

Object-Oriented Programming in Python

(Taken and Adapted from the course
notes of Dr. Greene of UCD School of
Computer Science and Informatics,
Dublin)

Classes and Objects

Object-Oriented Programming (OOP): A programming paradigm that involves designing programs around concepts represented as "objects"

- Python supports OOP through the provision of classes.
- Terminology
 - Class: A collection of functions and attributes, attached to a specific name, which represents an abstract concept.
 - Attribute: A named piece of data (i.e. variable associated with a class.
 - Object: A single concrete instance generated from a class

Instances of Classes

Classes can be viewed as factories or templates for generating new object instances.

Each object instance takes on the properties of the class from which it was created.

Instances of Classes

Abstract
concept



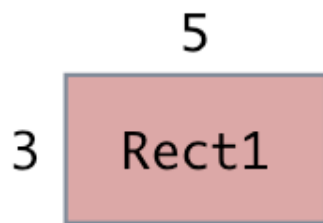
Attributes

- width
- height
- colour

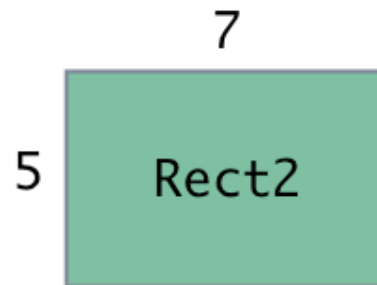
Functions

- area()

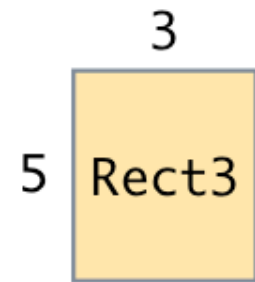
Multiple
concrete
instances



width=5
height=3
colour=(200,0,0)
area() -> 15



width=7
height=5
colour=(0,200,0)
area() -> 35



width=3
height=5
colour=(255,200,0)
area() -> 15

Creating Classes

Defining a class in Python is done using the class keyword, followed by an indented block with the class contents.

Class data
attributes



```
class <Classname>:
    data1 = value1
    ...
    dataM = valueM

    def <function1>(self, arg1, ..., argK):
        <block>
    ...
    def <functionN>(self, arg1, ..., argK):
        <block>
```

Class
member
functions



Defining Functions in Classes

- A class definition block can include multiple functions.
- These represent the functionality or behaviors that are associated with the class.

```
>>> class Maths:  
...     def subtract(self,i,j):  
...         return i-j  
...  
...     def add(self,x,y):  
...         return x+y
```

Argument (self) refers to the object itself

Calling Functions in Classes

- Using Class Functions from Outside a Class
Functions are referenced by using the dot syntax:
<objectName>.<methodName>()

```
>>> m = Maths()  
>>> m.subtract(10,5)  
5  
>>> m.add(6,7)  
13
```

← No need to
specify value for
self, Python does
this automatically

Calling Functions in Classes


- Using Class Functions from Inside a Class

When referring to functions from within a class, we must always prefix the function name with self

(e.g. self.subtract())

```
>>> class Maths:
...     def subtract(self,i,j):
...         return i-j
...
...     def testsub(self):
...         print self.subtract(8,4)
```

Tell Python to use function
associated with this object



Attributes

Class attribute
defined at top of
class

```
>>> class Person:
...     company = "ucd"
...
...     def __init__(self):
...         self.age = 23
```

Instance attribute
defined inside a class
function.
The `self` prefix is
always required.

```
>>> p1 = Person()
>>> p2 = Person()
>>> p1.age = 35
>>> print p2.age
23
```

Change to instance attribute `age`
affects only the associated
instance (p2)

```
>>> p1 = Person()
>>> p2 = Person()
>>> p1.company = "ibm"
>>> print p2.company
'ibm'
```


Change to class attribute `company`
affects all instances (p1 and p2)

Constructor

- When an instance of a class is created, the class constructor function is automatically called.
- The constructor is always named `__init__()`
- It contains code for initializing a new instance of the class to a specific initial state (e.g. setting instance attribute values).

