Object Oriented Programming

Learning Outcomes

Learning Outcomes

By the end of this section you will be able to:

- Explain the principle of object-oriented programming.
- Make use of classes, objects and methods.
- Explain the concept of inheritance in python and how classes and sub-classes relate.
- Appropriately use modules and packages when structuring code.

Introduction

Outline

- Object Orientation Programming
- Classes and Objects
  - Attributes
  - Methods
  - Static and Private methods
- Inheritance
- Modules
- Packages

Points

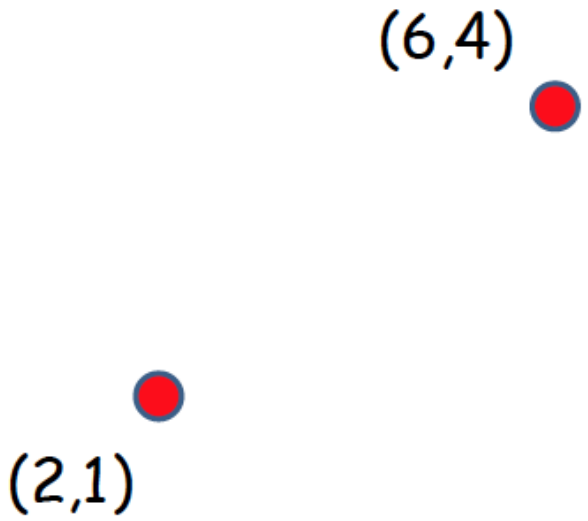
- We have used many of Python's built-in types; now we are going to define a new type.

- As an example, we will create a type called Point that represents a point in two-dimensional space.
  - In mathematical notation, 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:
  - We could store the coordinates separately in two variables, x and y.
  - We could store the coordinates as elements in a list or tuple.
  - We could create a new type to represent points as objects.

Points


```
>>> P = Point(2,1)
>>> Q = Point(6,4)
>>> d = P.Dist(Q)
>>> print d
5
```



Here, `Dist` is a method and `P.Dist(Q)` says "compute and return the distance from point P to point Q.

By having a `Point` class we can think at the "point level" instead of at the "xy level"

Let's define new types!

Recall that a type is a set of values and operations that can be performed on those values.

The four basic "built-in" types:

```
int, float, str, bool
```

Classes are a way to define new types.

By suitably defining a rectangle class, we could say something like

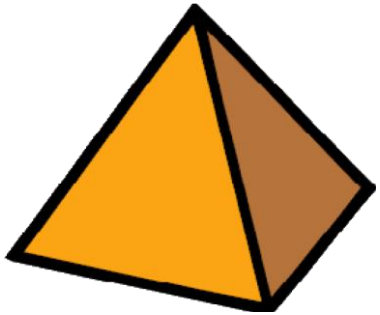
```
if R1.intersect(R2):
  print 'Rectangles R1 and R2 intersect'
```

Classes

- Python allows us to build our own types.
- These are called classes.
- When we have a class, we can create instances/objects of that class.
- Recall the difference between a type (str, int) and a value ('this is a string', 10).
  - This is analogous to the difference between a class and an object.

Classes and Objects

Class



- A class is like a blueprint or a concept.
- A pyramid is an example. We know what a pyramid is a concept.
- Actual pyramids like the pyramids of Giza, Mayan Pyramids and Aztec pyramids can be thought of as objects as they have physical manifestation; they are not just a concept
- Another example of a class is a car (which is a concept). Your car is an object.

Objects

The Object Oriented Paradigm

- Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures encapsulating data and functions together with their interactions – to design applications and computer programs.
- It was invented with the creation of the **Simula language in 1965**, and further developed in Smalltalk in the 1970s,
- It was not commonly used in mainstream software application development until the early 1990s.

OOP history

- Early 1980s Bjorn Stroustrup integrated object-oriented programming into the C language. The resulting language was called **C++** and it **became the first object- oriented language to be widely used commercially.**
- Early 1990s a group at Sun Microsystems led by James Gosling developed a simpler version of C++ called **Java** that was meant to be a programming language for video-on- demand applications.
- Java became one of the most popular languages
- **Many modern programming languages now support OOP including python.**

Fundamentals of OOP

- Every*thing* is an object that has specific characteristics, attributes, behavior. An actual car has the following characteristics: a color, max speed, number of doors… etc. In terms of behavior, it can accelerate, slow down, turn, break, etc.
- Objects have parts called *Members*
  - Attributes (Data) - (state: characteristics that can change) age, doors, engine…
  - Operations (Functions) – (abilities: things that can do) start, accelerate…
- Real world objects= State (properties) + abilities (behavior)
- Programming objects = data + functions.

Classes and Objects

Python is an *object-oriented* programming language

An *object* is a combination of variables (also called *attributes* or *instance variables* or *object variables*) and behaviors (i.e., functions, which are referred to as *methods* in the object context)

To create an object, you need to create a *class* using the keyword *class* as follows:

```
class Point:
    x = 3
    y = 4
```

These are attributes. You can have as many attributes as you want.

Creating Objects Out of Classes

After defining a class, you can create any number of objects out of it

```
p1 = Point()
print(p1.x)
p2 = Point()
print(p2.y)
p3 = Point()
print(p3.x)
```

But, all of the above points have the same coordinates!

How can we have point objects with different coordinates?

Defining the "Point" Class


```
class Point:
"""
Attributes:
x: float, the x-coordinate of a point
y: float, the y-coordinate of a point
"""
    def construct_point(x,y):
        Point.x = x
        Point.y = y
```

This is just Pseudo Code of what we need to do

A class provides a "blue print" for packaging data.
The data is stored in the attributes.

