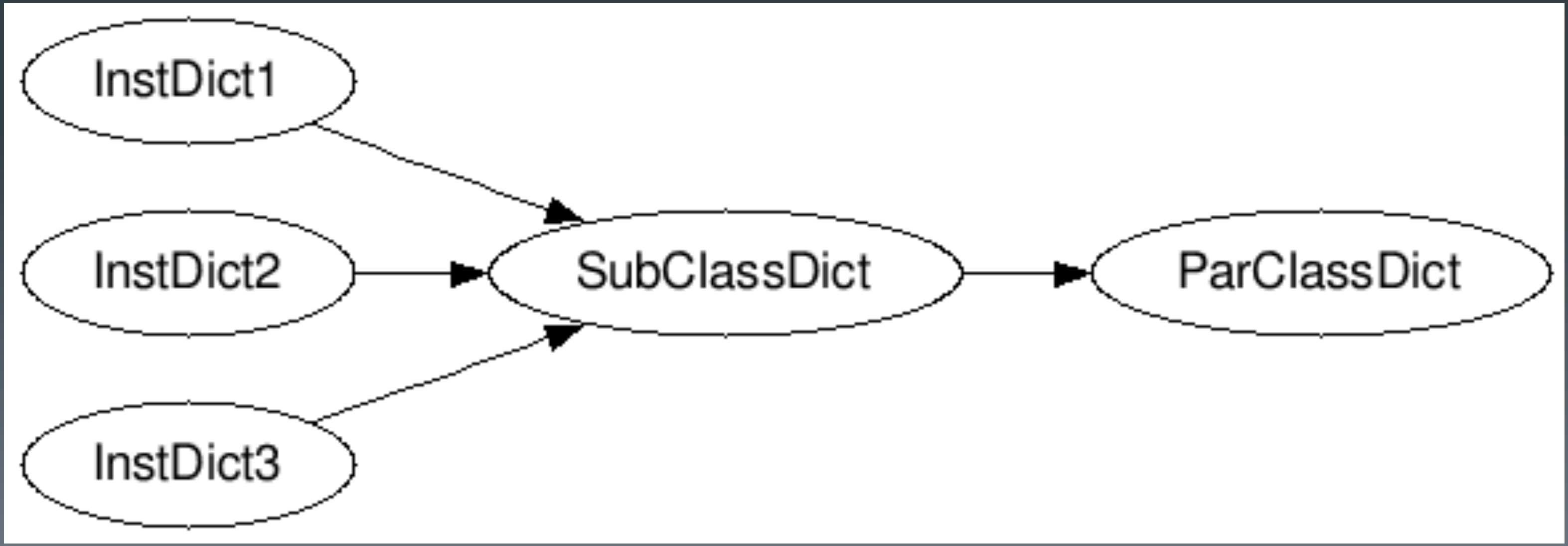
So what is a subclass?

- A subclass is just a class that delegates work to another class
- A subclass and its parent are just two different dictionaries that contain functions
- A subclass points to its parent
- The pointer means: "I delegate work to this class"



So what is subclassing?

- In other works, subclassing can be viewed as a technique for code re-use.
- It is the subclass that is in charge.
- The subclass decides what work gets delegated.



Contrast the operational view versus the conceptual view

Operational view of subclassing:

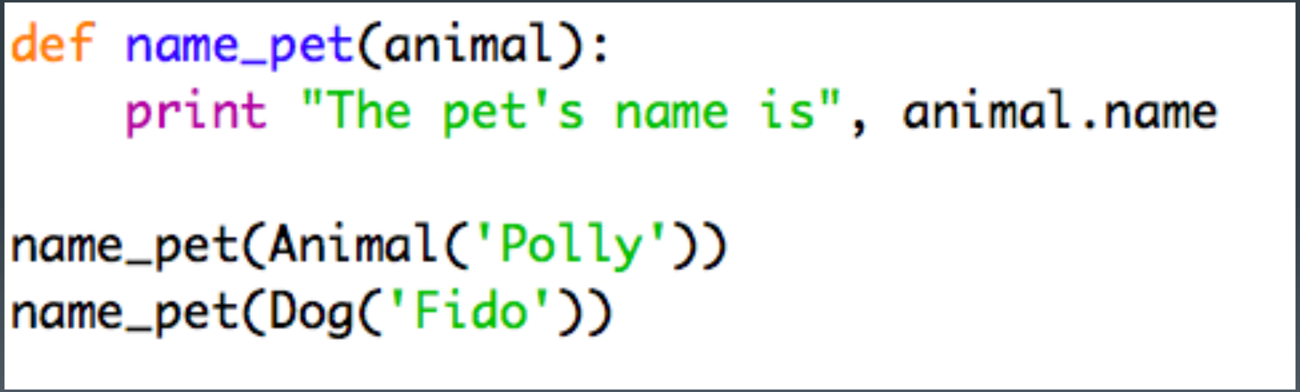
- Classes are dictionaries of functions
- Subclasses point to other dictionaries to reuse their code
- Subclasses are in complete control of what happens

