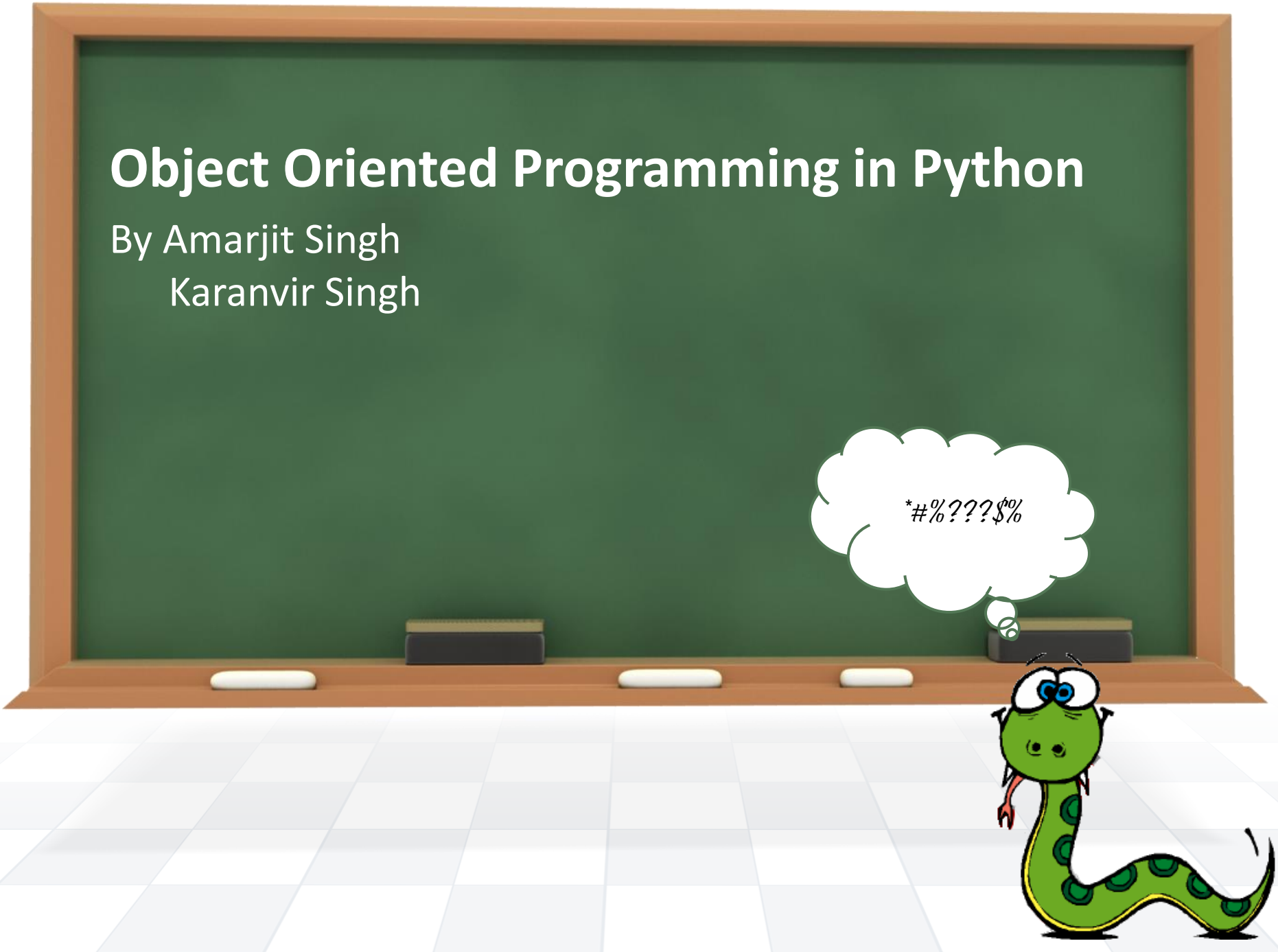
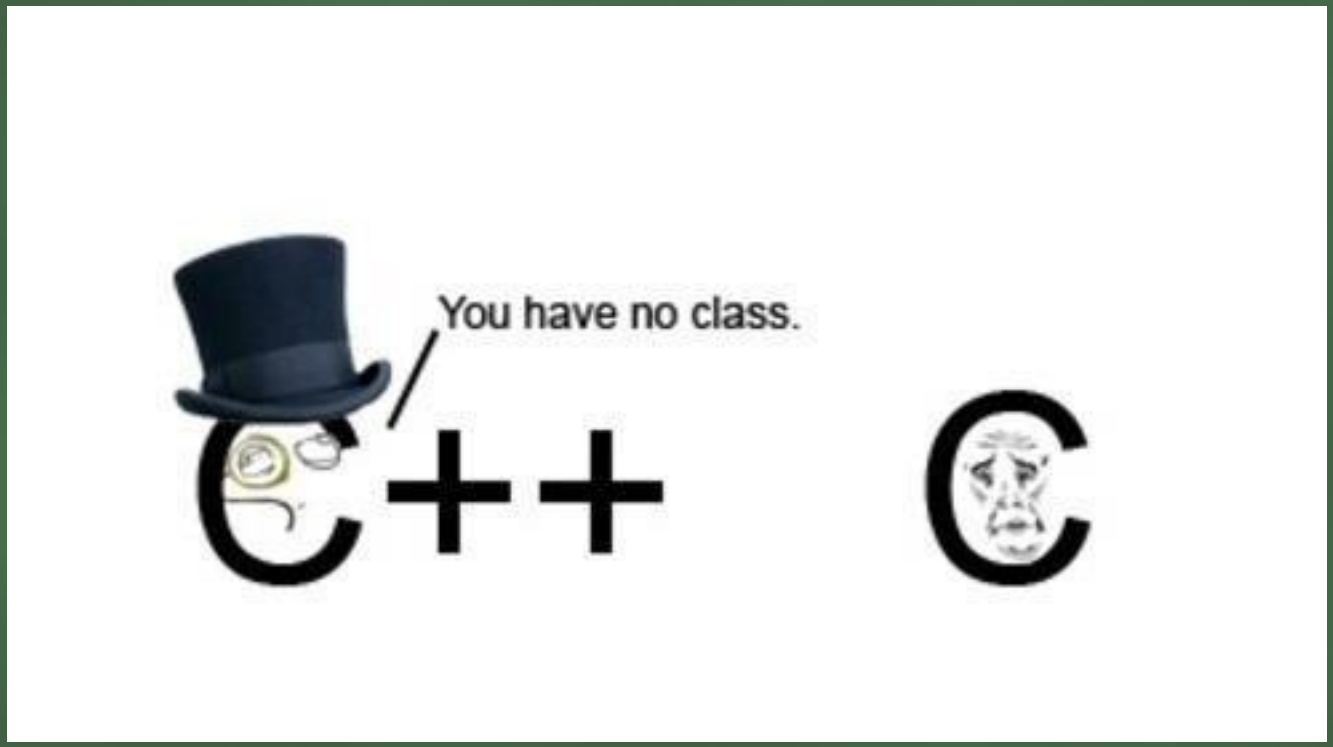
Object Oriented Programming in Python

