# Objects and classes

Amit Kotlovski
Open-Source Consultancy
http://amitkot.com

# Agenda

- Defining new classes
- Class methods
- Properties
- Class inheritance
- Abstract base classes

# Defining new classes

```
class LivingRoomTable(object):
    def __init__(self, size, height, color):
        self._size = size
        self._height = height
        self._color = color

    def is_high(self):
        return self._height > 50

>>> t = LivingRoomTable(120, 40, 'mahogany')
>>> print(‘is it high?', t.is_high())
is it high? False
```

# Class methods

```
class LivingRoomTable(object):
    def __init__(self, size, height, color):
        self._size = size
        self._height = height
        self._color = color

    def default_height(self):
        return 50

    def is_high(self):
        return self._height > self.default_height()

>>> t = LivingRoomTable(120, 40, 'mahogany')
>>> print('is it high?', t.is_high())
is it high? False
```

# Function → Method

Add first parameter self for accessing class instance

# Constructor method

```
class LivingRoomTable(object):
    def __init__(self, size, height, color):
        self._size = size
        self._height = height
        self._color = color
    .
    .
    .

>>> t = LivingRoomTable(120, 40, 'mahogany')
```

# Method name

- `_name`
- `name_`
- `__name`
- `__name__`

# `_internal`

- One underscore prefix
- Implementation detail
- Not part of API
- Might change without warning

# reserved_

- One underscore suffix

- When (ab)using reserved words

# __mangled

```
class A:
    def __foo(self):
        return 'A'
    def bar(self):
        return self.__foo()

class B(A):
    def __foo(self):
        return 'B'
    def bar(self):
        return self.__foo()
```

```
>>> b = B()

>>> b._A__foo()
'A'

>>> b._B__foo()
'B'

>>> b.bar()
'B'
```

# __magic__

Interacting with Python's Data Model

```
class A(object):
    def __len__(self):
        return 42

>>> a = A()
>>> len(a)
42

>>> a.__len__()
42
```

# Data Model Functions

- Class implementations
- Operator overloading

# `__len__`

The magic behind `len()` for getting object length

```
>>> l = [1, 2, 3, 4]
>>> print len(l)
4
>>> # is equal to:
>>> print l.__len__()
4
```

# `__str__`

The magic behind `str()` for getting string representation

```
>>> l = [1,2,    3,  2+2]
>>> print str(l)
[1, 2, 3, 4]
# is equal to:
>>> print l.__str__()
[1, 2, 3, 4]
```

# Operator overloading

```
class FunnyString(object):
    def __init__(self, string):
        self._data = string
    def __str__(self):
        return self._data
    def __pos__(self):
        return self._data.upper()
    def __neg__(self):
        return self._data.lower()

>>> fs = FunnyString("This is THE funniest string ever!")
>>> print fs
This is THE funniest string ever!
>>> print +fs
THIS IS THE FUNNIEST STRING EVER!
>>> print -fs
this is the funniest string ever!
```

# Non instance methods

- staticmethod
- classmethod

```
class Banana(object):
    count = 0

    def __init__(self, color):
        self._color = color
        __class__.count += 1

    @classmethod
    def how_many(cls):
        return cls.count

    @staticmethod
    def get_type():
        return "Fruit"

>>> b1 = Banana('yellow')
>>> b2 = Banana('green')
>>> print Banana.get_type()
Fruit
>>> print Banana.how_many()
2
```

# Member access

- Accessing class member variables directly
- Getters and setters
- Properties

# Direct access

```
class Table(object):
    def __init__(self, height):
        self.height = height

>>> t = Table(50)
>>> t.height = 700
>>> print t.height
700
```

⬆ Very comfortable

⬇ Breaks encapsulation

⬇ Exposes internal implementation

# Getters and setters

```
class Table(object):
    def __init__(self, height):
        self._height = height
    def get_height(self):
        return self._height
    def set_height(self, val):
        self._height = val

>>> t = Table(50)
>>> t.set_height(700)
>>> print t.get_height()
700
```

⬆ Exposes the data while hiding implementation

⬇ Annoying to implement

# Properties

## Implement getter and setter, expose direct access

```
class Table(object):
    def __init__(self):
        self._color = None

    def getcolor(self):
        return self._color

    def setcolor(self, value):
        self._color = value

    color = property(getcolor, setcolor)

>>> t = Table()
>>> t.color = 'Blue'     # property access
>>> print t.color        # property access
Blue
```

