Computer Programming in Python

Chapter 9

Classes and Objects

Chapter 9 Classes and Objects

- Two Approaches to Programming
  - ***Procedural***
    - Order of operations follows a flow of control where a specific task is completed, and then another, and then another, and so on
    - The functions are called (some are passed data) to complete a specific task
    - The data and the functionality are separate

Chapter 9 Classes and Objects

- Two Approaches to Programming
  - ***Object-oriented programming (OOP)***
    - Data and functionality are combined in an object and are *hidden* from the rest of the program
      - This is referred to as ***encapsulation***
    - Data items stored by an object are referred to as ***attributes*** or members (sometimes member variables)
    - Functionality within an object is referred to as ***methods*** or behaviors

Chapter 9 Classes and Objects

- Object-oriented Programming (OOP)
  - Terminology
    - Varies depending on the programming language
  - Objects have data elements (attributes)
    - Variables (also referred to as members)
  - Objects have methods (behaviors)
    - Operate on the data elements
    - Provide an interface for other objects and other parts of the program to access and operate on them

Chapter 9 Classes and Objects

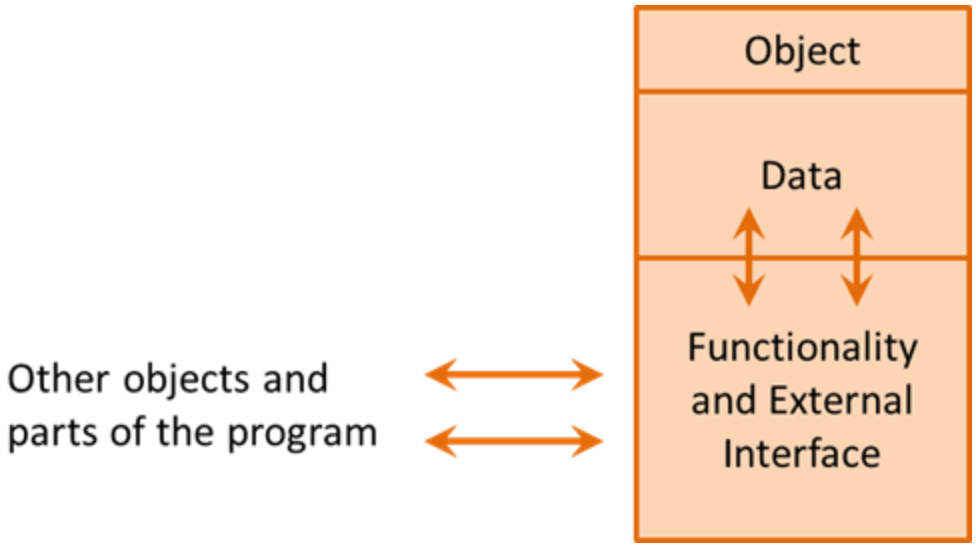
- Object-oriented Programming
  - The data is hidden from outside the object to protect it from being corrupted or changed arbitrarily
  - Parts of a program and other objects can access the object's attributes through a ***public interface*** which provides protection for the data
    - Referred to as information hiding
    - Programs can use an object without knowing the inner workings

Chapter 9 Classes and Objects

- Public Interface
  - Interacting with an object through the public interface only requires knowledge of the interface
  - Future changes to an object internally do not necessarily require changes to a program that uses that object
  - Unless the public interface has changed, there is typically no need to modify programs that use the object

Chapter 9 Classes and Objects

- Public Interface
  - Interacting with an object through the public interface only requires knowledge of the interface

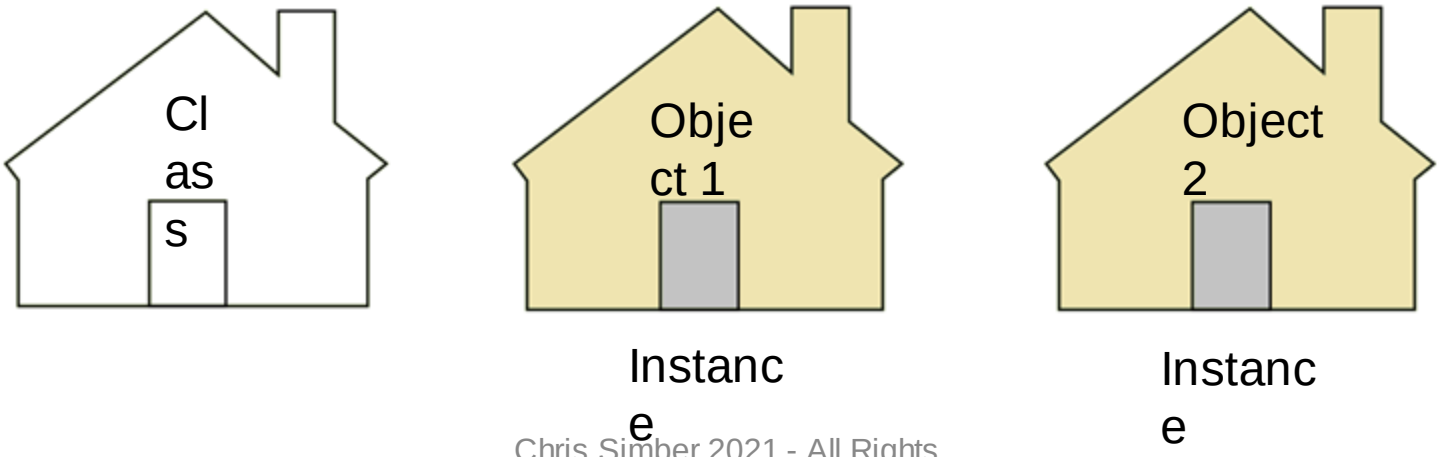


Chapter 9 Classes and Objects

- Classes
  - A ***class*** is a framework or blueprint of what the object will contain when it is created
    - An architect can provide a detailed drawing of a building that shows a door, but the door cannot be opened
    - A building must be built from the drawing, and then the door of the building can be opened
    - The building would be an instance of the drawing the same way that an object is an ***instance*** of a class