By Amarjit Singh
Karanvir Singh



•Contents

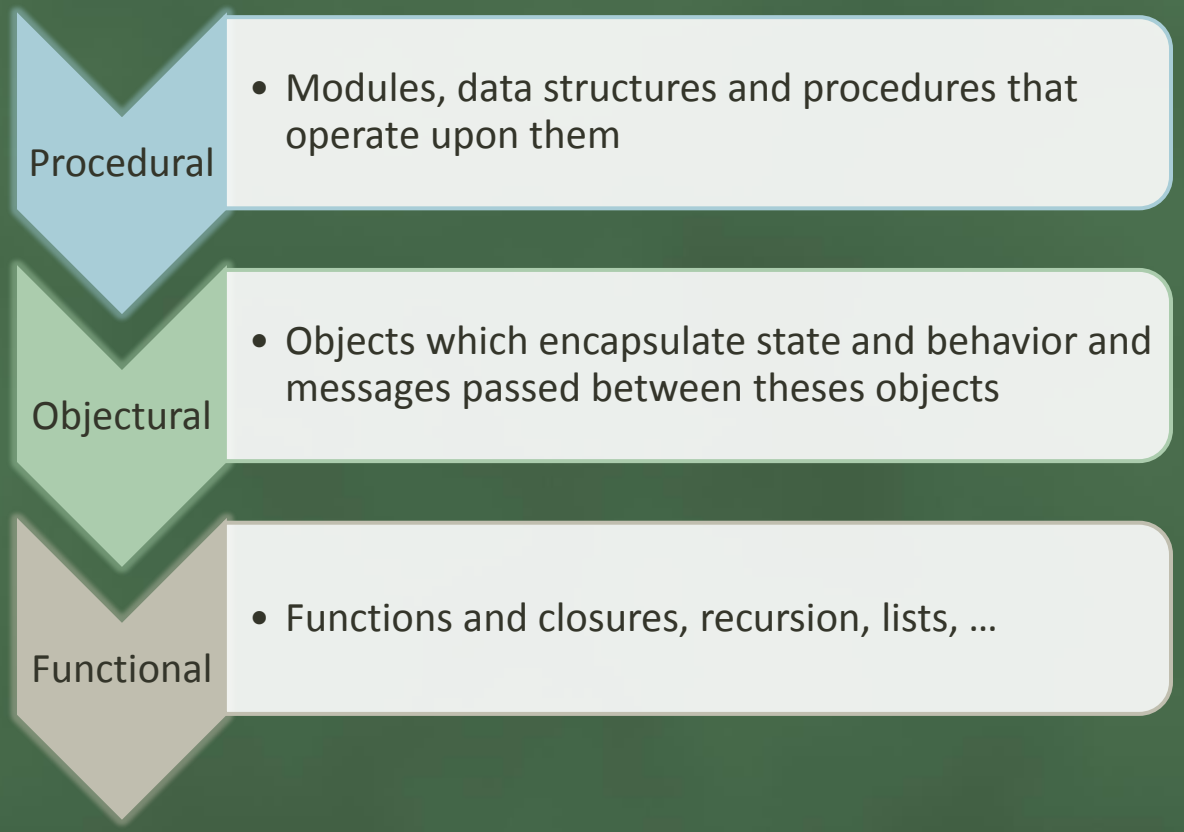
Object Oriented Programming Concepts



•Object Oriented Programming Basics

Programming Paradigms

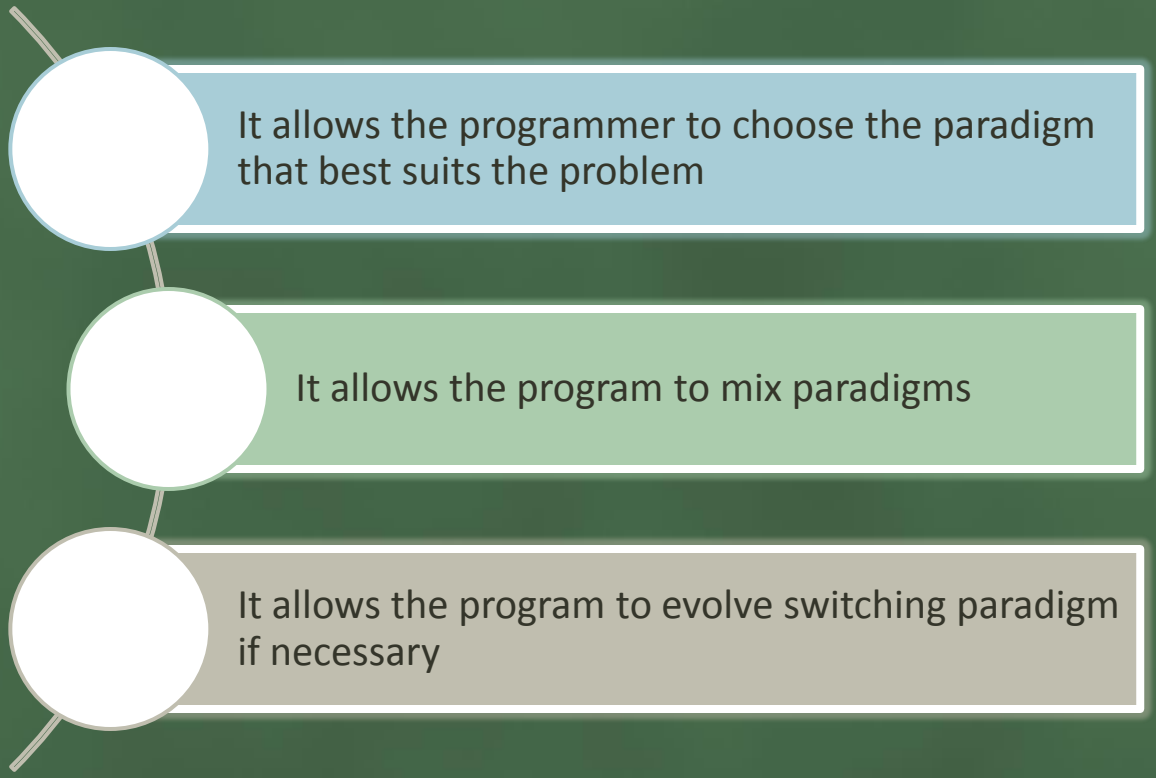
Before diving deep into the concept of Object Oriented Programming, let's talk a little about all the programming paradigms which exist in this world.



•Object Oriented Programming Basics

Programming Paradigms

Python is multiparadigm programming language