# Property decorator

```
class Table(object):
    def __init__(self):
        self._color = None

    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, value):
        self._color = value

>>> t = Table()
>>> t.color = 300
>>> print t.color
300
```

# Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

# Lab

## 3a - Creating classes, inheritance

# Class Inheritance

- Python classes can inherit from other classes or from the base `object` class

- Python supports multiple inheritance

# Inheritance

```
class Fruit(object):
    def __init__(self):
        self._type = 'fruit'
    def eat(self):
        print("eating a", self._type)

class Banana(Fruit):
    def __init__(self):
        self._type = ‘banana'
    def peel(self):
        print("peeling a", self._type)

>>> b = Banana()
>>> b.peel()
peeling a banana
>>> b.eat()
eating a banana
```

# super()

Run the parent implementation of a function

```
# Python 2
class A(object):
    def foo(self):
        return 'A'

class B(A):
    def foo(self):
        parent = super(B, self).foo()
        return 'B --> {}'.format(parent)

class C(B):
    def foo(self):
        parent = super(C, self).foo()
        return 'C --> {}'.format(parent)

>>> c = C()
>>> print(c.foo())
C --> B --> A
```

# super()

Python 3 has a simpler syntax

```
# Python 2
class B(A):
    def foo(self):
        parent = super(B, self).foo()
        return 'B --> {}'.format(parent)

# Python 3
class B(A):
    def foo(self):
        parent = super().foo()
        return 'B --> {}'.format(parent)
```

# Method resolution

- (0) A class method was called
- (1) Search actual class
- (2) Search parent class
- (3) Search parent's parent class
- …
- If we reach object and don't find it an `AttributeError` is raised

# Method resolution

```
class A(object):
    def whoami(self):
        print('A')
    def family(self):
        print('letters')


class B(A):
    def whoami(self):
        print('B')

>>> b = B()
>>> b.whoami()    # B --> whoami()
'B'
>>> b.family()    # B --> A --> family()
'letters'
```

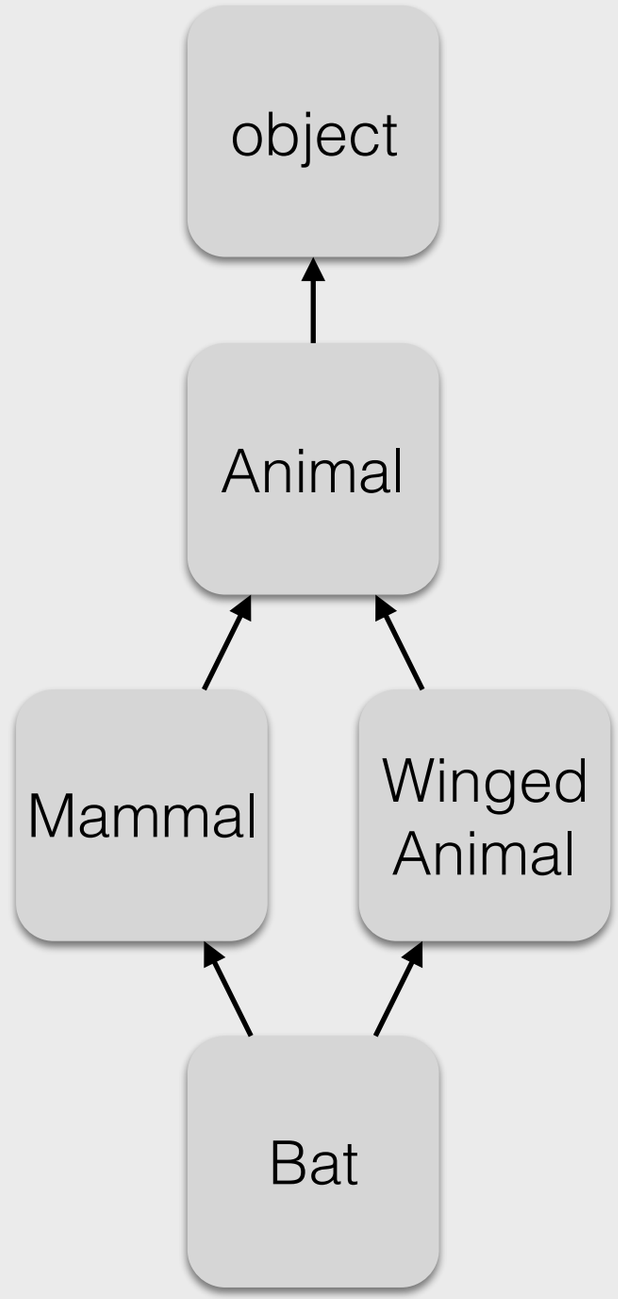
# Multiple inheritance

```
class Animal(object):
    def eat(self): pass

class Mammal(Animal):
    def breath(self): pass

class WingedAnimal(Animal):
    def flap(self): pass

class Bat(Mammal, WingedAnimal):
    pass
```