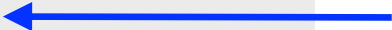
```
>>> class Person:
...     def __init__( self, s ):
...         self.name = s
...
...     def hello( self ):
...         print "Hello", self.name
```

Constructor function taking
initial value for instance
attribute name



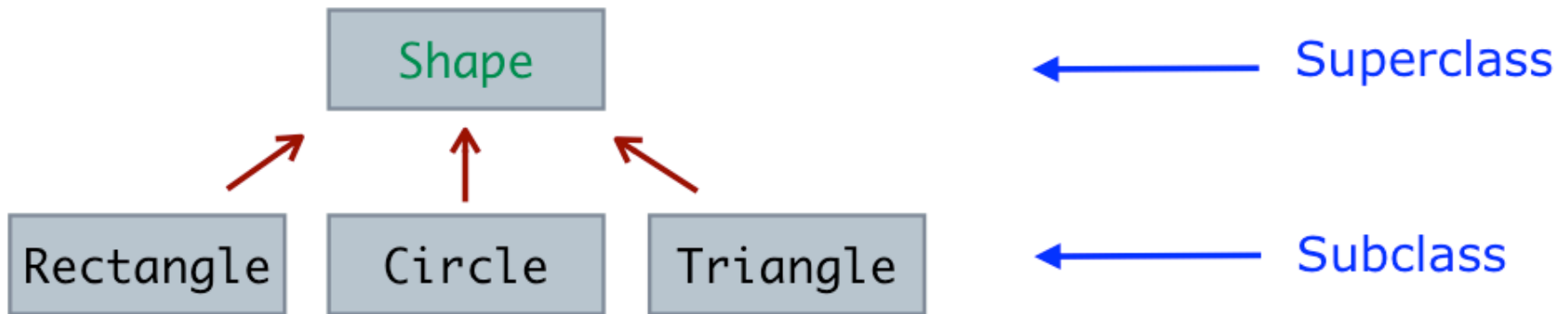
```
>>> t = Person("John")
>>> t.hello()
Hello John
```

Calls `__init__()`
on Person



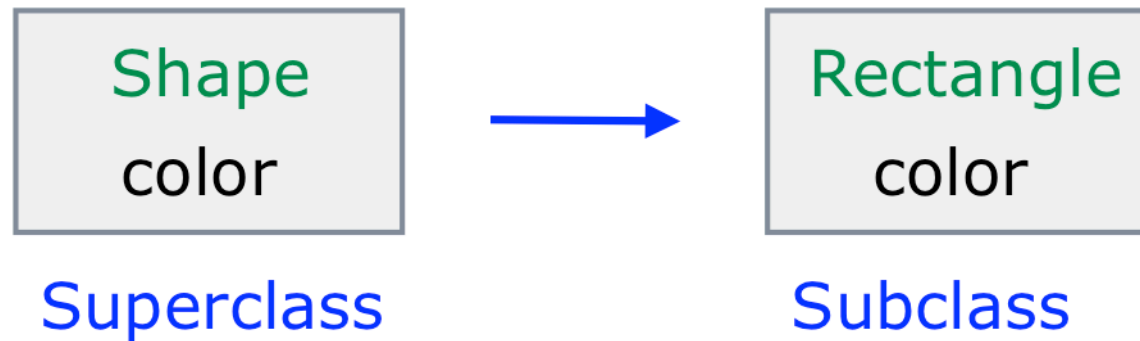
Inheritance

Class inheritance is designed to model relationships of the type "x is a y" (e.g. "a triangle is a shape")