•Object Oriented Programming Basics

What is an Object?

A software item that contains variables and methods.

Object Oriented Design focuses on :-

Encapsulation

- dividing the code into a public interface, and a private implementation of that interface

Polymorphism

- the ability to overload standard operators so that they have appropriate behavior based on their context

Inheritance

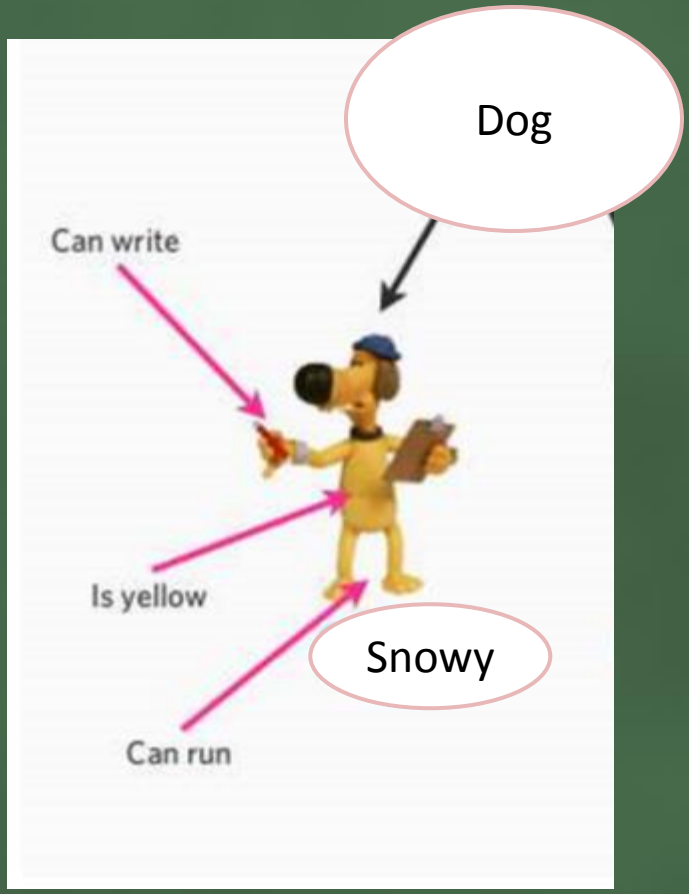
- the ability to create subclasses that contain specializations of their parents