Chapter 9 Classes and Objects

- Classes
  - Multiple buildings could be built from the same drawing, they would be identical, and they would each have their own door
  - Multiple objects can be ***instantiated*** from a single class, and they would each have their own set of attributes



Chapter 9 Classes and Objects

- Classes
  - The ***class definition*** defines the attributes for a class
    - Outlines the data attributes and methods for the class

```
class ClassName:
    def __init__(self, parameter, parameter, ...):

        statement
        statement
        ...
```

Chapter 9 Classes and Objects

- Class Definition
  - Begins with the class key word and the name of the class
    - The naming convention for classes is to begin each word with an uppercase letter

```
class ClassName:
    def __init__(self, parameter, parameter, ...):

        statement
        statement
        ...
```

Chapter 9 Classes and Objects

- Class Definition
  - The next line defines the ***constructor*** which looks like a function or method header
    - Includes the word __init__ (short for ***initializer***)
      - It is preceded and followed by two underscores

```
class ClassName:
    def __init__(self, parameter, parameter, ...):

        statement
        statement
        ...
```

Chapter 9 Classes and Objects

- Class Definition
  - The constructor executes when an object of the class is created
  - Any parameters used by the constructor would be included along with the parameter ***self*** which is a reference to the object being created
    - self is typical and considered the standard although root and others are common

```
class ClassName:
    def __init__(self, parameter, parameter, ...):
```

Chapter 9 Classes and Objects

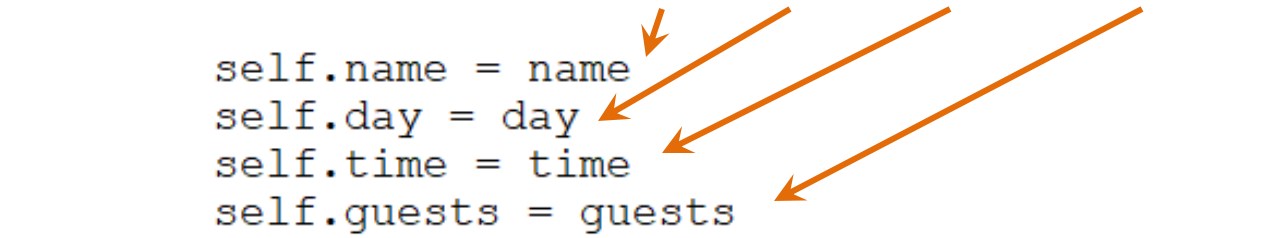
- Class Definition Example
  - A Reservation Class
    - Attributes for name, day, time, number of guests
  - To create an instance of the Reservation class requires passing the constructor a name, day, time, and number of guests

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests
```

Chapter 9 Classes and Objects

- Class Definition Example
  - The four lines within the class declare and ***initialize*** the instance attributes using the values of the parameters that were passed to the constructor

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests
```



Chapter 9 Classes and Objects

- Class Definition Example
  - r1 is an instance of the Reservation class

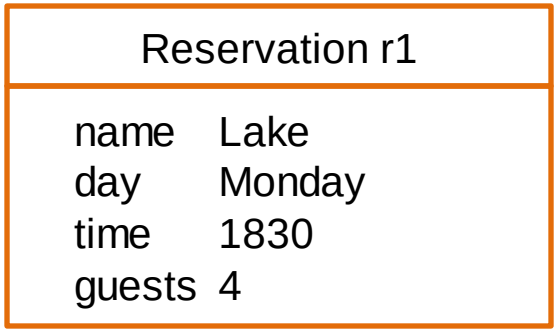
```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests


def main():

    r1 = Reservation('Lake', 'Monday', 1830, 4)
    print(r1.name, r1.day, r1.time, r1.guests)

main()
```

```
Lake Monday 1800 4
```

Chapter 9 Classes and Objects

- Class Definition Example
  - Each *Reservation* object created will have its own set of attributes with its own set of values
    - If there were methods, each object would have its own set of methods
  - The values stored in the instance attributes are referred to as the object's ***state***

state of r1



Chapter 9 Classes and Objects

- Two objects (r1 and r2) with different attribute values (states)

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests

def main():

    r1 = Reservation('Lake', 'Monday', 1830, 4)
    r2 = Reservation('Bethea', 'Tuesday', 1900, 6)
    print(r1.name, r1.day, r1.time, r1.guests)
    print(r2.name, r2.day, r2.time, r2.guests)

main()
```

```
Lake Monday 1830 4
Bethea Tuesday 1900 6
```

Chapter 9 Classes and Objects

- Classes
  - Some class constructors receive no parameters when an object is created
    - There would usually be methods within the class for assigning values to the attributes
    - Also, consider that a *Reservation* may need to be changed
  - To provide for this capability, a method would be added to the class definition which would allow a change to the state of the object
    - Through the public interface

Chapter 9 Classes and Objects

- Class Methods
  - The behavior of an object is specified by writing methods in the class definition
  - A ***method*** is like a function, but it is inside an object and interacts with the data elements of the object
  - These methods provide the public interface

Class Methods provide a public interface

Chapter 9 Classes and Objects

- Class Methods
  - The Reservation class could provide the functionality to change the time of the reservation
  - A program using the class could then change the state of time through this method

```
def change_time(self, time):
    self.time = time
```

Class Method to change the time attribute

Chapter 9 Classes and Objects

- Class Methods
  - The method would be included in the class definition
    - The first parameter is the object on which the method is being called
      - It is received as self by the method
    - The second is the time that will be assigned to the time attribute of the object

```
def change_time(self, time):
    self.time = time
```

Chapter 9 Classes and Objects

- Class Methods
  - The method is included in the class

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests

    def change_time(self, time):
        self.time = time
```