# Method resolution dilemma

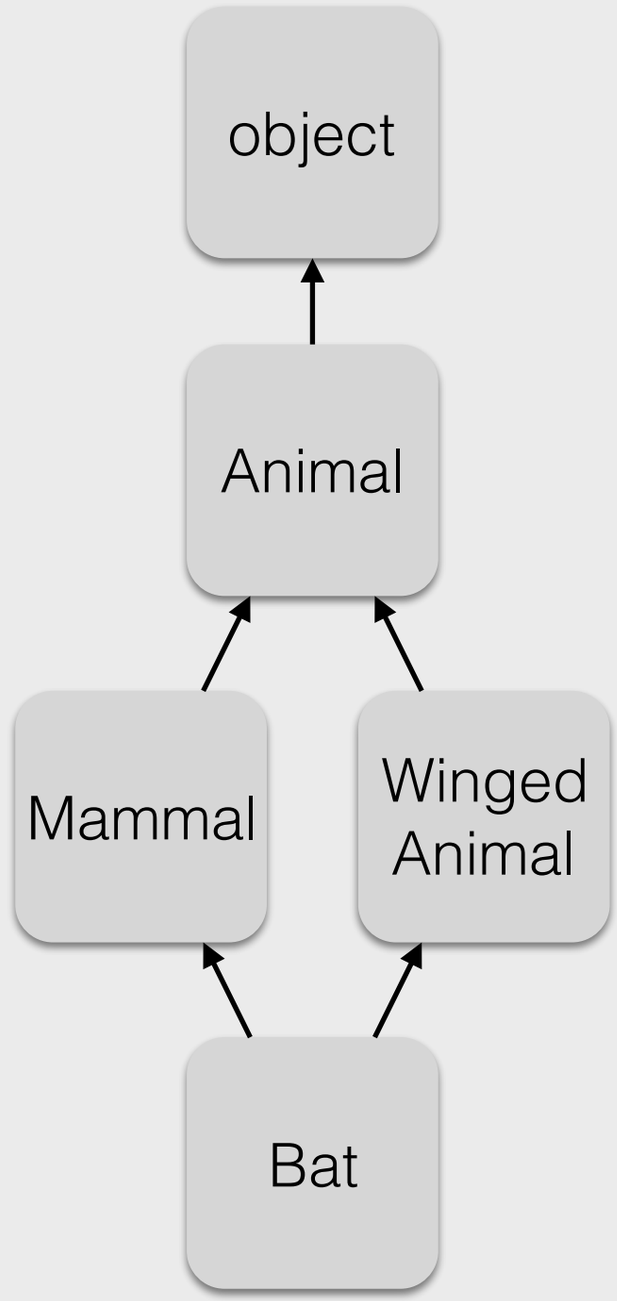
```
class Animal(object):
    def move(self):
        return "swim/fly/walk"

class Mammal(Animal):
    pass

class WingedAnimal(Animal):
    def move(self):
        return "fly"

class Bat(Mammal, WingedAnimal):
    pass

>>> b = Bat()
>>> print(b.move())
# What will be the output?
```