•Object Oriented Programming Basics

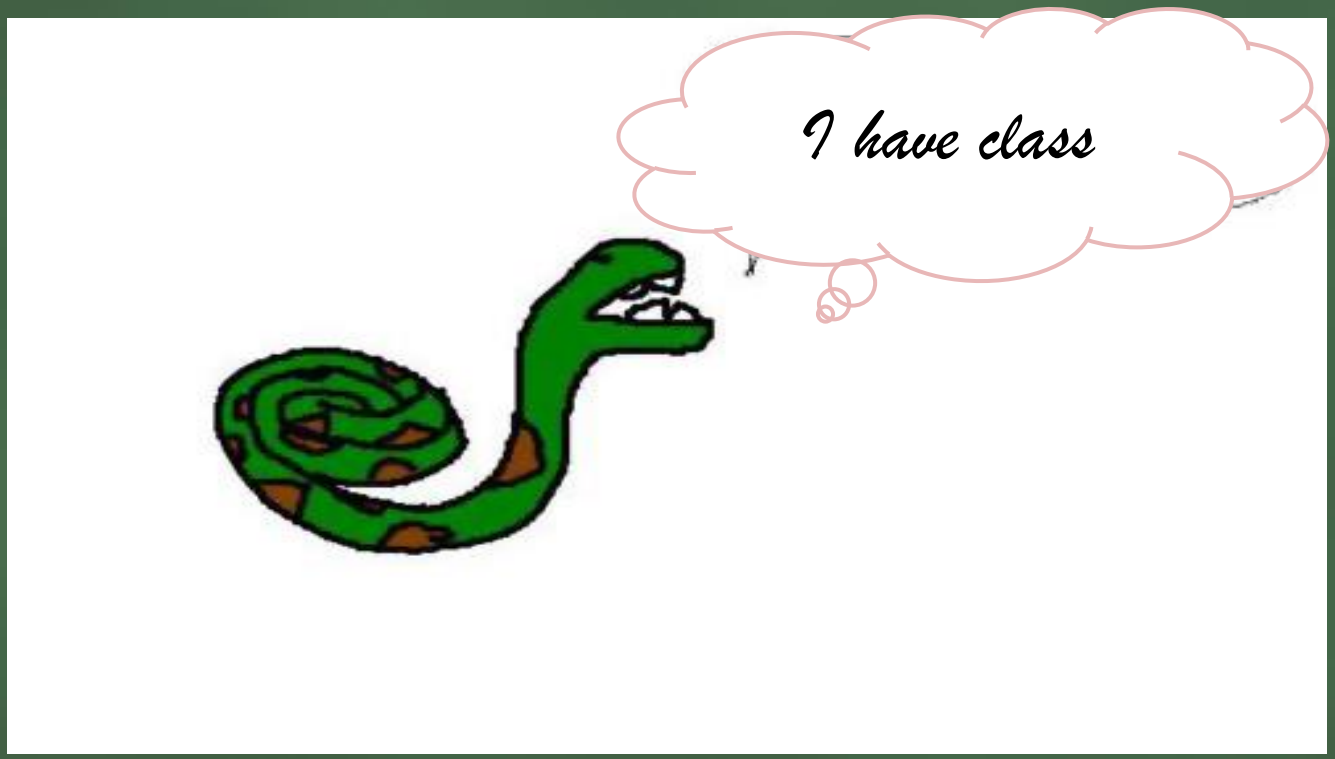
What is a Class?

Classes(in classic oo) define what is common for a whole class of objects, e.g.: “Snowy is a dog” can be translated to “The Snowy object is an instance of the dog class.” Define once how a dog works and then reuse it for all dogs. Classes correspond to variable types(they are type objects).

At the simplest level, classes are simply namespaces.



Object Oriented Programming in Python



•Object Oriented Programming in Python

Python Classes

- A class is a python object with several characteristics:
- You can call a class as it where a function and this call returns a new instance of the class
- A class has arbitrary named attributes that can be bound, unbound an referenced
- The class attributes can be descriptors (including functions) or normal data objects
- Class attributes bound to functions are also known as methods
- A method can have special python-defined meaning (they're named with two leading and trailing underscores)
- A class can inherit from other classes, meaning it delegates to other classes the look-up of attributes that are not found in the class itself

•Object Oriented Programming in Python

Python Classes in Detail (I)

- All classes are derived from object (new-style classes).

```
class Dog(object):
pass
```

- Python objects have data and function attributes (methods)

```
class Dog(object):
        def bark(self):
        print "Wuff!“

snowy = Dog()
snowy.bark() # first argument (self) is bound to this Dog instance
snowy.a = 1 # added attribute a to snowy
```

•Object Oriented Programming in Python

Python Classes in Detail (II)

- Always define your data attributes in __init__

```
class Dataset(object):
        def __init__(self):
                self.data = None

        def store_data(self, raw_data):
        ... # process the data
        self.data = processed_data
```