Note the indentation and alignment

Chapter 9 Classes and Objects

- Class Methods
  - After an object of the class is created, the method can be called
    - Object name, the dot operator, and the method name
  - Only one argument is passed to the method
  - The object reference is passed automatically

```
r1.change_time(1800)
```

The object reference is passed automatically

Chapter 9 Classes and Objects

- Class Methods
  - Methods to enable changing each of the *Reservation* attributes could be added
  - If the constructor for a class does not have parameters, all of the attributes for the class might be set through methods
  - Different requirements call for different solutions

Chapter 9 Classes and Objects

- Class Methods
  - So how does this protect the data attribute?
  - The public interface methods can include protections like input validation

```
def change_time(self, time):
    if time <= 0 or time > 2359:
        print('Invalid time entered.')
    else:
        self.time = time
```

Chapter 9 Classes and Objects

- Class Access Specifiers
  - Different parts of an object are designated for access using what are referred to as ***access specifiers***
    - ***Public*** Access Modifier: The members declared as public (which is the default) are deemed accessible from outside an object of the class.
    - ***Protected*** Access Modifier: The members designated as protected are deemed accessible only from a class derived from it (a subclass).
    - ***Private*** Access Modifier: These members are designated as only accessible from within the class. Access from outside the class is deemed inappropriate.

Chapter 9 Classes and Objects

- Class Access Specifiers
  - Python does not have an effective way of specifying or enforcing ***public***, ***protected***, and ***private*** access, but a convention is used to signify access
  - A single preceding underscore to *indicate* ***protected***
  - Two underscores preceding an item to *indicate* that it is ***private***
  - Since Python uses name mangling (beyond the scope of this text), some protection is afforded private elements (although they are still technically accessible)

Chapter 9 Classes and Objects

- Class Access Specifiers
  - The Reservation class with the attributes specified as private attributes

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.__name = name
        self.__day = day
        self.__time = time
        self.__guests = guests
```

Specify attributes as private using two underscores

Chapter 9 Classes and Objects

- Class Methods - Mutators
  - Methods that change the state (attributes) of an object are referred to as ***mutator*** methods
    - Should have names that indicate what they change and typically begin with the word *set*
      - Often referred to as 'setters'

```
set_some_value(value)      # mutator method
```

Mutator Methods change object attributes

Chapter 9 Classes and Objects

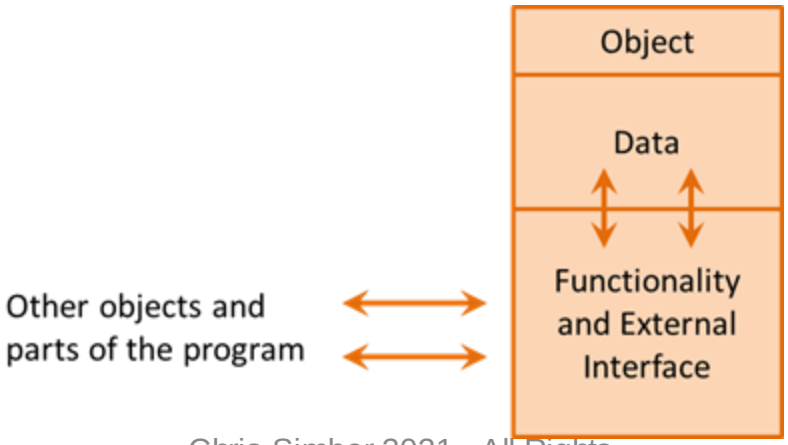
- Class Methods - Accessors
  - Methods that access an object's attributes without changing them are referred to as ***accessor*** methods
    - Should have names that indicate what they access and typically begin with the word *get*
      - Often referred to as getters

```
get_some_value( )          # accessor method
```

Accessor Methods access object attributes

Chapter 9 Classes and Objects

- Public Interface
  - With the attributes of an object defined as private, the public interface provides the only access to them
  - There is no other way for another part of the program or another object to access or change the data



Chapter 9 Classes and Objects

- Class Attributes
  - An attribute that is declared outside the __init__ function is a Class attribute and is shared by all objects of the class
    - Useful for class constants, tracking data across all instances of the class (similar to static variables), and for defining defaults values
  - A class attribute can be accessed using the class name or the objects name

Class attributes are shared by all objects of the class

Chapter 9 Classes and Objects

- Business Program Invoice Class
  - Class attributes for a state tax and list of invoices
    - Shared by all instances (objects) of the class
    - When an invoice object is created, the constructor adds it the list

Class Attributes Defined outside any function

```
class Invoice:

    state_tax = 0.05
    invoice_list = []

    def __init__(self, customer, address):