# Method resolution order

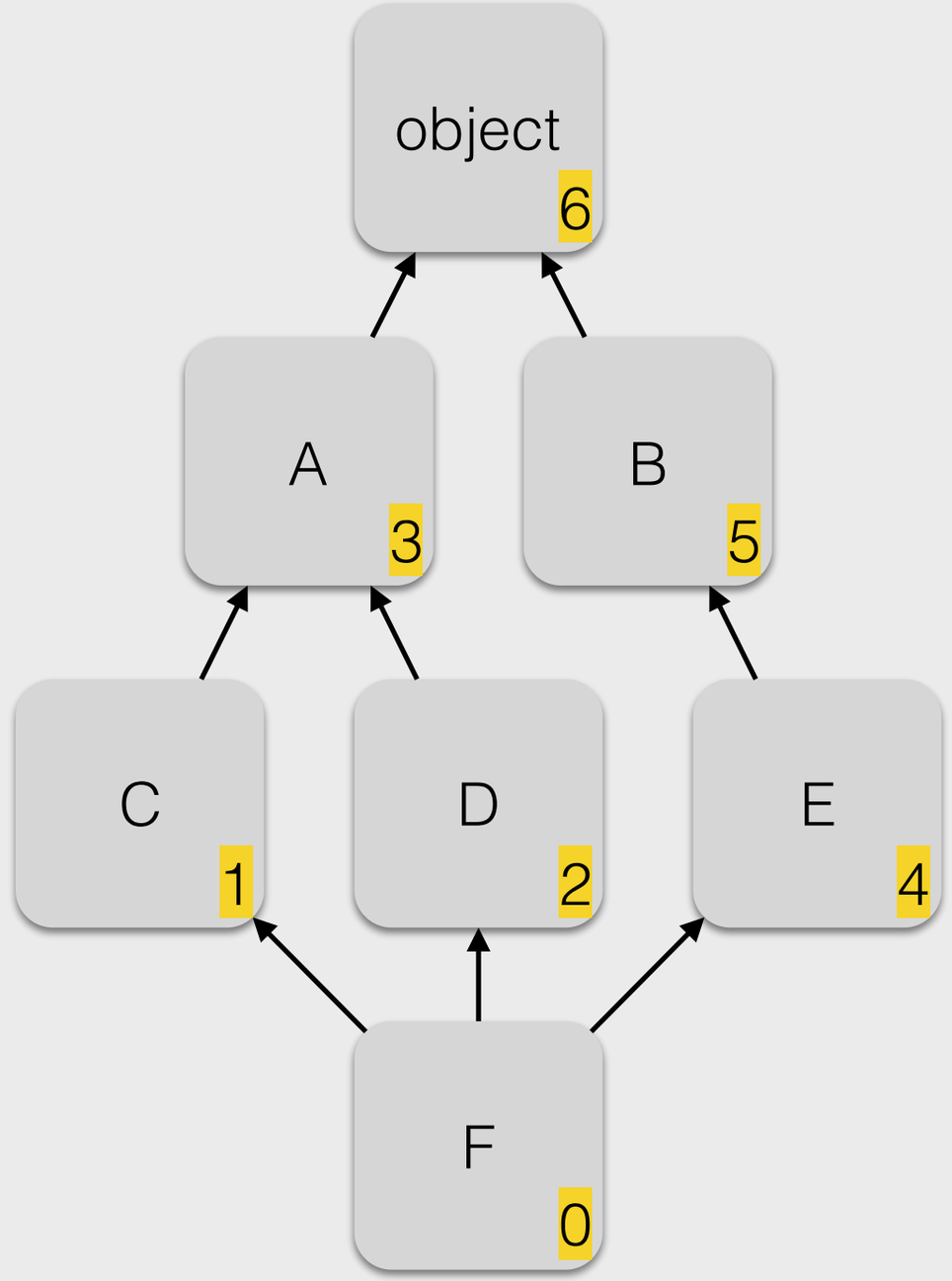
- An order that is defined during class initialisation
- Determines the order in which method resolution mechanism will look for next parent class
- Based on the C3 Linearization algorithm[1]

[1] “A Monotonic Superclass Linearization for Dylan" (K. Barrett, et al, presented at OOPSLA'96)

# MRO algorithm intuition

```
class A(object): pass
class B(object): pass
class C(A): pass
class D(A): pass
class E(B): pass
class F(C, D, E): pass

>>> F.__mro__
(__main__.F,
 __main__.C,
 __main__.D,
 __main__.A,
 __main__.E,
 __main__.B,
 object)
```



# `classname.__mro__`

- An attribute exposing the method resolution order of the class
- Class can override the function `mro()` to override default order
  - New order is stored in `__mro__`

# Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

# "Abstract" Classes

```
class Greeting:
    def greet(self):
        raise NotImplementedError()

class ThaiGreeting(Greeting):
    def greet(self):
        return "Sawadika!"
```

```
>>> g = Greeting()
>>> g.greet()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in greet
NotImplementedError
```

```
>>> tg = ThaiGreeting()
>>> tg.greet()
'Sawadika!'
```

# abc - Abstract Base Class

- Compile-time checks
- Defining Interfaces and Abstract classes

# Uses

- Define an API that classes can implement, e.g. - Plugins for an app

- Guiding 3’rd parties developers

# Defining an abstract base class

```
import abc

class Fruit(abc.ABC):

    @abc.abstractmethod
    def has_seeds(self):
        """ Checks if fruit has seeds """
```

