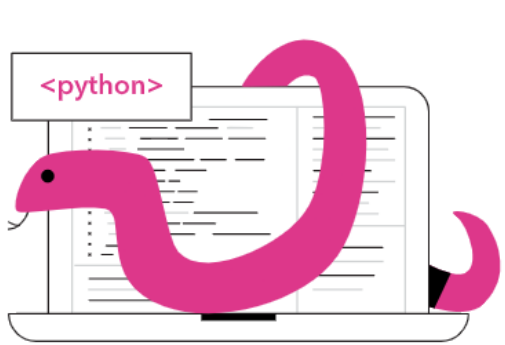
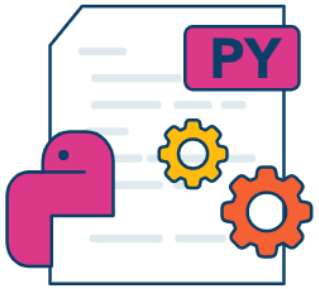
Classes and Objects





DR. VEENA R S
ASSOCIATE PROFESSOR
DEPARTMENT OF ISE
DSATM, BENGALURU

Class and Objects

- A user-defined prototype for an object that defines a set of attributes that characterize any object of the class.
- The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Programmer-defined types

- Define a new type.
- We will create a type called Point that represents a point in two-dimensional space.
- Points are often written in parentheses with a comma separating the coordinates.
- For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

- There are several ways we might represent points in Python:
  1. We could store the coordinates separately in two variables, x and y.
  2. We could store the coordinates as elements in a list or tuple.
  3. We could create a new type to represent points as objects.

- A programmer-defined type is also called a **class**.

- A class definition looks like this:

```
class Point:
"""Represents a point in 2-D space."“”
```

- The header indicates that the new class is called **Point**. The body is a docstring that explains what the class is for.
- You can define variables and methods inside a class definition

- Defining a class named Point creates a **class object**.

```
>>> Point
<class '__main__.Point'>
```

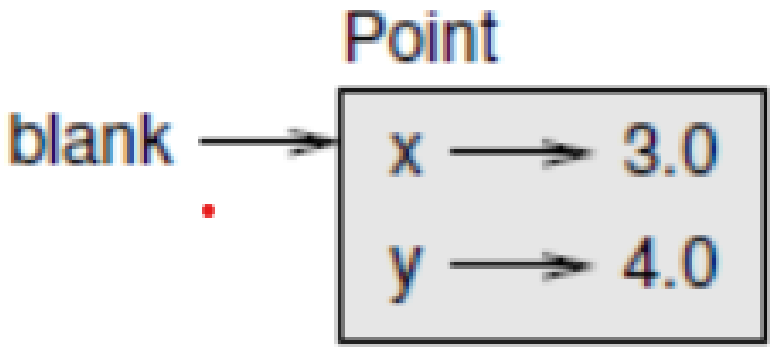
- Because Point is defined at the top level, its “full name” is __main__.Point.

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

Attributes

>>> blank.x = 3.0

>>> blank.y = 4.0



Print x and y

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

```
>>> print_point(blank)
(3.0, 4.0)
```

Rectangles

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

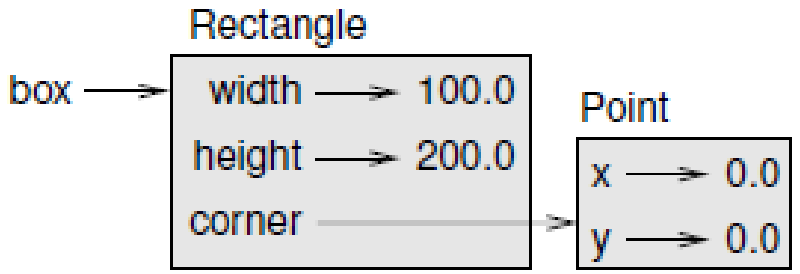


Figure 15.2: Object diagram.

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

Instances as return values

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

Objects are mutable

- You can change the state of an object by making an assignment to one of its attributes.

box.width = box.width + 50

box.height = box.height + 100

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Copying

- Copying an object is often an alternative to aliasing.
- The copy module contains a function called copy that can duplicate any object

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

p1 and p2 contain the same data, but they are not the same Point.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
```

Classes and functions

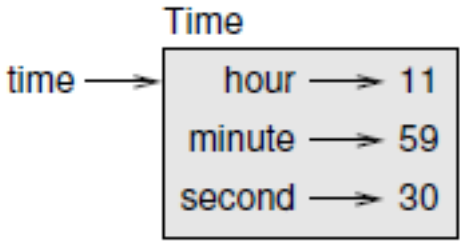


Figure 16.1: Object diagram.

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

We can create a new `Time` object and assign attributes for hours, minutes, and seconds:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

Classes and methods

Object-oriented features

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

Printing objects

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a `Time` object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

Syntax: Class Definition