        self.__customer = customer
        self.__address = address

        self.invoice_list.append(self)
                .
                .
```

Chapter 9 Classes and Objects

- Class Files and Modularization
  - Class definitions are typically located in separate files
    - Aligns with the process of ***modularization***, or separating a program into distinct parts
    - Provides many benefits including the ability to:
      - Reuse portions of the code
      - Divide the program development among multiple programmers
      - Simplify the overall project

Chapter 9 Classes and Objects

- Class Files and Modularization
  - Reservation Class File

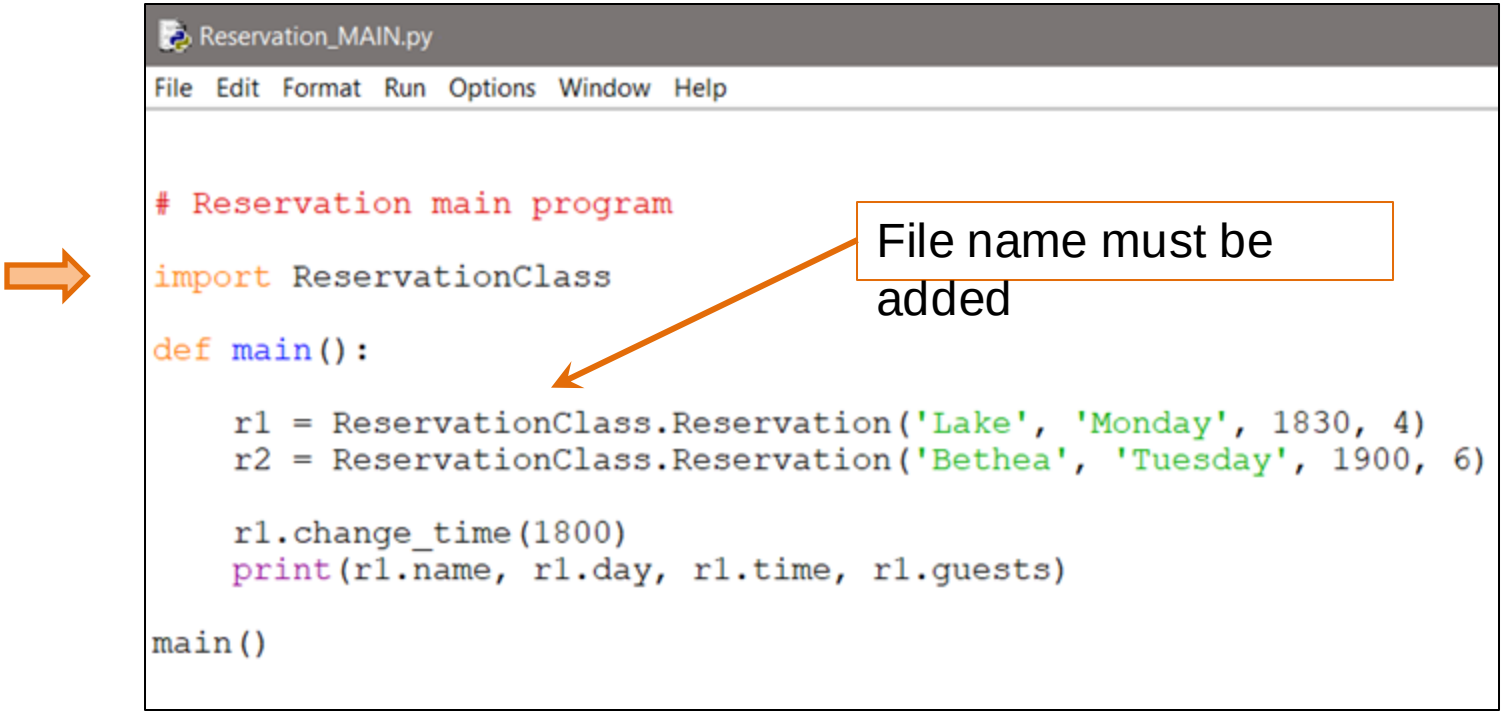
ReservationClass.py

File Edit Format Run Options Window Help

```
# Reservation.py file - class definition

class Reservation:
    def __init__(self, name, day, time, guests):

        self.name = name
        self.day = day
        self.time = time
        self.guests = guests


    def change_time(self, time):
        self.time = time
```

Chapter 9 Classes and Objects

- Class Files and Modularization
  - Reservation Class File imported into the main file



Chapter 9 Classes and Objects

- Displaying the State of an Object
  - The print function will not display an object's state
    - It will display an objects location in memory

```
<__main__.Reservation object at 0x0000026D923C0040>
```

  - Python provides the capability to write a method that returns a programmer defined string of what is to be displayed
    - The __str__ method will be used by the object when the print function is called

Chapter 9 Classes and Objects

- Displaying the State of an Object
  - When the print function is called with an object, the programmer defined __str__ method is executed
    - The method is defined within the class

```
def __str__(self):
    return 'The data  is ' + self.name + '-' + \
           str(self.day) + '-' + str(self.time) + \
           '-' + str(self.guests)
```

Chapter 9 Classes and Objects

- Objects as Arguments
  - When passing objects as arguments to functions and methods, the parameter is a reference to the object
    - Provides access to the object's methods and

```
def main():

    r1 = Reservation('Lake', 'Monday', 1830, 4)

    display_res_name(r1)


def display_res_name(obj):
    print('The name is ' + obj.name)