The Constructor


```
def __init__(self,x,y):
    """ Creates a Point object

    PreC: x and y are floats
    """
    self.x = x
    self.y = y
```

That's a double underscore: `__init__`

"`self`" is always the first argument for any method defined in a class.

The Constructor

The special method __init__ is called whenever you try and construct an instance of an object.

```
class Patient:
    def __init__(self,name,age):
        self.name = name
        self.age = age
```

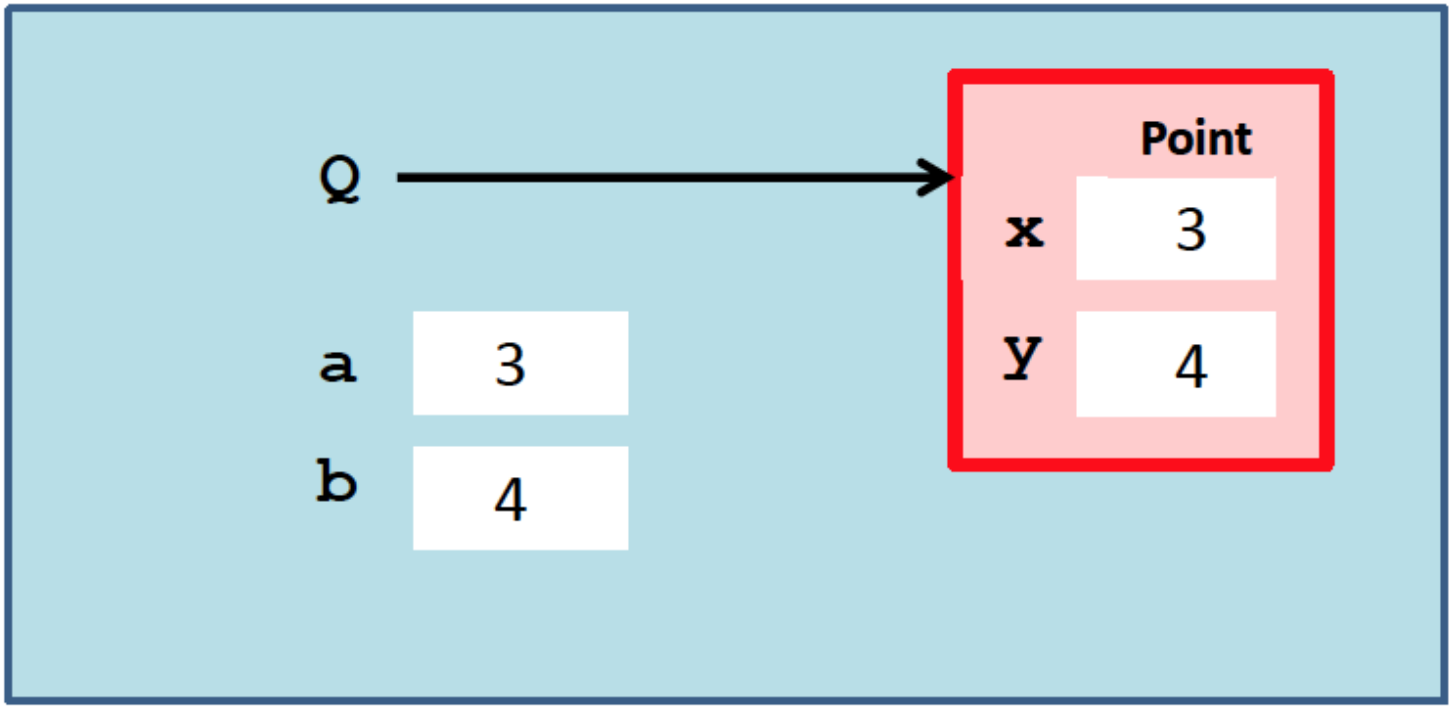
One can also just call the constructor without any values and set the attributes to default values. For example,

```
class Patient:
    def __init__(self):
        self.name = ''
        self.age = 0
```

Then we can construct a Patient object as follows:

x = Patient("Ahmed", 10)

Calling The Constructor!




```
>>> a = 3
>>> b = 4
>>> Q = Point(a,b)
```

This creates a Point object

Accessing the Attributes


```
>>> Q = Point(3,4)
>>> print Q
( 3.000, 4.000)
>>> Q.x = Q.x + 5
>>> print Q
( 8.000, 4.000)
```

Q.x is a variable and can "show up" in all the usual places, i.e., in an assignment statement.

Class variables

A *class variable* is a variable associated with a class, not an instance of a class (object), and is accessed by all instances of the class in order to keep track of some class-level information, such as how many instances of the class have been created at any point in time.

```
class Circle:
    pi = 3.14159
    def __init__(self,
radius):
        self.radius =
radius
    def area(self):
        return self.radius
* self.radius * Circle.pi
```

```
print(Circle.pi)
3.1415899999999999
Circle.pi = 4
print(Circle.pi)
>>> 4
Circle.pi = 3.14159
print(Circle.pi)
>>> 3.1415899999999999
```

Methods

- A crucial component of classes are methods.
  - You are already familiar with functions.

- Functions that define the behavior of a class are called methods.
  - i.e., Methods are functions that are defined inside a class definition.

Defining Classes and Methods

A crucial component of classes are methods. As we said before, they are very similar to functions in non-OOP languages.

```
class Circle:
    def __init__(self):
        self.radius = 1
    def area(self):
        return self.radius * self.radius * 3.1415
```

We use the **class** keyword to define a class. All properties and functions (methods) belonging to a class have to be idented under it..

This is a special function called the constructor. All class methods must have self passed as a parameter.

Another method in class Circle

Defining Objects and Calling methods


```
c = Circle()
c.radius = 3
print(c.area())
>> 28.27431
```

Here a new object c is defined

The radius property of object c is set to 3

Invoke c's area() method and print the result

Calculate the Distance!

