

Objects

Week 2, day 5

What is an object?

- A “thing”
- Something that can be passed around, assigned, copied
- Something that can have attributes (uses `dir()`)
- An instance of a class with defined properties and behaviours

Why objects?

- The trouble with procedural programming
 - Scales badly
 - Hides little (passing data around)
 - Reuse is tricky (coupling)
 - (Still at the base of every programming paradigm)

Object-oriented programming

- Objects have attributes: related data & functions that need to “know” about each other
 - Members (properties, data)
 - Methods (functions that work on properties)
- Hides detail from outside

What is an object? (2)

- A way to create new basic types
- An instance of a class
- An instantiation from a template
- In Python: almost everything

The simplest possible class

- `x` is the object that is constructed (or instantiated) by the function `VerySimpleClass()` from the class `VerySimpleClass`.
- `x` is an instance of `VerySimpleClass`.

```
class VerySimpleClass:  
    pass
```

```
>>> x = VerySimpleClass()
```

```
>>> type(x)  
<type 'instance'>
```

```
>>> dir (x)  
['__doc__', '__module__']
```

A less simple class

- The function to create an instance is named after the class.
- It copies the object and runs `__init__`.
- `__init__` is the class constructor.

```
class LessSimpleClass:  
    def __init__(self):  
        print "Setting prop"  
        self.prop = "A"
```

```
>>> x = LessSimpleClass()  
Setting prop
```

```
>>> dir(x)  
['__doc__', '__module__',  
'prop']
```

Members, methods, self

- `prop` is a member of `SimpleClass`
- `getProp` is a method of `SimpleClass`
- `self` means “this object / instance”

```
class SimpleClass:
    def __init__(self):
        self.prop = "A"

    def getProp(self):
        return self.prop

>>> x = SimpleClass()

>>> x.getProp()
"A"

>>> x.prop
"A"
```

Why do we need self?

```
class BadClass:
    def __init__(self):
        self.prop1 = "A"
        prop2 = "A"

    prop3 = "B"

>>> x = BadClass()

>>> dir (BadClass)

>>> dir (x)
```

Why do we need self?

(2)

- `self` is only used in the class definition
- It's a placeholder for "this instance"
- `obj.method (args)` and `class.method (obj, args)` are the same

```
class SimpleClass:
    def __init__(self):
        self.prop = "A"

    def getProp (self):
        return self.prop

>>> x = SimpleClass()

>>> x.getProp()
"A"

>>> SimpleClass.getProp (x)
"A"
```

Why objects? (2)

- Encapsulation: related functions and data are kept together
- Abstraction: hides implementation under interface
- Reflects problem domain
- Example: Coordinate

Coordinate

```
class Coord:
    """
    A class to represent geographical
    coordinates.
    """
    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon

    def getPosn (self):
        return (self.lat, self.lon)
```

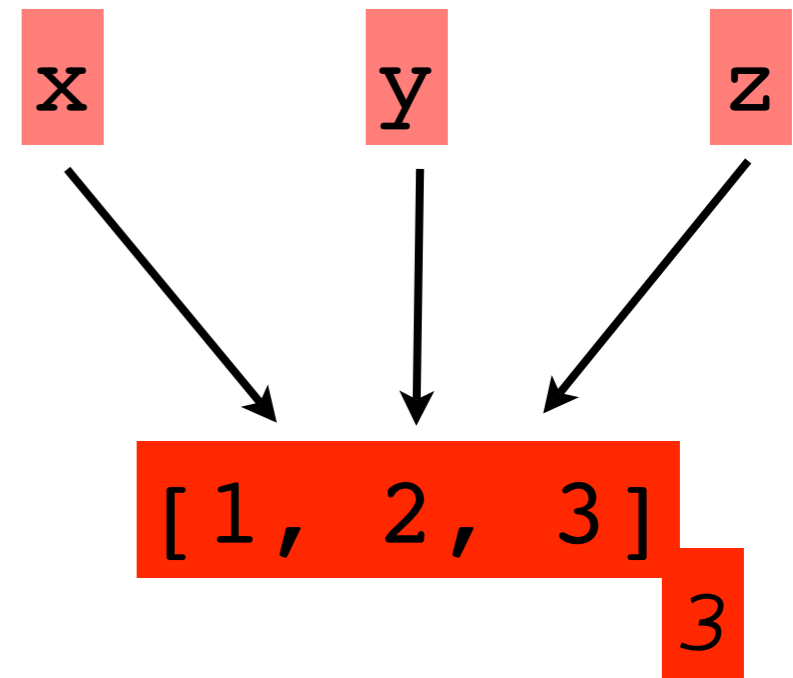
What an object can have

- docstring
- members, should be created in `__init__`
- methods, with `self` as first arg
- `__init__`, with args and default args
- `__del__`, called when object deleted

Memory management

- Largely automatic “garbage collection”
- Reference-counting
- Break cycles

```
x = [1, 2, 3]
y = x
z = x
```



Coordinate

```
class Coord:
    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon

    def getPosn (self):
        return (self.lat, self.lon)

    def toPolar (self):
        [...]

    def distanceFrom (self, other_coord):
        [...]
```

Magic methods

- “double underscore” methods
- Implement them to get particular behaviours
- e.g. `__cmp__`: how to compare two instances so they are sortable

Coordinate

```
class Coord:
    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon

    def __cmp__(self, other_coord):
        if (self.lat < other_coord.lat):
            return -1
        elif (other_coord.lat < self.lat):
            return 1
        else:
            if (self.lon < other_coord.lon):
                return -1
            elif (other_coord.lon < self.lon):
                return 1
            else:
                return 0
```

Other magic methods

- `__nonzero__`: is it true (nonzero) or not, so it can be used as a boolean
- `__len__`: size, so it can be passed to `len()`
- `__str__`: what happens when converted to string

Magic list-like methods

- So the object can act like a list
 - `__getitem__` (self, index)
 - `__setitem__` (self, index, value)
 - `__delitem__` (self, index)
 - `__getslice__` (self, start, stop)
 - etc. and dict methods

Area

```
class Area:
    "A region defined by a sequence of coordinates."

    def __init__(self, *args):
        "Call with bounds as args e.g. Area (c1, c2, c3)"
        self.bounds = args

    def __getitem__(self, index):
        return self.bounds[index]

    def __setitem__(self, index, value):
        self.bounds[index] = value

    def __delitem__(self, index):
        del self.bounds[index]
```

Magic math methods

- So the object can use normal math operators
 - `__add__` (self, other)
 - `__pow__` (self, power)
 - `__and__` (self, other)
 - `__float__` (self) etcetera etcetera