main()
```

Chapter 9 Classes and Objects

Designing Classes

Chapter 9 Classes and Objects

- Designing Classes
  - Determine the data attributes
  - Determine the public interface (methods) needed to access and change the data when appropriate
  - Most classes model real-world objects and should represent a clear and single ***abstraction***
    - The class should not include functionality that is outside its responsibilities like getting user input, or anything that is specific to a particular program

Chapter 9 Classes and Objects

- Designing Classes
  - A goal of OOP is reuse, another is ***cohesion***
    - The degree to which a class represents a single abstraction without external dependencies – high cohesion
  - The degree to which a class depends on another is referred to as ***coupling***
    - OOP strives for loose coupling
  - A class should have cohesion (represent a stand-alone entity), and loose coupling (no external dependencies).

Chapter 9 Classes and Objects

- Designing Classes - Tools
  - Class Responsibility and Collaborator or ***CRC Cards***
    - Teams brain-storm ideas and note the nouns and verbs used when describing a class
    - Nouns represent instance variables (data elements)
    - Verbs represent the methods that will act on the data including the public interface
    - The goal is to capture all possible data elements and methods, and then refine the lists as the design evolves.

Chapter 9 Classes and Objects

- Designing Classes - Tools
  - Consider the Reservation Class

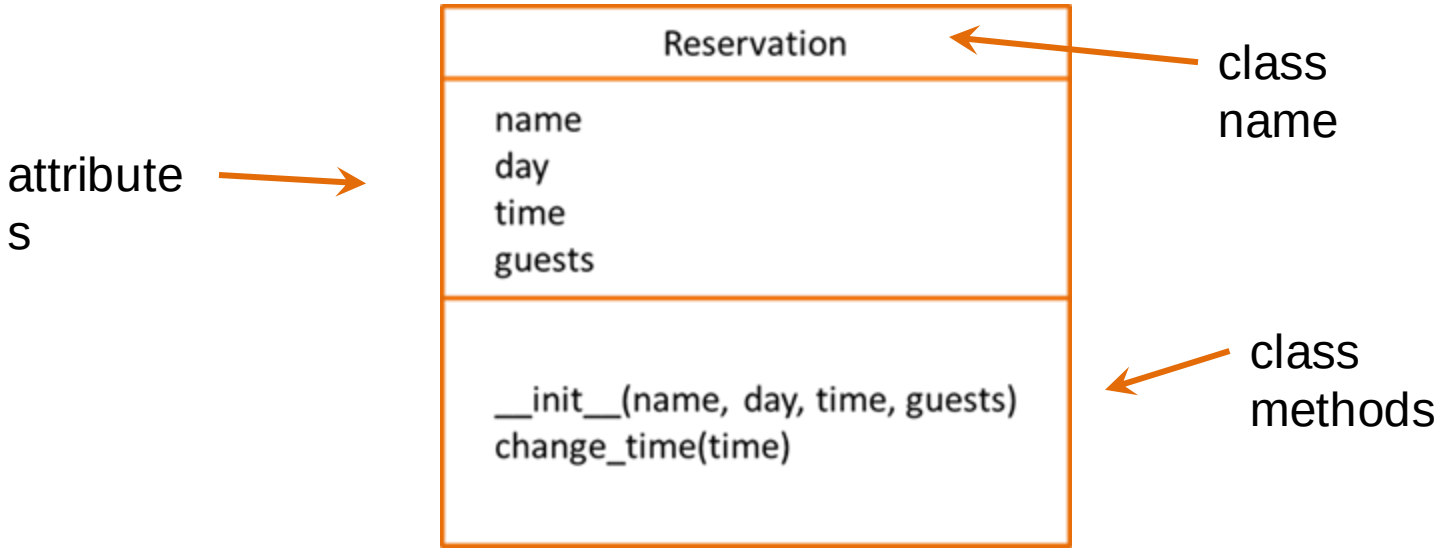
Nouns	Verbs
name	get name
day	get, change day
time	get, change time
guests	get, change guests

Chapter 9 Classes and Objects

- Designing Classes - Tools
  - The Unified Modeling Language (UML) was developed as a standard for visualizing and documenting systems and software
  - UML diagrams are used to design and document classes and object oriented software systems including object interaction

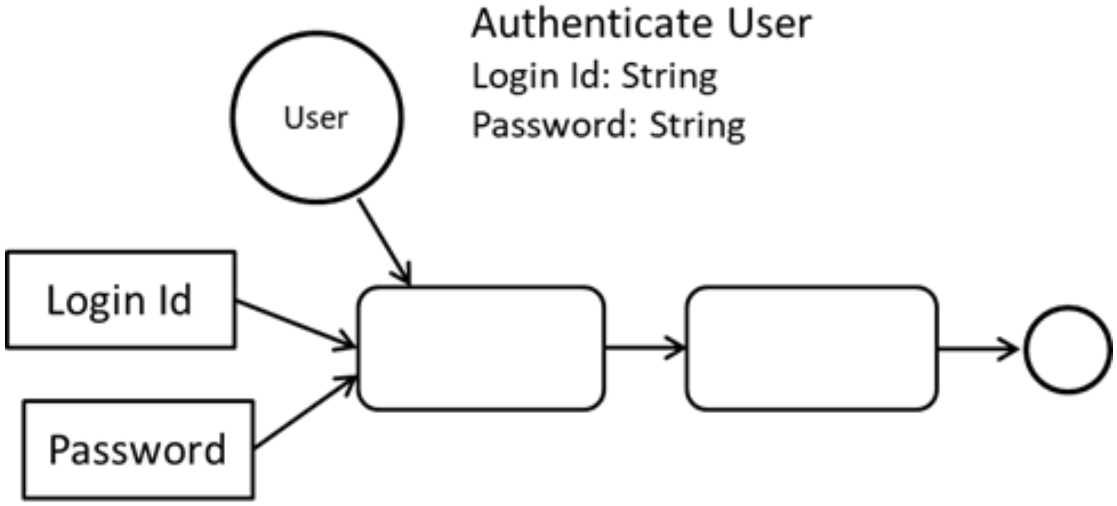
Chapter 9 Classes and Objects

- UML Class Diagram
  - The top section contains the name of the class, the next section describes the data attributes, and the bottom section lists the methods for the class



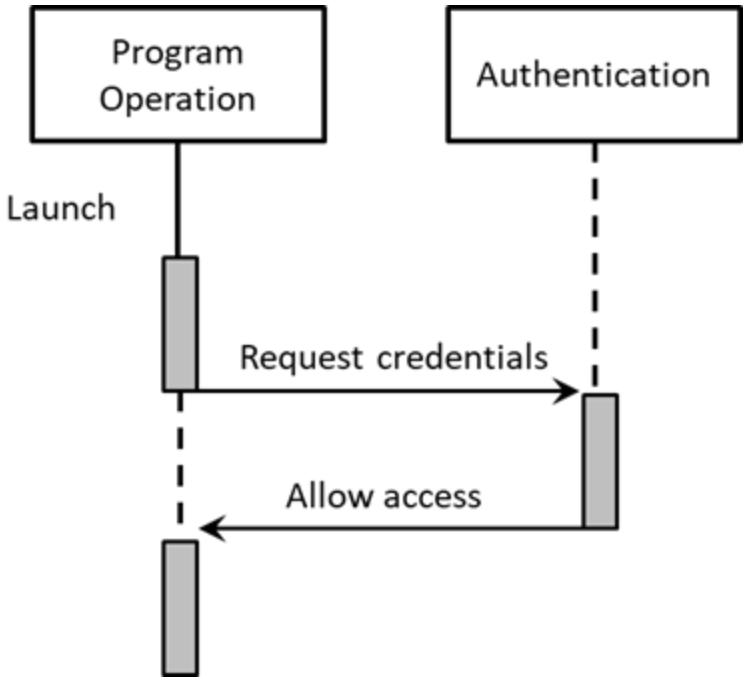
Chapter 9 Classes and Objects

- Designing Classes - Tools
  - UML behavior diagrams (object activity diagrams) are used to show the flow of control, data, and transactions.



Chapter 9 Classes and Objects

- Designing Classes - Tools
  - The Object Sequence Diagram adds chronology



Chapter 9 Classes and Objects

- A More Complete Reservation Class