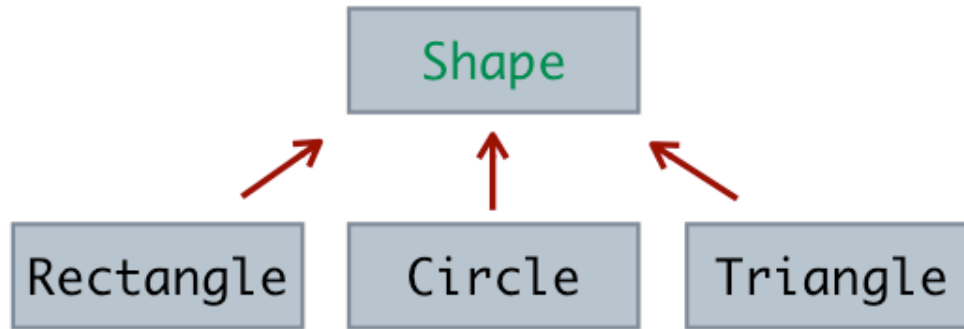
Inheritance

The functions and attributes of a superclass are inherited by a subclass.



An inherited class can override, modify or augment the functions and attributes of its parent class.

Creating Subclasses



Superclass →

```
>>> class Shape:
...     pass
```

Subclass →

```
>>> class Rectangle(Shape):
...     def area(self):
...         return self.width*self.height
```

Name of superclass




Simple superclass

```
>>> class Shape:
...     def __init__( self ):
...         self.color = (0,0,0)
```

Simple subclass
inheriting from
Shape

```
>>> class Rectangle(Shape):
...     def __init__( self, w, h ):
...         Shape.__init__( self )
...         self.width = w
...         self.height = h
...
...     def area( self ):
...         return self.width*self.height
```

Need to call
constructor
function in
superclass



```
>>> r1 = Rectangle( 10, 5 )
>>> print r1.width
10
>>> print r1.height
5
>>> print r1.area()
50
>>> print r1.color
(0, 0, 0)
```

Construct
object instance



Inherited
attribute



Overriding

When inheriting from a class, we can alter the behavior of the original superclass by "overriding" functions (i.e. declaring functions in the subclass with the same name).

Functions in a subclass take precedence over functions in a superclass.

Overriding

```
class Counter:  
    def __init__(self):  
        self.value = 0  
  
    def increment(self):  
        self.value += 1
```

Overriding

```
class CustomCounter(Counter):  
    def __init__(self, size):  
        Counter.__init__(self)  
        self.stepsize = size  
  
    def increment(self):  
        self.value += self.stepsize
```

```
>>> cc = CustomCounter(4)  
>>> cc.increment()  
>>> cc.print_current()  
Current value is 4
```

Calls increment()
on CustomCounter
not Counter

Composition

Classes can be built from other smaller classes, allowing us to model relationships of the type "**x has a y**" (e.g. a department has students).


```
class Department:
    def __init__( self ):
        self.students = []

    def enroll( self, student ):
        self.students.append(student)
```

```
class Student:
    def __init__( self, last, first ):
        self.lastname = last
        self.firstname = first
```

```
>>> compsci = Department()
>>> compsci.enroll( Student( "Smith", "John" ) )
>>> compsci.enroll( Student( "Murphy", "Alice" ) )
```

Create Student
instances and add
to Department
instance



```
>>> for s in compsci.students:
...     print "%s %s" % (s.firstname,s.lastname)
...
John Smith
Alice Murphy
```

Polymorphism

Two objects of different classes but supporting the same set of functions or attributes can be treated identically.

The implementations may differ internally, but the outward "appearance" is the same.

Polymorphism

Two different classes that contain the function `area()`

```
class Rectangle(Shape):
    def __init__(self, w, h):
        Shape.__init__(self)
        self.width = w
        self.height = h

    def area(self):
        return self.width*self.height
```

```
class Circle(Shape):
    def __init__(self, rad):
        Shape.__init__(self)
        self.radius = rad

    def area(self):
        return math.pi*(self.radius**2)
```

Instances of the two classes can be treated identically...

Result of `area()` in Rectangle →

Result of `area()` in Circle →

```
>>> l = []
>>> l.append( Rectangle(4,5) )
>>> l.append( Circle(3) )
>>> for someshape in l:
...     print someshape.area()
...
20
28.2743338823
```