```
class ClassName:
     # Statement
```

Syntax: Object Definition

```
obj = ClassName()
print(obj.atrr)
```

```
#Python Code to demonstrate classes and objects


class Rectangle:
    """
    attributes:l, b"""
    def display(self):
        print(self.l,self.b)
r=Rectangle()
r.l=int(input('Enter the length'))
r.b=int(input('Enter the beadth'))
r.display()
```

The init method

- The init method (short for “initialization”) is a special method that gets invoked when an object is instantiated.
- Its full name is __init__ (two underscore characters, followed by init, and then two more underscores).

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

If you provide one argument, it overrides `hour`:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

If you provide two arguments, they override `hour` and `minute`.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

The __str__ method

- Overrides print()

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you `print` an object, Python invokes the `str` method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

Class attributes

- In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits.

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

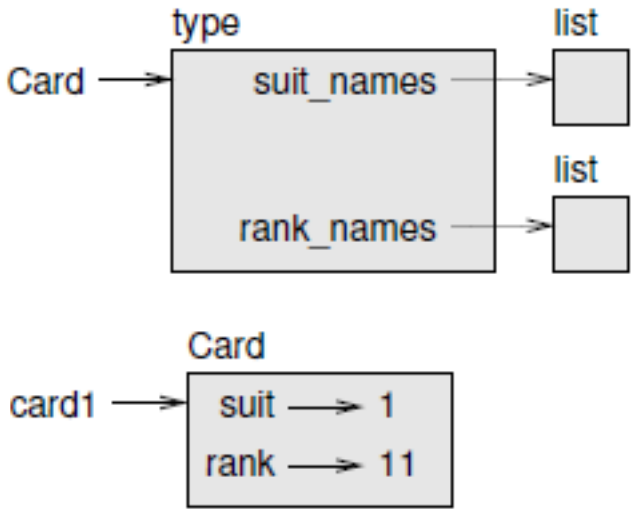
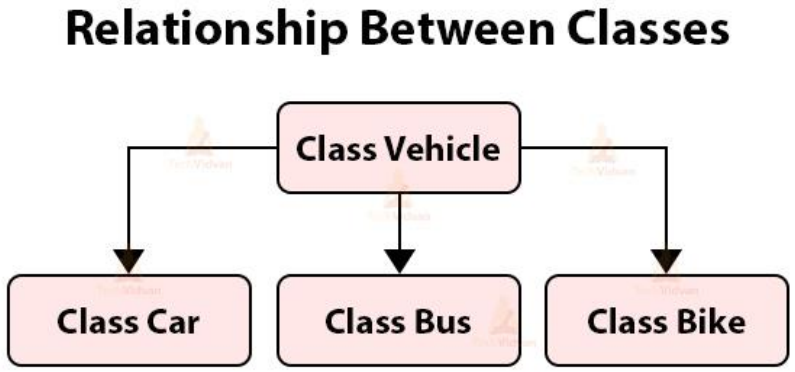


Figure 18.1: Object diagram.

Inheritance

- When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**.
- we want a class to represent a “hand”, that is, the cards held by one player.
- A hand is like a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.



Inheritance

- When a new class inherits from an existing one, the existing one is called the *parent* and the new class is called the *child*.
- There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks.
- The suits are Spades, Hearts, Diamonds, and Clubs.

Inheritance

- The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

- If we want to define a new object to represent a playing card, it is obvious what the attributes should be: rank and suit.

Inheritance

- One possibility is to use strings containing words like 'Spade' for suits and 'Queen' for ranks.
- One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.
- An alternative is to use integers to **encode** the ranks and suits.

$$
\begin{aligned}
\text{Spades} &\mapsto 3 \\
\text{Hearts} &\mapsto 2 \\
\text{Diamonds} &\mapsto 1 \\
\text{Clubs} &\mapsto 0
\end{aligned}
$$

Inheritance

The class definition for `Card` looks like this:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

As usual, the init method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a Card, you call `Card` with the suit and rank of the card you want.