- Class attributes are shared across all instances.

```
class Platypus(Mammal):
        latin_name = "Ornithorhynchus anatinus"
```

•Object Oriented Programming in Python

Python Classes in Detail (III)

- Use super to call a method from a superclass.

```
class Dataset(object):
        def __init__(self, data=None):
                self.data = data

class MRIDataset(Dataset):
        def __init__(self, data=None, parameters=None):
                # here has the same effect as calling
                # Dataset.__init__(self)
                super(MRIDataset, self).__init__(data)
                self.parameters = parameters
mri_data = MRIDataset(data=[1,2,3])
```

•Object Oriented Programming in Python

Python Classes in Detail (IV)

- Special methods start and end with two underscores and customize standard Python behavior (e.g. operator overloading).

```
class My2Vector(object):
        def __init__(self, x, y):
                self.x = x
                self.y = y
        def __add__(self, other):
                return My2Vector(self.x+other.x, self.y+other.y)
v1 = My2Vector(1, 2)
v2 = My2Vector(3, 2)
v3 = v1 + v2
```

•Object Oriented Programming in Python

Python Classes in Detail (V)

- Properties allow you to add behavior to data attributes:

```
class My2Vector(object):
        def __init__(self, x, y):
                self._x = x
                self._y = y
        def get_x(self):
                return self._x
        def set_x(self, x):
                self._x = x
        x = property(get_x, set_x)
# define getter using decorator syntax
@property
        def y(self):
                return self._y
v1 = My2Vector(1, 2)
x = v1.x # use the getter
v1.x = 4 # use the setter
x = v1.y # use the getter
```

•Object Oriented Programming in Python

Python Example (I)

```
import random

class Die(object): # derive from object for new style classes
"""Simulate a generic die."""“

        def __init__(self, sides=6):
                """Initialize and roll the die.
                sides -- Number of faces, with values starting at one
                (default is 6).
                """
                self._sides = sides # leading underscore signals private
                self._value = None # value from last roll
                self.roll()

        def roll(self):
                """Roll the die and return the result."""
                self._value = 1 + random.randrange(self._sides)
                return self._value
```

•Object Oriented Programming in Python

Python Example (II)

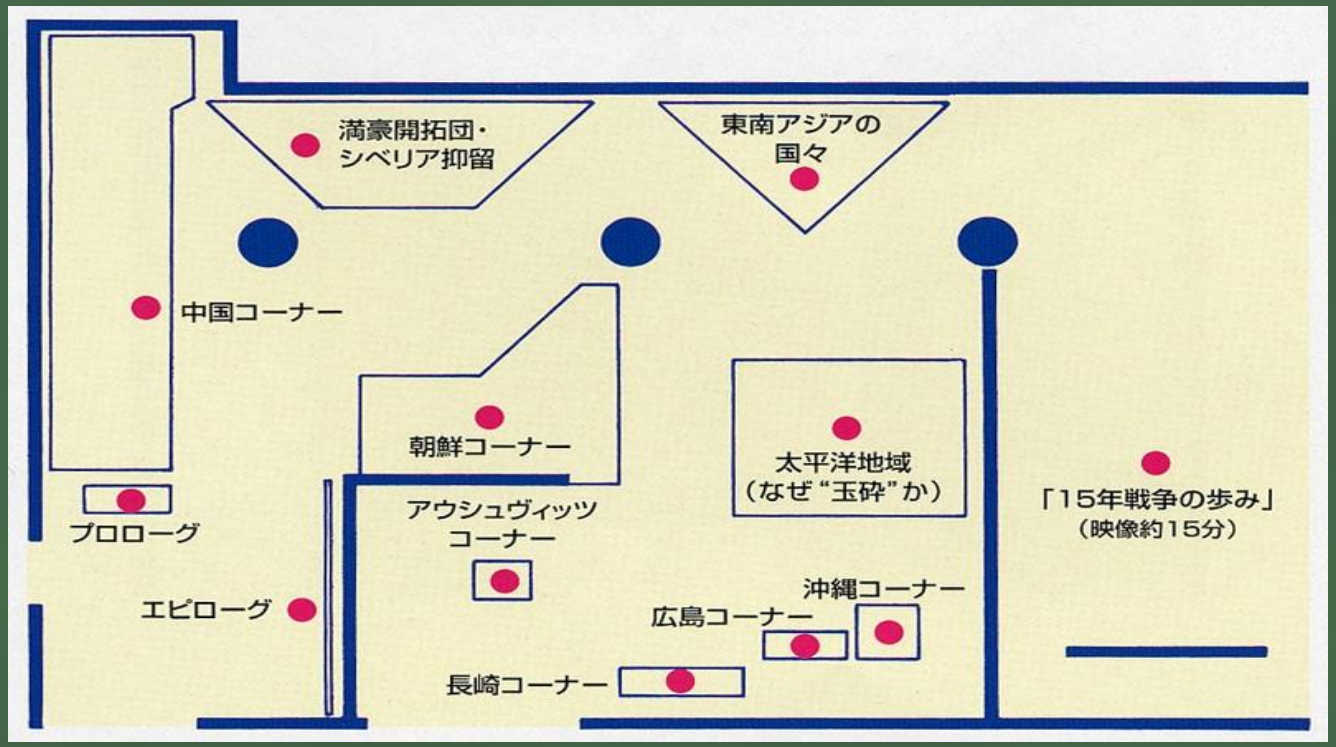
```
        def __str__(self):
                """Return string with a nice description of the die state."""
                return "Die with %d sides, current value is %d." %
                (self._sides, self._value)

class WinnerDie(Die):
        """Special die class that is more likely to return a 1."""
        def roll(self):
                """Roll the die and return the result."""
                super(WinnerDie, self).roll() # use super instead of
                Die.roll(self)

                if self._value == 1:
                        return self._value
                else:
                        return super(WinnerDie, self).roll()
```