Conceptual views of subclass:

- Parent classes define an interface
- Subclasses can extend that interface
- Parents are in charge
- Subclasses are just specializations of the parent
  - Dog is an instance of Animal that knows how to bark
  - Counter is an instance of dict that has a default of zero
  - A named tuple is an instance of tuple that also has attribute access

Liskov Substitution Principle

"If S is a subtype of T, then objects of type T may be replaced with objects of the S"



```
def name_pet(animal):
    print "The pet's name is", animal.name

name_pet(Animal('Polly'))
name_pet(Dog('Fido'))
```

Why do we care about Liskov?

It is all about polymorphism and substitutability so that our subclass can be used in client code without changing the client code.

Example: consider a large body of code for a cash register that calls an accept_payment() method on a payment object

We can write separate classes for credit cards, cash, checks, money orders, and coupons that all work without changing the cash register code.

We can write subclasses of the credit card class that provides custom handling for Amex, Visa, MasterCard, etc.

Substitutability is a big win

- Lots of code in python works with dictionaries
- An OrderedDict is a dict subclass that keeps most of the API intact (fully substitutable)
- Accordingly, it can be used just about everywhere in Python (the json loader and configparser were prime use cases)

Liskov Violations

- Any part of the API that isn’t fully substitutable
- This is common and normal
- In particular, useful subclasses commonly have different constructor signatures.
- For example, the array API is very similar to the list API but the constructor is different:

```
s = list(someiterable)
s = array(‘c’, someiterable)
```

Goal is to isolate or minimize the impact

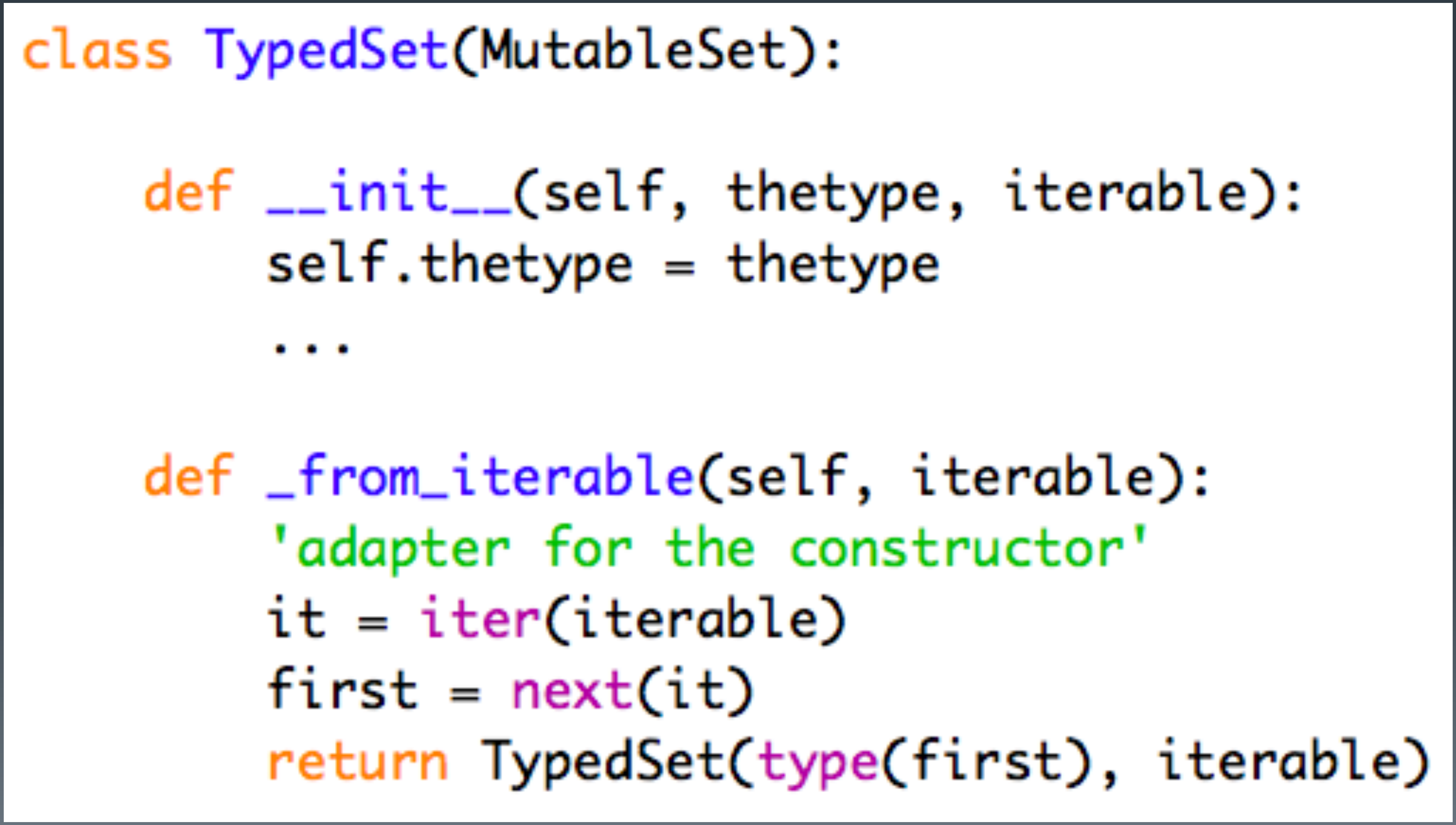
MutableSet instances support union(), intersection(), and difference()