We will design a method for the `Point` class that can be used to compute the distance between two points.

It will be used like this:

```
delta = P.Dist(Q)
```

Note the dot notation syntax for method Calls.

Calculate the Distance!


```
def __init__(self,x,y):
    self.x = x
    self.y = y

def Dist(self,other):
    """ Returns distance from self to other.
    PreC: other is a point
    """
    dx = self.x - other.x
    dy = self.y - other.y
    d = sqrt(dx**2+dy**2)
    return d
```

Now Let's Use our Dist Method

Let's create two point objects and compute the distance between them. This can be done two ways…

```
>>> P = Point(3,4)
>>> Q = Point(6,8)
>>> deltaPQ = P.Dist(Q)
>>> deltaQP = Q.Dist(P)
>>> print deltaPQ,deltaQP
5.0 5.0
```

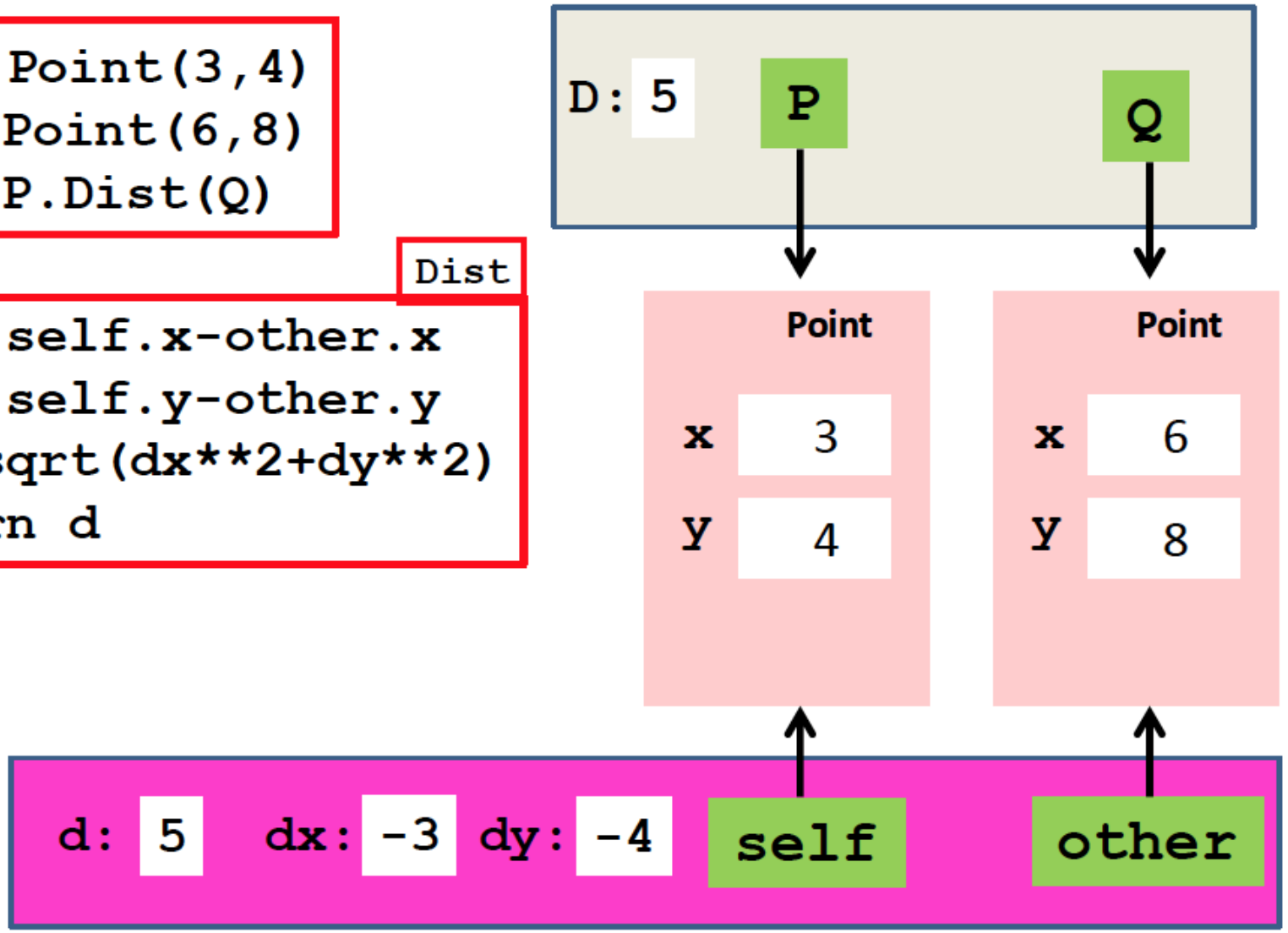
The usual "dot" notation for invoking a method

Dist Method Inside Out


```
P = Point(3,4)
Q = Point(6,8)
D = P.Dist(Q)
```

Dist

```
dx = self.x-other.x
dy = self.y-other.y
d = sqrt(dx**2+dy**2)
return d
```



Creating vs Using Classes

- Make a distinction between **creating** a class and using an **instance** of the class

- **Creating** the class involves
  - Defining the class **name**
  - Defining class **attributes**
  - For example, someone wrote code to implement a ”Circle” class

- **Using** the class involves
  - Creating new **instances** of **objects**
  - Doing **operations** on the instances
  - For example, c=Circle(), and c.radius = 3

Class Conventions

- Class names start with upper case letters.
- Class methods and instances start with lower case letters.
- Method definitions should have docstrings just like function definitions.
- Classes should have docstrings just like modules have docstrings that describe what the class does.

A List of Points

We would like to assemble a list whose elements are not numbers or strings, but references to objects.