•Object Oriented Programming in Python

Python Example (III)

```
>>> die = Die()
>>> die._sides # we should not access this, but nobody will stop us
6
>>> die.roll
<bound method Die.roll of <dice.Die object at 0x03AE3F70>>
>>> for _ in range(10):
...         print die.roll()
2 2 6 5 2 1 2 6 3 2

>>> print die # this calls __str__
Die with 6 sides, current value is 2.
>>> winner_die = dice.WinnerDie()
>>> for _ in range(10):
...         print winner_die.roll(),
2 2 1 1 4 2 1 5 5 1
>>>
```

Design Patterns & Python



•Design Patterns & Python

What is a Design Pattern?

Design Patterns are concrete solutions for reoccurring problems.

They satisfy the design principles and can be used to understand and illustrate them.

They provide a NAME to communicate effectively with other programmers.

Iterator Pattern

- The essence of the Iterator Factory method Pattern is to "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.".

Decorator Pattern

- The decorator pattern is a design pattern that allows behavior to be added to an existing object dynamically.

Strategy Pattern

- The strategy pattern (also known as the policy pattern) is a particular software design pattern, whereby algorithms behavior can be selected at runtime.

Adapter Pattern

- The adapter pattern is a design pattern that translates one interface for a class into a compatible interface

Iterator Pattern

•Iterator Pattern

Problem

- How would you iterate elements from a collection?

```
>>> my_collection = ['a', 'b', 'c']
>>> for i in range(len(my_collection)):
...         print my_collection[i],
a b c
```

- But what if my_collection does not support indexing?

```
>>> my_collection = {'a': 1, 'b': 2, 'c': 3}
>>> for i in range(len(my_collection)):
...         print my_collection[i],
# What will happen here?
```

- This violates one of the design principles!

•Iterator Pattern

Description

- store the elements in a collection (iterable)
- manage the iteration over the elements by means of an iterator
- object which keeps track of the elements which were already delivered
- iterator has a next() method that returns an item from the
- collection. When all items have been returned it raises a
- Stop Iteration exception.
- iterable provides an __iter__() method, which returns an iterator
- object.

•Iterator Pattern

Example (I)

```
class MyIterable(object):
"""Example iterable that wraps a sequence."""
        def __init__(self, items):
                """Store the provided sequence of items."""
                self.items = items

        def __iter__(self):
                return MyIterator(self)

class MyIterator(object):
        """Example iterator that is used by MyIterable."""
        def __init__(self, my_iterable):
                """Initialize the iterator.
                my_iterable -- Instance of MyIterable.
        """
                self._my_iterable = my_iterable
                self._position = 0
```

•Iterator Pattern

Example (II)

```
def next(self):
        if self._position < len(self._my_iterable.items):
                value = self._my_iterable.items[self._position]
                self._position += 1
                return value
        else:
                raise StopIteration()

# in Python iterators also support iter by returning self
def __iter__(self):
        return self
```

•Iterator Pattern

Example (III)

- First, lets perform the iteration manually:

```
iterable = MyIterable([1,2,3])
iterator = iter(iterable) # or use iterable.__iter__()
try:
while True:
item = iterator.next()
print item
except StopIteration:
pass
print "Iteration done."
```

- A more elegant solution is to use the Python for-loop:

```
for item in iterable:
print item
print "Iteration done."
```

- In fact Python lists are already iterables:

```
for item in [1,2,3]:
        print item
```

Decorator Pattern

•Decorator Pattern

Problem (I)

```
class Beverage(object):

        # imagine some attributes like temperature, amount left,..
        .
        def get_description(self):
                return "beverage“

        def get_cost(self):
                return 0.00

class Coffee(Beverage):

        def get_description(self):
                return "normal coffee"
        def get_cost(self):
                return 3.00
class Tee(Beverage):

        def get_description(self):
                return "tee"
        def get_cost(self):
                return 2.50
```

•Decorator Pattern

Problem (II)

```
class CoffeeWithMilk(Coffee):

        def get_description(self):
                return super(CoffeeWithMilk, self).get_description() + ", with milk“

        def get_cost(self):
                return super(CoffeeWithMilk, self).get_cost() + 0.30

class CoffeeWithMilkAndSugar(CoffeeWithMilk):

# And so on, what a mess!
```

•Decorator Pattern

Description

We have the following requirements:

- adding new ingredients like soy milk should be easy and work with all beverages,
- anybody should be able to add new custom ingredients without touching the original code (open-closed principle),
- there should be no limit to the number of ingredients.