So, they need to be able to create new instances of MutableSets

But, the signature of the constructor is unknown

So, we factor out calls to the constructor in _from_iterable()

Example of using _from_iterable()



```
class TypedSet(MutableSet):

    def __init__(self, thetype, iterable):
        self.thetype = thetype
        ...

    def _from_iterable(self, iterable):
        'adapter for the constructor'
        it = iter(iterable)
        first = next(it)
        return TypedSet(type(first), iterable)
```

The Circle / Ellipse Problem

- In mathematics, the circle is just a special case of an ellipse where the major and minor axes happen to be of equal length
- So, is the Circle an appropriate subclass of Ellipse?
- If one Ellipse method stretches an axis, what does that mean for Circle instances?
- The problem is that circles have less information than an ellipse and have constraints that don't apply to general ellispes.
- The reverse wouldn't work either because circles have capabilities that don't apply to ellipses (i.e. the bounding box is a square)

Lessons of the Circle / Ellipse Problem

- Taxonomy hierarchies do not neatly transform into useful class hierarchies.
- Substitutability can be a hard problem.
- More importantly, it challenges our conceptual view of a subclass as simply a form of specialization
- Clarity comes from thinking about the design in terms of code reuse (the class that has the most reusable code should be the parent)

The Open-Closed Principle

“software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”