For example, we have a hundred points in the plane and a length-100 list of points called `ListOfPoints`.

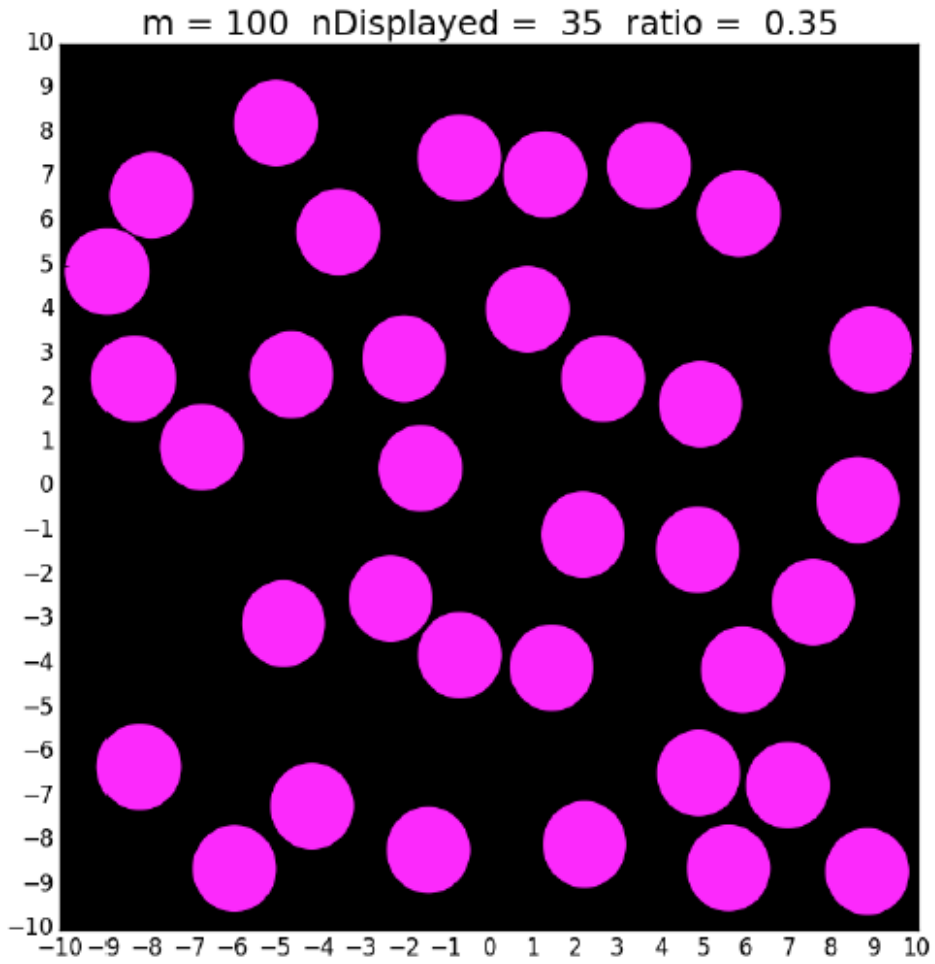
Let's compute the average distance to (0,0).

A List of Points


```
Origin = Point(0,0)
d = 0
for P in ListOfPoints:
    d += P.Dist(Origin)
N = len(ListOfPoints)
AveDist = d/N
```

A lot of familiar stuff: Running sums. A for-loop based on "in". The len function, Etc

Exercise: Disc Intersection



We have a 10-by-10 target

for k in range(100):

Generate a random disk D

Display D if it does not touch any of the previously displayed disks

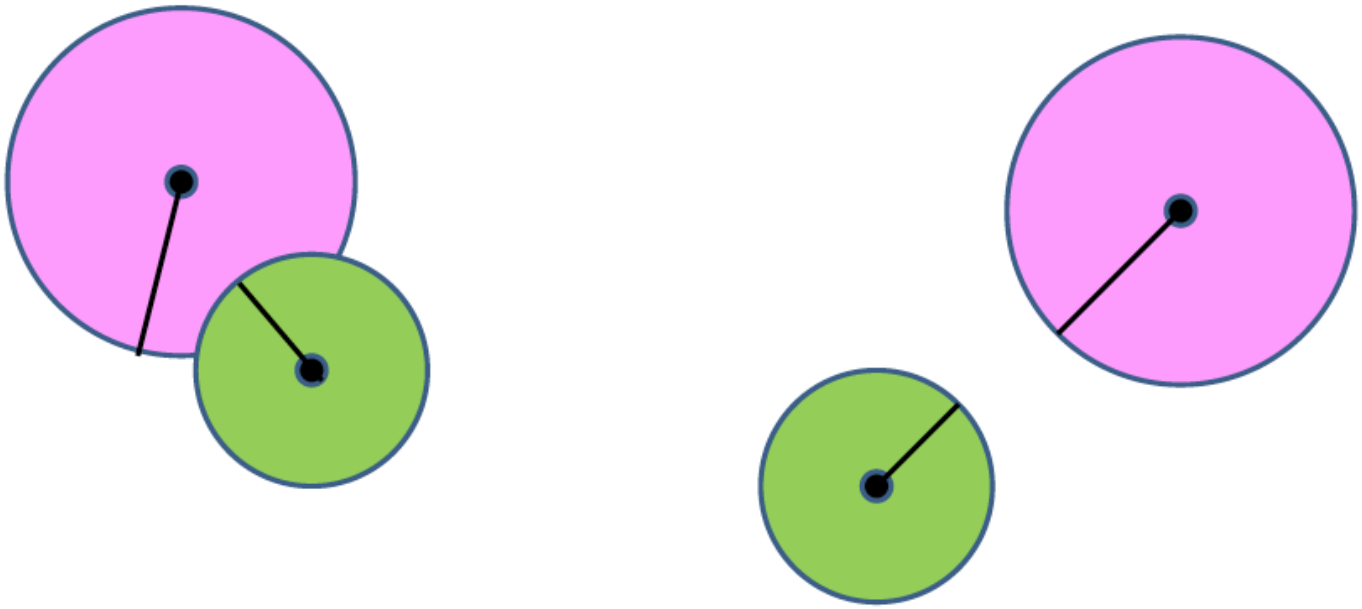
Assume all the disks have radius 1 and all inside the target.