•Decorator Pattern

Solution

```
class Beverage(object):

        def get_description(self):
                return "beverage“
        def get_cost(self):
                return 0.00

class Coffee(Beverage):
#[...]

class BeverageDecorator(Beverage):

        def __init__(self, beverage):
                super(BeverageDecorator, self).__init__() # not really needed here
                self.beverage = beverage

class Milk(BeverageDecorator):
        def get_description(self):
                #[...]
        def get_cost(self):
                #[...]
                coffee_with_milk = Milk(Coffee())
```

Strategy Pattern

•Strategy Pattern

Problem

```
class Duck(object):

        def __init__(self):
                # for simplicity this example class is stateless

        def quack(self):
                print "Quack!“

        def display(self):
                print "Boring looking duck.“

        def take_off(self):
                print "I'm running fast, flapping with my wings.“

        def fly_to(self, destination):
                print "Now flying to %s." % destination

        def land(self):
                print "Slowing down, extending legs, touch down."
```

•Strategy Pattern

Problem (I)

```
class RedheadDuck(Duck):

        def display(self):
                print "Duck with a read head.“

class RubberDuck(Duck):

        def quack(self):
                print "Squeak!“

        def display(self):
                print "Small yellow rubber duck."
```

- Oh man! The RubberDuck is able to fly!
- Looks like we have to override all the flying related methods.
- But if we want to introduce a DecoyDuck as well we will have to override all three methods again in the same way (DRY).
- And what if a normal duck suffers a broken wing?
- **Idea:** Create a FlyingBehavior class which can be plugged into theDuck class.

•Strategy Pattern

Solution (I)

```
class FlyingBehavior(object):
        """Default flying behavior."""
        def take_off(self):
                print "I'm running fast, flapping with my wings."
        def fly_to(self, destination):
                print "Now flying to %s." % destination
        def land(self):
                print "Slowing down, extending legs, touch down.“

class Duck(object):
        def __init__(self):
                self.flying_behavior = FlyingBehavior()
        def quack(self):
                print "Quack!"
        def display(self):
                print "Boring looking duck."
        def take_off(self):
                self.flying_behavior.take_off()
        def fly_to(self, destination):
                self.flying_behavior.fly_to(destination)
        def land(self):
                self.flying_behavior.land()
```

•Strategy Pattern

Solution (II)

```
class NonFlyingBehavior(FlyingBehavior):
        """FlyingBehavior for ducks that are unable to fly."""
        def take_off(self):
                print "It's not working :-("
        def fly_to(self, destination):
                raise Exception("I'm not flying anywhere.")
        def land(self):
                print "That won't be necessary.“

class RubberDuck(Duck):
        def __init__(self):
                self.flying_behavior = NonFlyingBehavior()
        def quack(self):
                print "Squeak!"
        def display(self):
                print "Small yellow rubber duck.“

class DecoyDuck(Duck):
        def __init__(self):
                self.flying_behavior = NonFlyingBehavior()
        def quack(self):
                print ""
        def display(self):
                print "Looks almost like a real duck."
```

Adapter Pattern

•Adapter Pattern

Problem

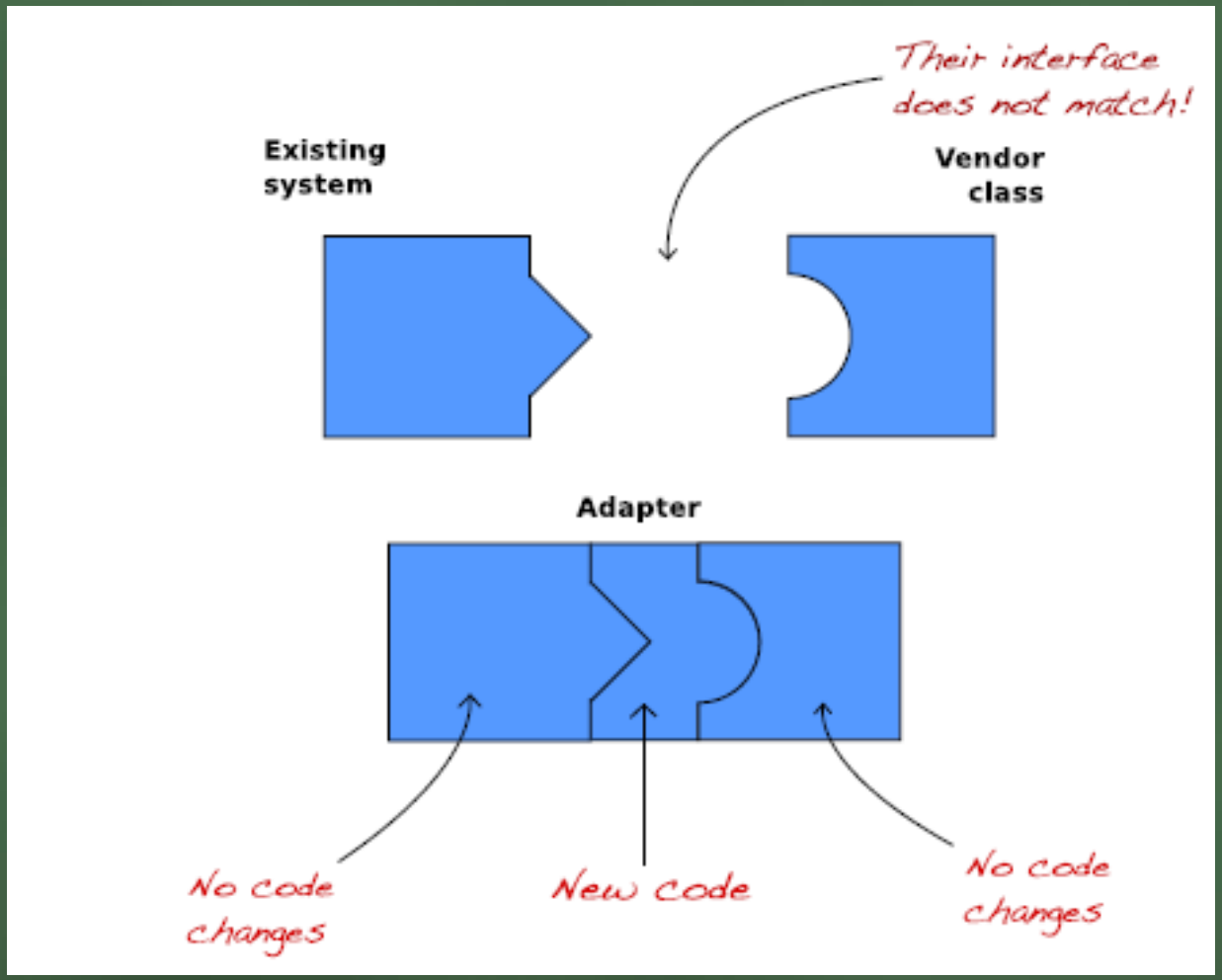
- Lets say we obtained the following class from our collaborator:

```
class Turkey(object):

        def fly_to(self):
                print "I believe I can fly...“

        def gobble(self, n):
                print "gobble " * n
```