```
queen_of_diamonds = Card(1, 12)
```

Inheritance

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that `Hand` inherits from `Deck`; that means we can use methods like `pop_card` and `add_card` for Hands as well as Decks.

Class diagrams

- A class diagram is a more abstract representation of the structure of a program.
- These diagrams represent a snapshot in the execution of a program, so they change as the program runs.
- There are several kinds of relationship between classes:
  - Objects in one class might contain references to objects in another class ---This kind of relationship is called HAS-A, as in, “a Rectangle has a Point.”
  - One class might inherit from another This relationship is called IS-A, as in, “a Hand is a kind of a Deck.”
  - Dependency- objects in one class take objects in the second class as parameters.

Class diagrams

- The standard arrowhead represents a HAS-A relationship; in this case a Deck has references to Card objects.
- The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.
- The star (*) near the arrowhead is a multiplicity; it indicates how many Cards a Deck has.

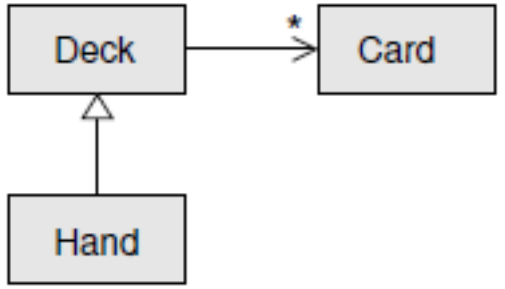


Figure 18.2: Class diagram.

Class diagrams

- The standard arrowhead represents a HAS-A relationship; in this case a Deck has references to Card objects.
- The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.
- The star (*) near the arrowhead is a multiplicity; it indicates how many Cards a Deck has.

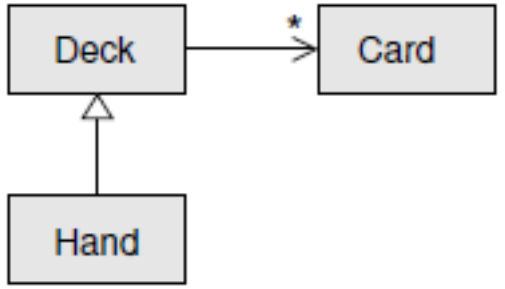


Figure 18.2: Class diagram.

```
# A Python program to demonstrate inheritance
class Person(object):

  # Constructor
  def __init__(self, name, id):
    self.name = name
    self.id = id

  # To check if this person is an employee
  def Display(self):
    print(self.name, self.id)


# Driver code
emp = Person("XYZ", 102) # An Object of Person
emp.Display()
```

```
class Emp(Person):

  def Print(self):
    print("Emp class called")

Emp_details = Emp(“ABC", 103)

# calling parent class function
Emp_details.Display()

# Calling child class function
Emp_details.Print()
```

7. a) By using the concept of inheritance write a python program to find the area of triangle, circle and rectangle.

```
import math
class Shape:
        def __init__(self):
                self.area = 0
                self.name = ""
        def showArea(self):
                print("The area of the", self.name, "is", self.area, "units")
```

```
class Circle(Shape):
                    def __init__(self,radius):
                                self.area = 0
                                self.name = "Circle"
                                self.radius = radius
                    def calcArea(self):
                                self.area = math.pi * self.radius * self.radius
class Rectangle(Shape):
                    def __init__(self,length,breadth):
                                self.area = 0
                                 self.name = "Rectangle"
                                self.length = length
                                self.breadth = breadth
                    def calcArea(self):
                                self.area = self.length * self.breadth
```

```
class Triangle(Shape):
                        def __init__(self,base,height):
                                self.area = 0
                         self.name = "Triangle"
                        self.base = base
                        self.height = height
                        def calcArea(self):
                                self.area = self.base * self.height / 2
c1 = Circle(5)
c1.calcArea()
c1.showArea()

r1 = Rectangle(5, 4)
r1.calcArea()
r1.showArea()

t1 = Triangle(3, 4)
t1.calcArea()
t1.showArea()
```

b. Write a python program by creating a class called Employee to store the details of Name, Employee_ID, Department and Salary, and implement a method to update salary of employees belonging to a given department.

```
class Employee:
        def_init_(self, name, emp_id, salary, department):
                self.name = name
                self.id = emp_id self.salary = salary
                self.department = department
        def update_salary(self, dept, increment):
                if self.department == dept:
                self.salary = self.salary + increment
        def print_employee_details(self):
                 print("\nName: ", self.name)
                print("ID: ", self.id)
                print("Salary: ", self.salary)
                print("Department: ", self.department)
                print("....")
```

```
Employee_list = []


Employee_list.append(Employee("ADAMS", "E7876", 50000, "ACCOUNTING"))
Employee_list.append(Employee("JONES", "E7499", 45000, "RESEARCH"))
Employee_list.append(Employee("MARTIN", "E7900", 50000, "SALES"))
Employee_list.append(Employee("SMITH", "E7698", 55000, "OPERATIONS"))
print("Original Employee Details:")
for obj in Employee_list:
        print(obj.print_employee_details())
obj.update_salary("SALES",5000) # Change the salary of employees belonging to a department


print("Updated Employee Details:")
for obj in Employee_list:
        obj.print_employee_details()
```