```
class Reservation:
    def __init__(self, name, day, time, guests):

        self.__name = name
        self.__day = day
        self.__time = time
        self.__guests = guests

    def set_day(self, day):
        self.__day = day

    def set_time(self, time):
        self.__time = time

    def set_guests(self, guests):
        self.__guests = guests
```

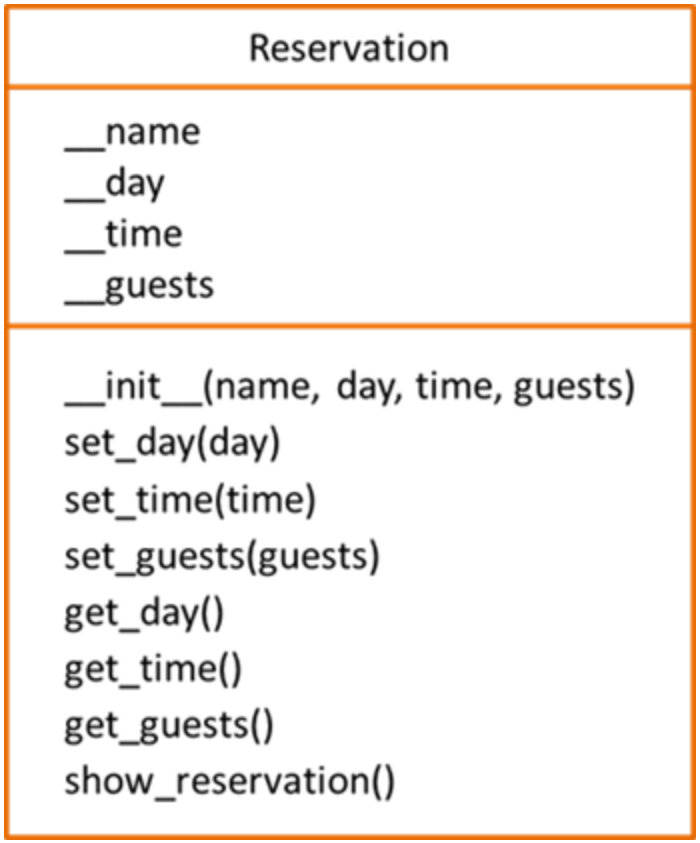
Chapter 9 Classes and Objects

- A More Complete Reservation Class

```
def get_day(self):
    return self.__day

def get_time(self):
    return self.__time

def get_guests(self):
    return self.__guests

def show_reservation(self):
    print('Name:' + self.__name + '\nDay: ' + str(self.__day) +
          '\nTime: ' + str(self.__time) +
          '\nGuest: ' + str(self.__guests))
```

Chapter 9 Classes and Objects

- A More Complete Reservation Class UML



Chapter 9 Classes and Objects

- Pickling
  - Objects can be saved in a file either as text or they can be ***serialized***
  - In Python, serializing objects is called ***pickling*** which converts objects into byte streams (0s and 1s)
    - Import pickle
  - This allows preserving an object's state, and for streaming objects across networks from one server to another

Chapter 9 Classes and Objects

- Pickling
  - The *pickle* module's ***dump*** function serializes an object and writes it to a file
  - The ***load*** function retrieves an object from a file and de-serializes it
  - Serializing uses two different file modes
    - 'wb' to write binary
    - 'rb' to read binary

Pickling uses the binary file modes

Chapter 9 Classes and Objects

- Pickling

```
r1 = ResClass.Reservation('Lake', 'Monday', 1830, 4)

outfile = open('resData.txt', 'wb')

pickle.dump(r1, outfile)

outfile.close()

infile = open('resData.txt', 'rb')

res = pickle.load(infile)

infile.close()

res.show_reservation()
```

serialize

reconstitute

Chapter 9 Classes and Objects

- Pickling Warnings
  - There are many warnings associated with pickling
    - The pickle module is not secure
    - Only un-pickle (reconstitute) trusted data
    - Malicious code can be executed during un-pickling
  - There are also some data type and function/method limitations
  - Other products such as JSON provide this capability
    - JavaScript Object Notation

Only un-pickle (reconstitute) trusted data

Chapter 9 Classes and Objects

Object Inheritance

Chapter 9 Classes and Objects

- Inheritance
  - Objects are often specialized versions of a general class
  - A restaurant is a business, a clothing store is a business, and so is a theater
  - They have many things in common like employees, sales, and expenses, but some differences or special characteristics as well

Chapter 9 Classes and Objects

- Inheritance
  - The common characteristics for the businesses could be implemented in a Business class, and then each specific business type could be ***derived*** from that class
  - The derived classes would inherit the common characteristics from the Business class, and would implement those that are specific to them
  - They would ***extend*** the Business Class
    - This is referred to as the “is a” relationship
  - This relationship is established through ***inheritance***

Chapter 9 Classes and Objects

- Inheritance
  - The specialized classes (derived classes or subclasses) inherit the characteristics of the general class (base or super class)
  - In the diagram, the Business Class is the base class, and the spe es

Chapter 9 Classes and Objects

- Inheritance – a note about terminology
  - Different programming languages and texts use different terms to refer to the same thing

General Class	Specialized Class
Base Class	Derived Class
Parent Class	Child Class
Super Class	Sub-class

Base (General Class) and Derived (Specialized Class)

Chapter 9 Classes and Objects

- Inheritance – Example:
  - A program is needed that can be used by a corporation with different businesses
  - The common characteristics for all of the businesses would be implemented in the Business Class
    - Name, and number of employees
  - The derived classes would contain their specific items and methods

Chapter 9 Classes and Objects

- Inheritance – Example:
  - Business class containing common attributes