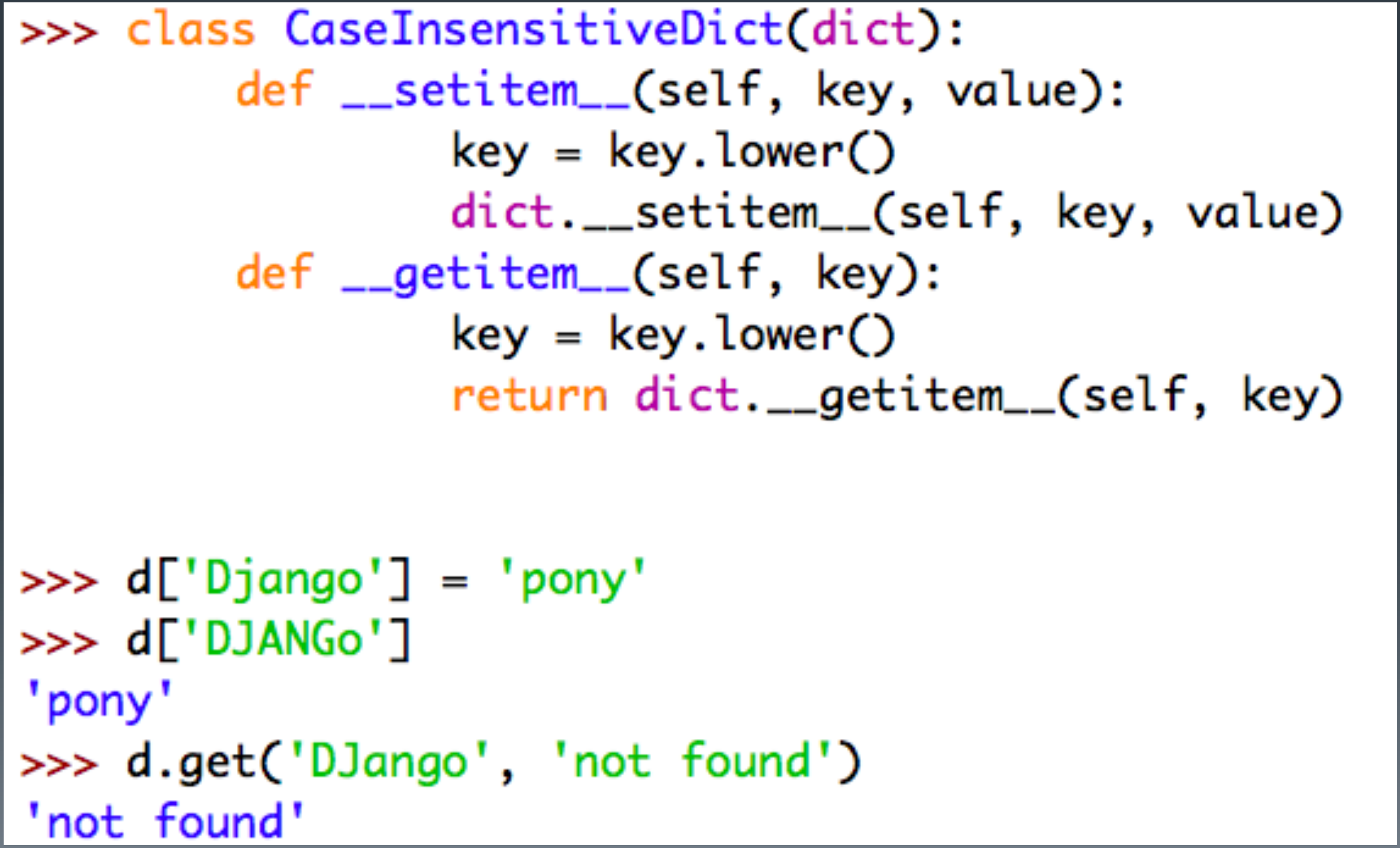
This idea has many different interpretations.

Sometimes it refers to use of abstract base classes to create fixed interfaces with multiple implementations.

The view we take is that objects have internal invariants and that subclasses shouldn’t be able to break those invariants.

In other words, the classes capabilities can be extended but the underlying class shouldn’t get broken.

Facts of life when subclassing builtins



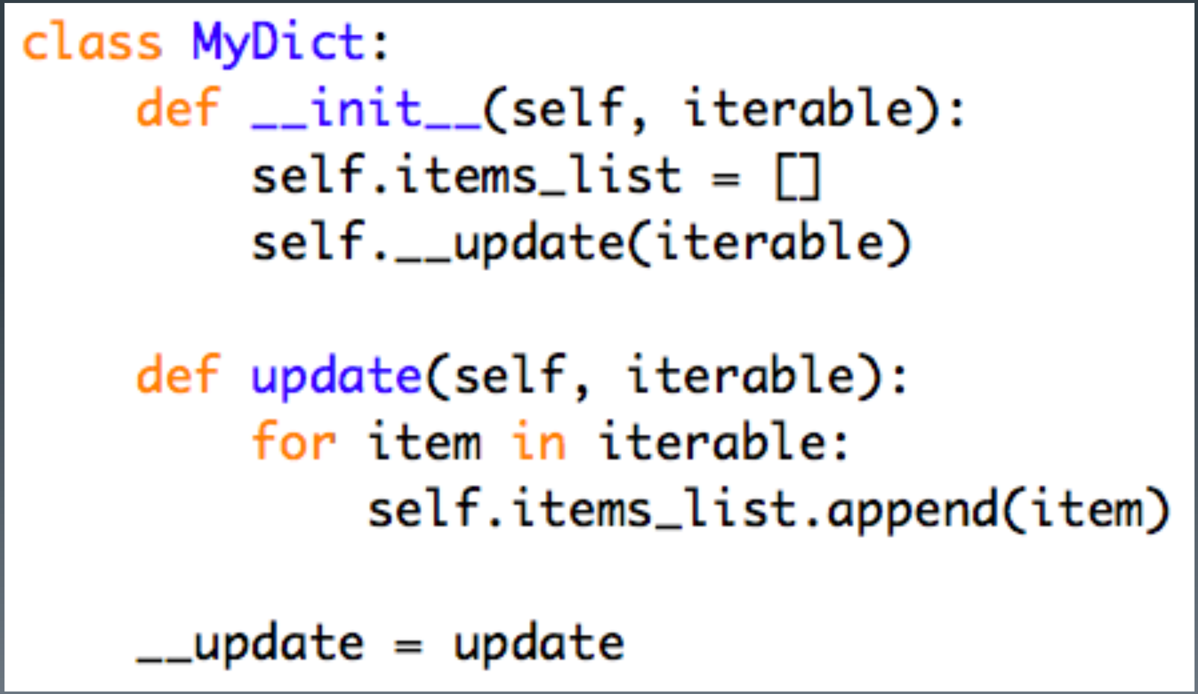
```
>>> class CaseInsensitiveDict(dict):
        def __setitem__(self, key, value):
                key = key.lower()
                dict.__setitem__(self, key, value)
        def __getitem__(self, key):
                key = key.lower()
                return dict.__getitem__(self, key)


>>> d['Django'] = 'pony'
>>> d['DJANGo']
'pony'
>>> d.get('DJango', 'not found')
'not found'
```