# Implementing by inheritance

```
import abc

class Fruit(abc.ABC):
    @abc.abstractmethod
    def has_seeds(self):
        """ Checks if fruit has seeds """
```

```
class Cucumber(Fruit):
   pass

>>> c = Cucumber()
...
TypeError: Can't instantiate
abstract class Cucumber with
abstract methods has_seeds
```

```
class Mango(Fruit):
    def has_seeds(self):
        return True

>>> m = Mango()
>>> m.has_seeds()
True
>>> print(issubclass(Mango, Fruit))
True
>>> print(isinstance(Mango(), Fruit))
True
```

# Registering

- When the class implements the abstract "contract" but is not part of the inheritance tree

- builtin `issubclass()` and `isinstance()` functions - for classes and instances

- No effect on MRO

- Method implementation in abstract not accessible

```
import abc

class Fruit(abc.ABC):
    @abc.abstractmethod
    def has_seeds(self):
        """ Checks if fruit has seeds """

@Fruit.register
class Tomato:
    def has_seeds(self):
        return True
    def shape(self):
        return "Round"


>>> print(issubclass(Tomato, Fruit))
True
>>> print(isinstance(Tomato(), Fruit))
True
```

# All inheriting classes

To find all inheriting classes of a given abstract run:

```
>>> for sc in Fruit.__subclasses__():
...     print(sc.__name__)
Cucumber
Mango
```

# Calling abstract class concrete method implementation using **super()**

```
class Fruit(abc.ABC):
    @abc.abstractmethod
    def is_healthy(self):
        return True

class Pomegranate(Fruit):
    def is_healthy(self):
        res = super().is_healthy()
        return '{}, indeed!'.format(res)

>>> p = Pomegranate()
>>> p.is_healthy()
'True, indeed!'
```

# Abstract properties

- Abstract properties are decorated with `@abstractproperty`
- To provide setter and getter call `abstractproperty(getter, setter)`
- Concrete classes must implement abstract properties to be instantiated

# Abstract Property

```
class Fruit(abc.ABC):
    @property
    @abc.abstractmethod
    def taste(self): pass

class Apple(Fruit):
    @property
    def taste(self):
        return 'sour'

>>> a = Apple()
>>> print('Apples are', a.taste)
Apples are sour
```

# Abstract Property: Getter and Setter

```
class Fruit(abc.ABC):
    @property
    @abc.abstractmethod
    def taste(self):
        """Get taste of the fruit"""

    @taste.setter
    @abc.abstractmethod
    def taste_setter(self, newtaste):
        """Set taste of the fruit"""
```

```
class Banana(Fruit):
    def __init__(self):
        self._taste = 'sweet'

    @property
    def taste(self):
        return self._taste

    @taste.setter
    def taste(self, taste):
        self._taste = taste

>>> b = Banana()
>>> print('Bananas are {}'.format(b.taste))
Bananas are sweet

>>> b.taste = 'super-sweet'
>>> print('Some bananas are {}'.format(b.taste))
Some bananas are super-sweet
```

# Python 2 --> Python 3

## Defining an abstract class

```python
# Python 2
class Base(object):
    __metaclass__ = abc.ABCMeta
    pass
```

```python
# Python 3
class Base(abc.ABC):
    pass
```

# Python 2 --> Python 3

## Registering a concrete class

```
# Python 2
class Base(object):
    __metaclass__ = abc.ABCMeta
    pass

class Implementation(Base):
    pass

Base.register(Implementation)
```

```
# Python 3
class Base(abc.ABC):
    pass

@Base.register
class Implementation(Base):
    pass
```

# Python 2 --> Python 3

## Abstract Properties

```python
# Python 2
class Base(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractproperty
    def value(self):
        return None

class Implementation(Base):
    @property
    def value(self):
        return 'some property'
```

```python
# Python 3
class Base(abc.ABC):

    @property
    @abc.abstractmethod
    def value(self):
        return None

class Implementation(Base):
    @property
    def value(self):
        return 'some property'
```

# Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

# Lab

## 3b - Operator overloading, properties, abstract base classes

# Summary

- Operator overloading is easy and useful
- Properties allow regulated member access
- Python classes support multiple inheritance
- Abstract base classes allow defining “contracts”

QUESTION
EVERYTHING

# Thanks!

twitter @amitkot
www amitkot.com