```
class Business:
    def __init__(self, name, employees):

        self.__name = name
        self.__employees = employees

    def get_name(self):
        return self.__name
```

Chapter 9 Classes and Objects

- Inheritance – Example:
  - The first line defines a class ClothingStore that inherits from Business


```
class ClothingStore(Business):

    def __init__(self, name, employees, inventory):

        Business.__init__(self, name, employees)
```

Chapter 9 Classes and Objects

- Inheritance – Example:
  - The constructor for Business is called inside the constructor for the ClothingStore

```
class ClothingStore(Business):

    def __init__(self, name, employees, inventory):

        Business.__init__(self, name, employees)
```

Business constructor

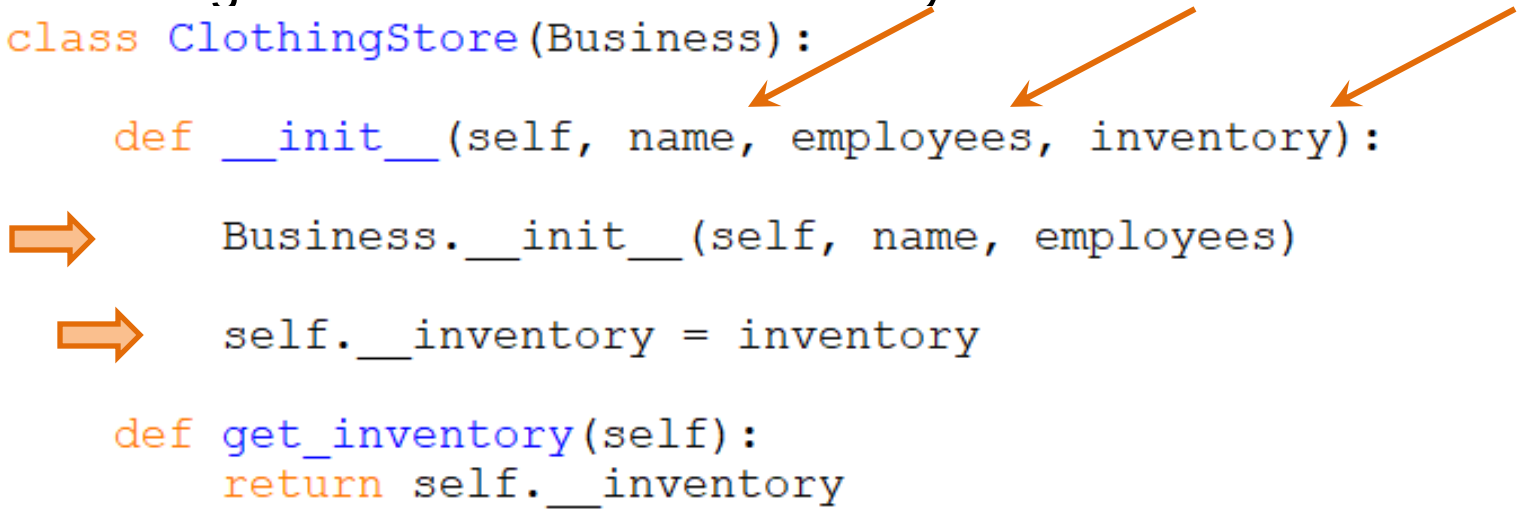
Chapter 9 Classes and Objects

- Inheritance – Example:
  - ClothingStore inherits attributes from the Business class
  - The ClothingStore has an attribute for inventory

```
class ClothingStore(Business):

    def __init__(self, name, employees, inventory):

        Business.__init__(self, name, employees)

        self.__inventory = inventory

    def get_inventory(self):
        return self.__inventory
```

Chapter 9 Classes and Objects

- Note the Clothing Store constructor parameters
  - Clothing Store needs name and employees which are passed to the constructor for the Business class (along with self)
  - Clothing Store also needs inventory



```
class ClothingStore(Business):

    def __init__(self, name, employees, inventory):

        Business.__init__(self, name, employees)

        self.__inventory = inventory

    def get_inventory(self):
        return self.__inventory
```

Chapter 9 Classes and Objects

- Business Class Methods are Inherited
  - ClothingStore inherits name, employees, and get_name() from the Business class

```
def main():

    b1 = ClothingStore("Ferry's", 4, 20)

    print(b1.get_inventory())
    print(b1.get_name())

main()
```

```
20
Ferry's
```

Chapter 9 Classes and Objects

- Adding a Theater class as a derived class
  - In addition to name and employees, the Theater class has an attribute for the number of seats

```
class Theater(Business):

    def __init__(self, name, employees, seats):

        Business.__init__(self, name, employees)