First: Disk Constructor


```
class Disk:
    """
    Attributes:
        center: Point, the center of the disk
        radius: float, the radius of the disk
    """
    def __init__(self,P,r):
        """ Creates a Disk object with
        center P and radius r
        PreC: P is a Point,r is a pos float
        """
        self.center = P
        self.radius = r
```

Exercise: Disc Intersection

When Does a Pair of Disks Intersect?



Answer: When the distance between their centers is less than the sum of their radii.

The Method Intersects


```
def Intersects(self,other):
        """ Returns True if self and other
        intersect and False otherwise.
        PreC: self and other are Disk objects
        """
        # The center-to-center distance:
        c1 = self.center
        c2 = other.center
        d = c1.Dist(c2)
        # The sum of the two radii
        radiusSum = self.radius + other.radius
        TheyIntersect = (radiusSum >= d )
        return TheyIntersect
```

Now Let's Check Intersection


```
def outsideAll(D0,L):
    """ Returns True if D0 doesn't
    intersect any of the disks in L

    PreC: D0 is a Disk and L is a
    list of Disks
    """
    for D in L:
        if D.Intersects(D0):
            return False
    return True
```

Display Disks without Intersection

```
# The list of displayed disks…
m = 10
DiskList = []
for k in range(100):
    D = RandomDisk(m-1)
    if outsideAll(D,DiskList):
        # D does not intersect any
        # of the displayed disks
        ShowDisk(D,MAGENTA)
        DiskList.append(D)
nDisplayed = len(DiskList)
```

Starts out as the empty list

Display D and append it to the list of displayed disks

Getter and Setter Methods


```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

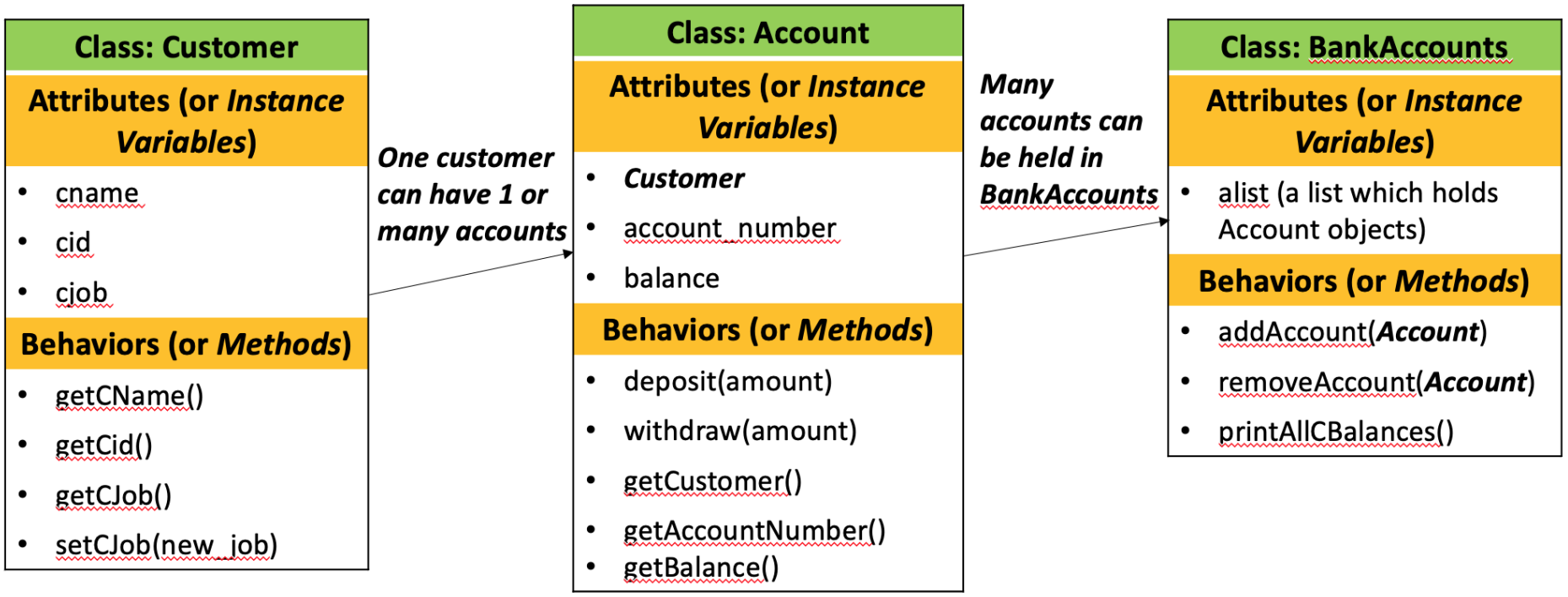
getter

setter

- **getters and setters** should be used outside of class to access data attributes

Example: A Simplified Bank Software

- Let us write a simplified *object-based* program for a bank

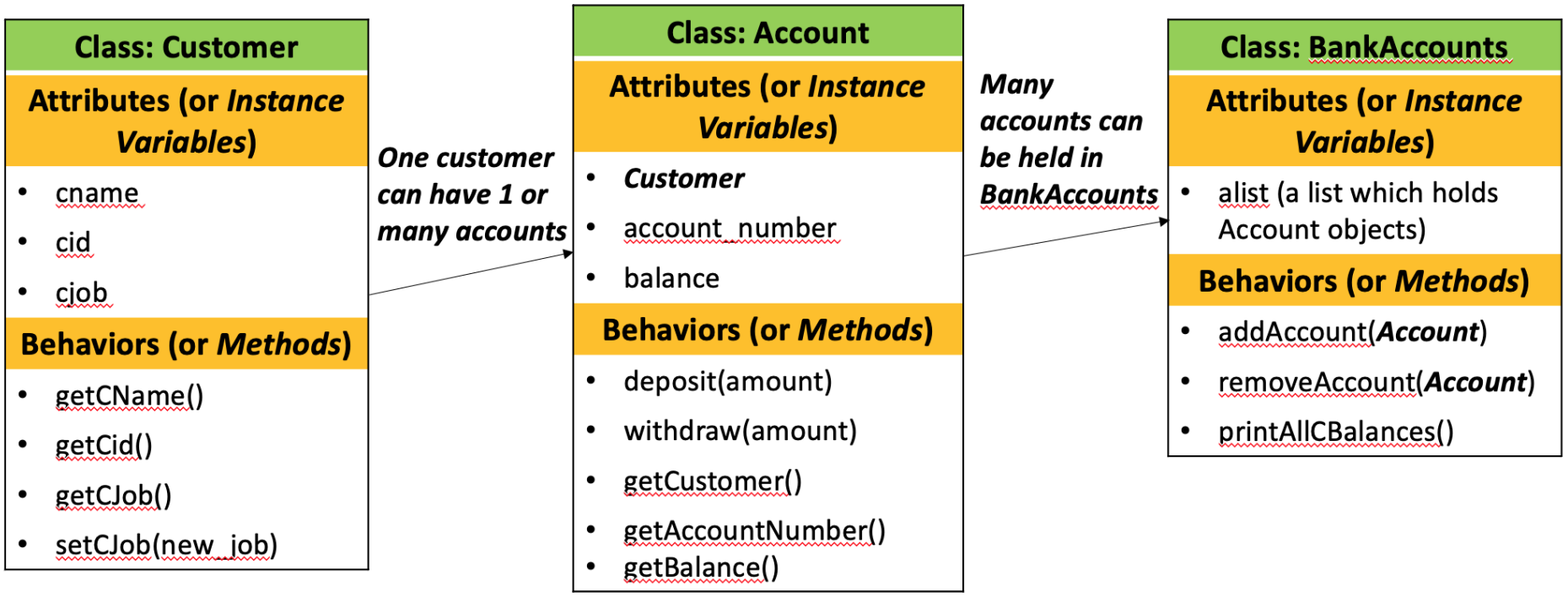


The Customer