OCP in Python with name mangling

A method named __update in a class called MyDict transforms the name into '_MyDict__update’

This makes the method invisible to subclasses. Use this to create protected internal calls in addition to overridable public methods:



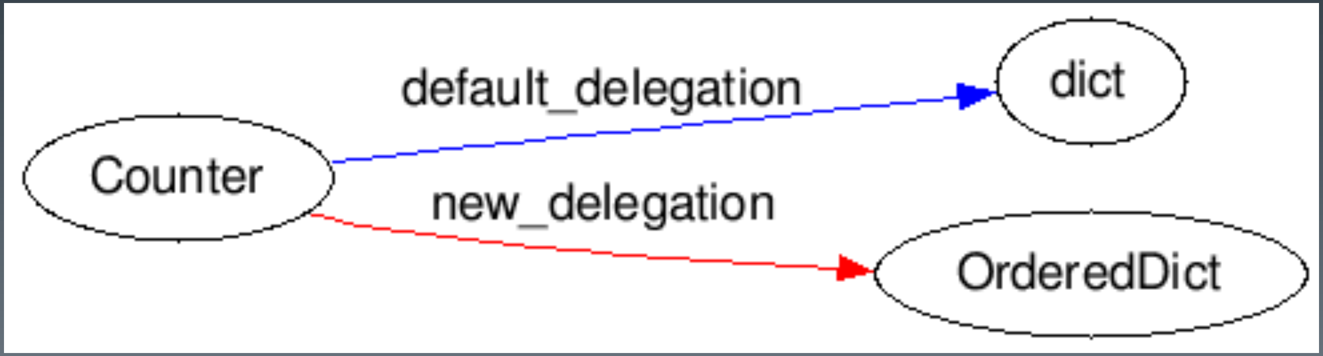
```python
class MyDict:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update
```

What Python's super() can do

- Python's super has a super power
- It lets subclasses override the default choice of delegation
- collections.Counter needs an underlying mapping
- By default it delegates that work to dict
- Subclasses can change that
- class OrderedCounter(Counter, OrderedDict): pass



Beware the Concrete C API

- PyDict_SetItem
- PyList_Append

- They operate on dict and list subclasses and there is *nothing* you can do to intercept them

Writing classes with subclassing in mind

Isolate API assumptions.   MutableSet._from_iterable

Protect internal calls.   OrderedDict.__update

Use super() when delegating work to another class

Pick specific methods designed to be overriden: do_GET()

Make a class extendable with dynamic dispatch: cmd.py

Provide hooks, overrides, or controllable delegation for every piece of functionality that a subclasser may want to change.

Writing subclasses

- Liskov substitution principle
- You’re in control
- Think strictly in terms of maximum code reuse
- Overwrite every method that needs to change (do not rely on implicit undocumented relationships).

Question and Answer

Other Issues

- Int(5) + Int(8) → real int
- Dict().copy → dict
- self.__class__.__name__