How to integrate it with our Duck Simulator: turkeys can fly and gobble but they can not quack!

•Adapter Pattern

Description



•Adapter Pattern

Solution

```
class TurkeyAdapter(object):
        def __init__(self, turkey):
                self.turkey = turkey
                self.fly_to = turkey.fly_to #delegate to native Turkey method
                self.gobble_count = 3
        def quack(self): #adapt gobble to quack
                self.turkey.gobble(self.gobble_count)

>>> turkey = Turkey()
>>> turkeyduck = TurkeyAdapter(turkey)
>>> turkeyduck.fly_to()
I believe I can fly...
>>> turkeyduck.quack()
gobble gobble gobble
```

Adapter Pattern applies several good design principles:

- uses composition to wrap the adaptee (Turkey) with an altered interface,
- binds the client to an interface not to an implementation

More About Python

More About Python

Object models

Since Python2.2 there co-exist two slightly dierent object models in the language

Old-style (classic) classes : This is the model existing prior to Python2.2

New-style classes : This is the preferred model for new code

Old Style

```
>>> class A: pass
>>> class B: pass
>>> a, b = A(), B()
>>> type(a) == type(b)
True
>>> type(a)
<type 'instance'>
```

New Style

```
>>> class A(object): pass
>>> class B(object): pass
>>> a, b = A(), B()
>>> type(a) == type(b)
False
>>> type(a)
<class ' main .A'>
```

More About Python

New-style classes

- Defined in the type and class unification effort in python2.2
- (Introduced without breaking backwards compatibility)
- Simpler, more regular and more powerful
  - Built-in types (e.g. dict) can be subclassed
  - Properties: attributes managed by get/set methods
  - Static and class methods (via descriptor API)
  - Cooperative classes (sane multiple inheritance)
  - Meta-class programming
- It will be the default (and unique) in the future
- Documents:
  - Unifying types and classes in Python 2.2
  - PEP-252: Making types look more like classes
  - PEP-253: Subtyping built-in types
-

More About Python

The class statement

```
class classname(base-classes):
statement(s)
```

- classname is a variable that gets (re)bound to the class object after the class statement finishes executing
- base-classes is a comma separated series of expressions whose values must be classes
  - if it does not exists, the created class is old-style
  - if all base-classes are old-style, the created class is old-style
  - otherwise it is a new-style class1
  - since every type subclasses built-in object, we can use object to
  - mark a class as new-style when no true bases exist
- The statements (a.k.a. the class body) dene the set of class attributes which will be shared by all instances of the class

More About Python

Class-private attributes

- When a statement in the body (or in a method in the body) uses an identifier starting with two underscores (but not ending with them) such as __private, the Python compiler changes it to _classname__private
- This lets classes to use private names reducing the risk of accidentally duplicating names used elsewhere
- By convention all identifiers starting with a single underscore are
- meant to be private in the scope that binds them

```
>>> class C5(object):
... private = 23
>>> print C5.__private
AttributeError: class A has no attribute ' private'
>>> print C5. C5 private
23
```

More About Python

Descriptors

- A descriptor is any new-style object whose class supplies a special method named __get__
- Descriptors that are class attributes control the semantics of accessing and setting attributes on instances of that class
- If a descriptor's class also supplies method __set__ then it is called an overriding descriptor (a.k.a. data descriptor)
- If not, it is called non-overriding (a.k.a. non-data) descriptor
- Function objects (and methods) are non-overriding descriptors
- Descriptors are the mechanism behind properties, methods, static methods, class methods, and super (cooperative super-classes)
- The descriptor protocol also contains method __delete__ for unbinding attributes but it is seldom used

Thank You