```
class Customer:
    def __init__(self, cname, cid, cjob):
        self.cname = cname
        self.cid = cid
        self.cjob = cjob

    def getCName(self):
        return self.cname

    def getCid(self):
        return self.cid

    def getCJob(self):
        return self.cjob

    def setCJob(self, new_job):
        self.cjob = new_job
```

Now Let's Create Accounts for each Customer

- Let us write a simplified *object-based* program for a bank

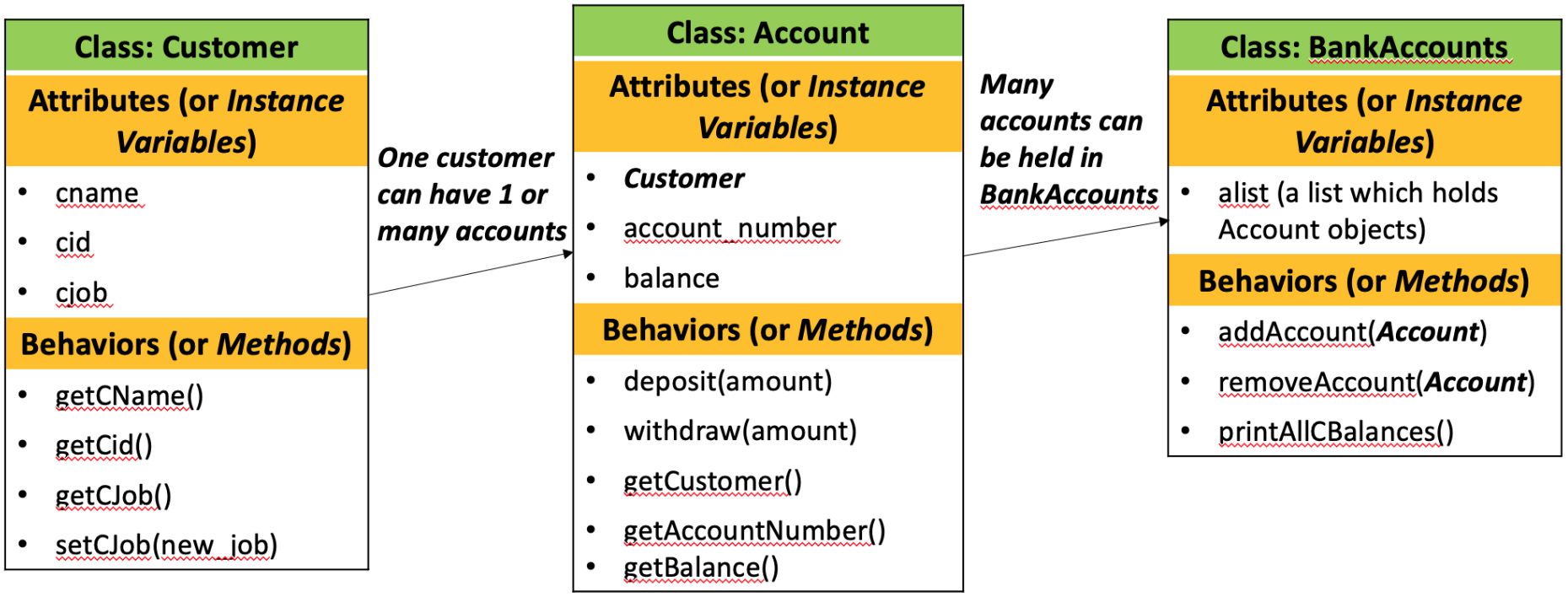


Accounts