        self.__seats = seats

    def get_seats(self):
        return self.__seats
```

Chapter 9 Classes and Objects

- Inheritance is One Way
  - Base classes do not know anything about derived classes
  - Derived classes know everything about base classes
    - Except as noted by Access Qualifiers

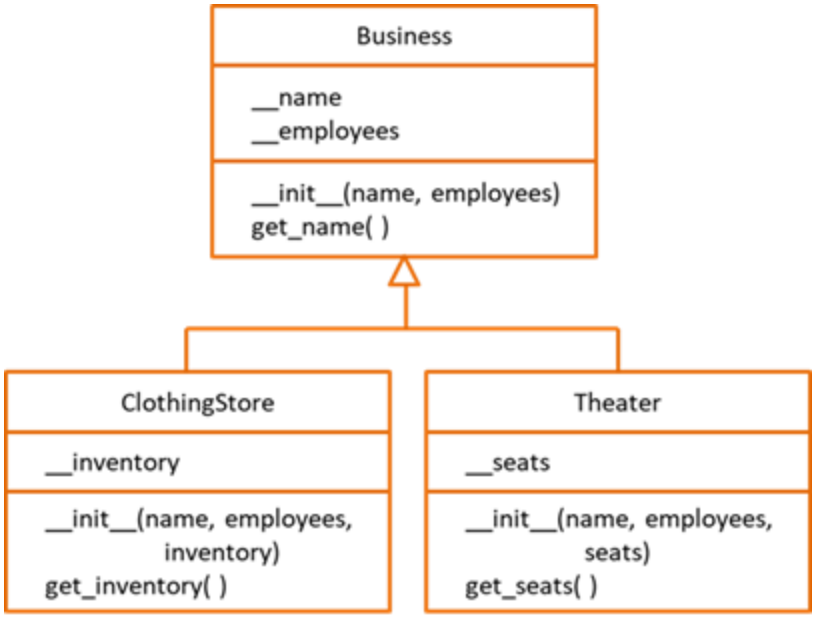
Public Access Modifier: The members declared as public (which is the default) are deemed accessible from outside an object of the class.

Protected Access Modifier: The members designated as protected (preceded by a single underscore) are deemed accessible only from a class derived from it (in a subclass).

Private Access Modifier: These members are designated (preceded by two underscores) as only accessible from within the class. Access from outside the class is deemed inappropriate.

Chapter 9 Classes and Objects

- Inheritance in UML
  - The UML diagram shows that Clothing Store and Theater both inherit from the Business class



Chapter 9 Classes and Objects

- Polymorphism
  - A derived class can have a method with the same name as a method in the base class that operates differently
  - This is known as ***polymorphism***
    - The ability to take on many forms
  - The derived class method can ***override*** the base class method

Chapter 9 Classes and Objects

- Polymorphism
  - If the Business Class had a display method, it could include *name* and *employees*
    - They are attributes of the base class
  - It could not include attributes from the derived classes

```
class Business:
    def __init__(self, name, employees):
        self.__name = name
        self.__employees = employees

    def get_name(self):
        return self.__name
```

Chapter 9 Classes and Objects

- Polymorphism
  - The derived classes could have their own display method that overrides the display method in the base class
    - The Clothing Store class could have a display method that includes inventory
    - The Theater class could have a display method that includes number of seats
  - The proper method would be called based upon the object that is calling the method

Derived class methods can override base class methods

Chapter 9 Classes and Objects

- Polymorphism
  - Each method has the same name and parameter list

```
def show_data(self):
        print(name, employees)
```

Business class method

```
def show_data(self):
        print(name, employees, inventory)
```

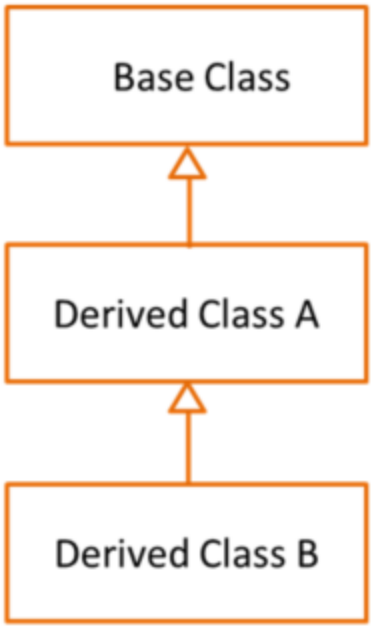
Clothing Store class method

```
def show_data(self):
        print(name, employees, seats)
```

Theater class method

Chapter 9 Classes and Objects

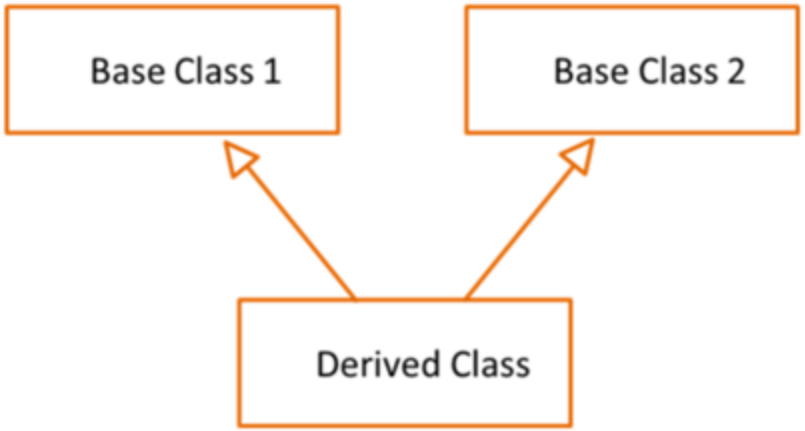
- Multiple Class Inheritance
  - A class can inherit (be derived) from multiple classes
  - The derived class inherits all of the attributes of both classes
    - Inline with Access Qualifiers
  - The hierarchy diagram indicates the direct inheritance of Derived Class B from Derived Class A which inherits from the Base Class



Direct Inheritance

Chapter 9 Classes and Objects

- Multiple Class Inheritance
  - A class can also inherit from two or more distinct classes



Both base classes should be reviewed carefully to avoid conflicts

Chapter 9 Classes and Objects

- Multiple Class Inheritance
  - When defining a class with multiple-inheritance:
    - Both classes are listed as parameters for the derived class
    - Both initializers for the other classes are called

```
class DerivedB(DerivedA, BaseClass):
    def __init__(self):
        DerivedA.__init__(self)
        BaseClass.__init__(self)
```

Chapter 9 Classes and Objects

- Multiple Class Inheritance
  - Both forms of multiple-inheritance add complexity and make the code and classes harder to re-use in other programs
  - Both of the inherited classes need to be copied as well
  - Lowers cohesion and increases coupling

Goal: High cohesion and loose coupling

Chapter 9 Classes and Objects

- Determining Class Identity
  - With complex classes, it is often necessary to determine if an object is an instance of a class
  - The ***isinstance*** function
    - Returns True if the object is an instance of a class
    - False if it is not

```
isinstance(object,Class)
```

Chapter 9 Classes and Objects