```
class Account:
    def __init__(self, customer, account_number, balance):
        self.customer = customer
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        self.balance = self.balance + amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance = self.balance - amount
        else:
            print("You do not have sufficient funds to make this withdrawal!")

    def getCustomer(self):
          return self.customer
```

Now Let's add all accounts to the Bank

- Let us write a simplified *object-based* program for a bank



Accounts


```
class BankAccounts:
    def __init__(self, accounts):
        self.alist = accounts

    def addAccount(self, account):
        for i in self.alist:
            if account is i:
                print("This account has already been added")
                return
        self.alist.append(account)
```

Accounts


```
def removeAccount(self, account):
        counter = 0
        for i in self.alist:
            if account is i:
                self.alist.pop(counter)
                return
            counter = counter + 1

        print("This account does not exist!")

def printAllCBalances(self):
        for i in self.alist:
            print(i.getCustomer().getCName(), i.getBalance())
```

Bank Accounts in Action

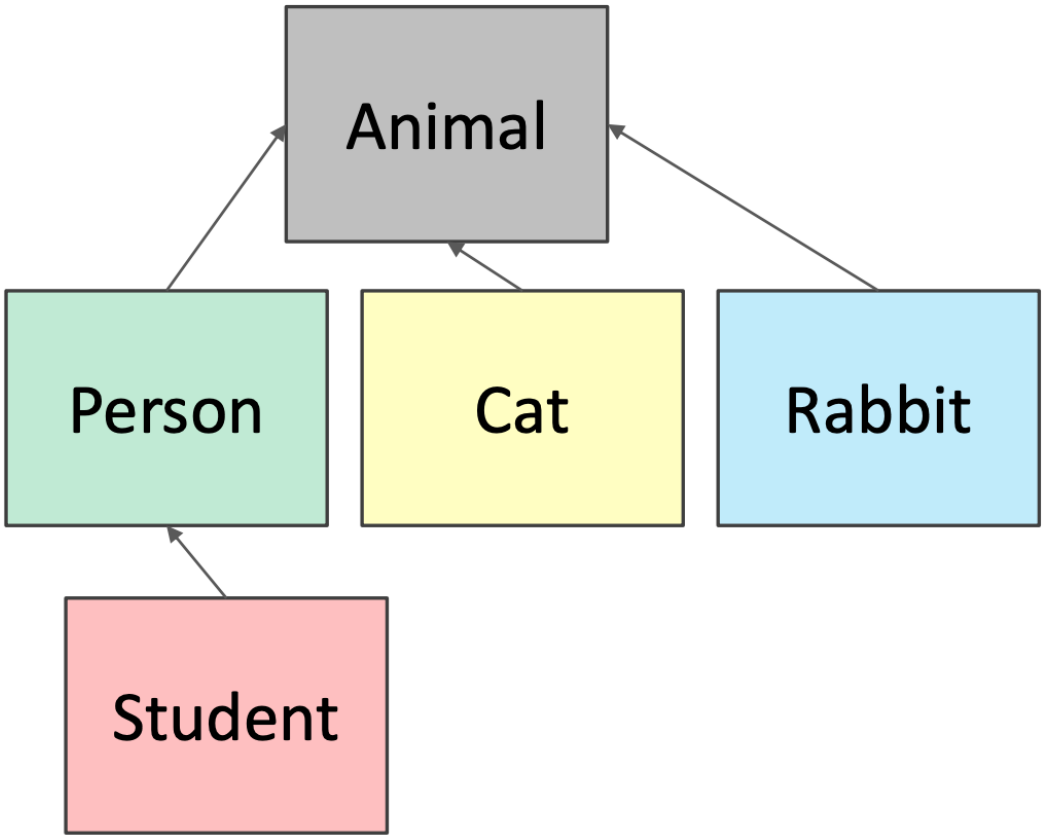

```
c1 = Customer("Maram", 12345, "Student")
c2 = Customer(”Mohamed", 12333, "Teacher")

c1_account = Account(c1, 100, 0)
c2_account = Account(c2, 101, 5000)

bas = BankAccounts([])
bas.addAccount(c1_account)
bas.addAccount(c2_account)
bas.addAccount(c2_account)

c1_account.deposit(20000)
c1_account.withdraw(230)
c2_account.withdraw(1500)
bas.printAllCBalances()

bas.removeAccount(c2_account)
bas.printAllCBalances()
```

Aren't we all Animals?

HIERARCHIES

Hierarchy

- **parent class** (superclass)
- **child class** (subclass)
  - **inherits** all data and behaviors of parent class
  - **add** more **info**
  - **add** more **behavior**
  - **override** behavior



Inheritance

- A crucial concept in Object Oriented Programming is that of Inheritance. Classes can inherit variables and methods of other classes. The new class is called a sub-class.

- Inheritance in Python is easier and more flexible than inheritance in compiled languages such as Java and C++ because the dynamic nature of Python doesn’t force as many restrictions on the language.

The Animal Class


```
class Animal:
    def __init__(self, age):
        self.age = age
        self.name = ""

    def get_name(self):
        return self.name

    def set_name(self, newname=""):
        self.name = newname
```

Let's Bring in Some Cats!


```
class Cat(Animal):

    def speak(self):
        print("Meow")

    def set_name(self, newname=""):
        self.name = newname
```

- add new functionality with `speak()`
  - instance of type `Cat` can be called with new methods
  - instance of type `Animal` throws error if called with `Cat`'s new method
- `__init__` is not missing, uses the `Animal` version

Now with Persons


```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
```

parent class is Animal

call Animal constructor

call Animal's method

add a new data attribute

new methods

Your Turn our Dearest Students


```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
```

bring in methods from `random` class

inherits `Person` and `Animal` attributes

adds new data

- I looked up how to use the `random` class in the python docs
- `random()` method gives back float in [0, 1)

The Super Constructor!


```
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y
class Square(Shape):
    def __init__(self, side=1, x=0, y=0):
        super().__init__(x, y)
        self.side = side
class Circle(Shape):
    def __init__(self, r=1, x=0, y=0):
        super().__init__(x, y)
        self.radius = r
```

In Python, `super()` has two major use cases:

- Allows us to avoid using the base class name explicitly
- Working with Multiple Inheritance

Method Overriding


```
class BaseClass():
def __init__(self):
    self.value = 4
def get_value(self): # overriden method
    return self.value
class Subclass(BaseClass):
def get_value(self): # overriding method
    return self.value * 2
sb = Subclass() print(sb.get_value())
```

- Overriding is the property of a class to change the implementation of a method provided by one of its base classes.
- Method overriding is thus a part of the inheritance mechanism.
- In Python method overriding occurs by simply **defining a method in the child class, with the same name of a method in the parent class.**

Inheritance, Recap

- Classes are allowed to inherit methods and variables from other classes.
- If class A inherits from class B, then class B is called the superclass, and class A the subclass.
- Classes inherit all of the methods and variables in the superclass.
- One can overwrite or add new methods in the subclass as appropriate.

The Power of Inheritence

- Inheritance is a really powerful tool that is easy to abuse.
- Inheritance should be used to represent 'is-a' relations.
  - So a Surgery Patient is a type of Patient.
  - A mammal is a type of animal.
  - A party is a type of event.
- When coming up on to a new problem, a common first step is to think about class structures and what objects you'll need.

Dear OOP, We Love You!

- create your own **collections of data**
- **organize** information
- **division** of work
- access information in a **consistent** manner
- add **layers** of complexity
- like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

Modules

- A module is a file consisting of Python code (.py file).
- A module file groups related functionality you want to include in your application.
- A module can define functions, classes and variables.
- Before being able to use a module, it needs to be imported using the *import* statement

Importing and using a module


```
# support.py
myString = "it is Monday!"
myList = [10, 20, 30]
def myFunc(name):
    print("Hello", name)

class someClass:
    pass
```

Using the 'support' module


```
# usemodule.py
import support
print(support.myString)
print(support.myList)
support.myFunc("Lisa")
o = support.someClass()
```

The import statement

Example

```
import <module_name> #Use the dot notation
import support
print(support.myString)
```

```
from <module_name> import <name(s)>
#Use myString, someClass, etc without the
imported module prefix and dot notation
#Warning: possible name conflict.
```

Example

```
from support import myString, myList
print(myString)
print(myList)
```

Modules search order

When you import a module, the Python interpreter searches for the module in the following order:

1. The current directory from where you called the module.
2. If the module isn't found, Python then searches each directory in the shell variable
3. *PYTHONPATH* if it is set.
4. If all else fails, Python checks the installation-dependent list of directories

Note: The resulting search path is accessible in the Python variable *sys.path*, which can be obtained from a built-in module named *sys*:

```
import sys
for path in sys.path:
    print(path)
```

Some important modules

math

https://docs.python.org/3/library/math.html

random

https://docs.python.org/3/library/random.html

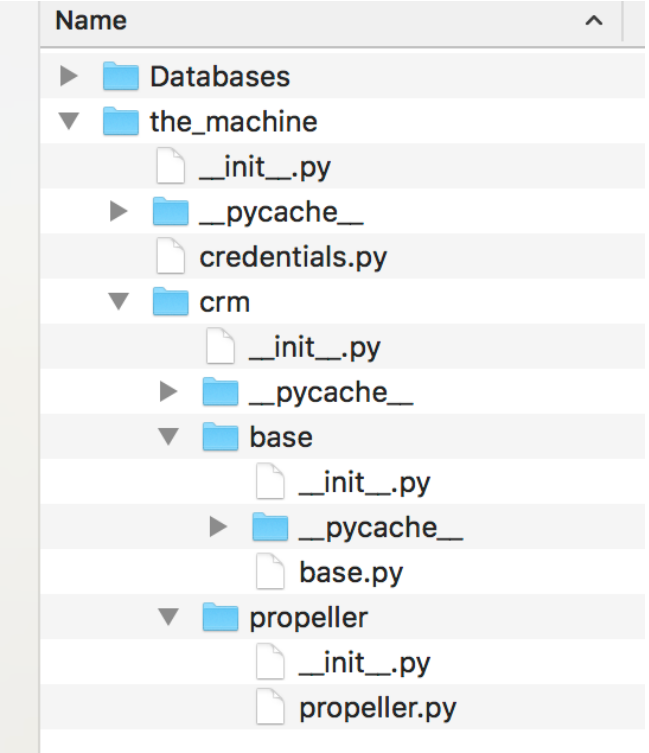
datetime

https://docs.python.org/3/library/datetime.html#module-

os https://docs.python.org/3/library/os.html

Packages

- Packages allow for a hierarchical structuring of modules.
- They help avoid collisions between module names.
- A directory must contain a file named __init__.py in order for Python to consider it as a package.



Summary

- Object Orientation Programming
- Classes and Objects
  - Attributes
  - Methods
  - Static and Private methods
- Inheritance
- Modules
- Packages