

# 13.1 Derived classes

A class will commonly share attributes with another class, but with some additions or variations. For example, a store inventory system might use a class called Item, having name and quantity attributes. But for fruits and vegetables, a class Produce might have name, quantity, and expiration date attributes. Note that Produce is really an Item with an additional feature, so ideally a program could define the Produce class as being the same as the Item class but with the addition of an expiration date attribute.

Such similarity among classes is supported by indicating that a class is *derived* from another class, as shown below.

Figure 13.1.1: A derived class example: Class Produce is derived from class Items.

```
class Item:
    def __init__(self):
        self.name = ''
        self.quantity = 0

    def set_name(self, nm):
        self.name = nm

    def set_quantity(self, qnty):
        self.quantity = qnty

    def display(self):
        print(self.name, self.quantity)

class Produce(Item): # Derived from Item
    def __init__(self):
        Item.__init__(self) # Call base class constructor
        self.expiration = ''

    def set_expiration(self, expr):
        self.expiration = expr

    def get_expiration(self):
        return self.expiration

item1 = Item()
item1.set_name('Smith Cereal')
item1.set_quantity(9)
item1.display()

item2 = Produce()
item2.set_name('Apples')
item2.set_quantity(40)
item2.set_expiration('May 5, 2012')
item2.display()
print(' (Expires:(%s))' % item2.get_expiration())
```

Smith Cereal 9  
Apples 40  
(Expires:(May 5, 2012))

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

The example defines a class named Item. In the script, an instance of Item called item1 is created, the instance's attributes are set to Smith Cereal and 9, and the display() method is called. A class named Produce is also defined, that class was *derived* from the Item class by including the base class Item within parentheses after Produce, i.e., `class Produce(Item):`. As such, instantiating a Produce instance item2 creates an instance object with data attributes name and quantity (from Item), plus expiration (from Produce), as well as with the methods `set_name()`, `set_quantity()`, and `display()` from Item, and `set_expiration()` and `get_expiration()` from Produce. In the script, item2 has instance data attributes set to Apples, 40, and May 5, 2012. The `display()` method is called, and then the expiration date is printed using the `get_expiration()` method.<sup>interfaces</sup>

All of the class attributes of Item are available to instances of Produce, though instance attributes are not. The `__init__` method of Item must be explicitly called in the constructor of Produce, e.g., `Item.__init__(self)`, so that the instance of Produce is assigned the name and quantity data attributes. When an instantiation of a Produce instance occurs, `Produce.__init__()` executes and immediately calls `Item.__init__()`. The newly created Produce instance is passed as the first argument (`self`) to the Item constructor, which creates the name and quantity attributes in the new Item instance's namespace. `Item.__init__()` returns, and `Produce.__init__()` continues, creating the expiration attribute. The following tool illustrates:

### PARTICIPATION ACTIVITY

#### 13.1.1: Derived class explicitly calls base class' constructor.



```

1  class Item:
2      def __init__(self):
3          self.name = ''
4          self.quantity = 0
5
6  class Produce(Item):
7      def __init__(self):
8          Item.__init__(self)
9          self.expiration = ''
10
11 item1 = Item()
12 item2 = Produce()

```

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

<< First < Back Step 1 of 16 Forward > L

→ line that has just executed

→ next line to execute

Frames

Objects

The term **derived class** refers to a class that inherits the class attributes of another class, known as a **base class**. Any class may serve as a base class; no changes to the definition of that class are required. The derived class is said to *inherit* the attributes of its base class, a concept commonly called **inheritance**. An instance of a derived class type has access to all the attributes of the derived class as well as the *class* attributes of the base class by default, including the base class' methods. A derived class instance can simulate inheritance of *instance* attributes as well by calling the base class constructor manually. The following animation illustrates the relationship between a derived class and a base class.

©zyBooks 11/11/19 21:02 569464  
Dien VU  
UNIONCSC103OrhanFall2019

#### PARTICIPATION ACTIVITY

13.1.2: Derived class example: Produce derived from Item.



#### Animation captions:

1. Item is the base class.
2. Produce is derived so Produce inherits Item's attributes.

The inheritance relationship is commonly drawn as follows, using Unified Modeling Language (UML) notation ([Wikipedia: UML](#)).

#### PARTICIPATION ACTIVITY

13.1.3: UML derived class example: Produce derived from Item.



#### Animation content:

undefined

#### Animation captions:

1. A class diagram depicts a class' name, data members, and methods.
2. A solid line with a closed, unfilled arrowhead indicates a class is derived from another class.
3. The derived class only shows additional members.

e40e04a9-f040-47f8-ac41-1216ba2405cc

©zyBooks 11/11/19 21:02 569464

In the above animation, the +, -, and # symbols refer to the access level of an attribute, i.e., whether or not that attribute can be accessed by anyone (public), instances derived from that class (private), or only instances of that class (protected), respectively. In Python, all attributes are public.<sup>privacy</sup>. Many languages, like Java, C, and C++, explicitly require setting access levels on every variable and function in a class, thus UML as a language-independent tool includes the symbols.

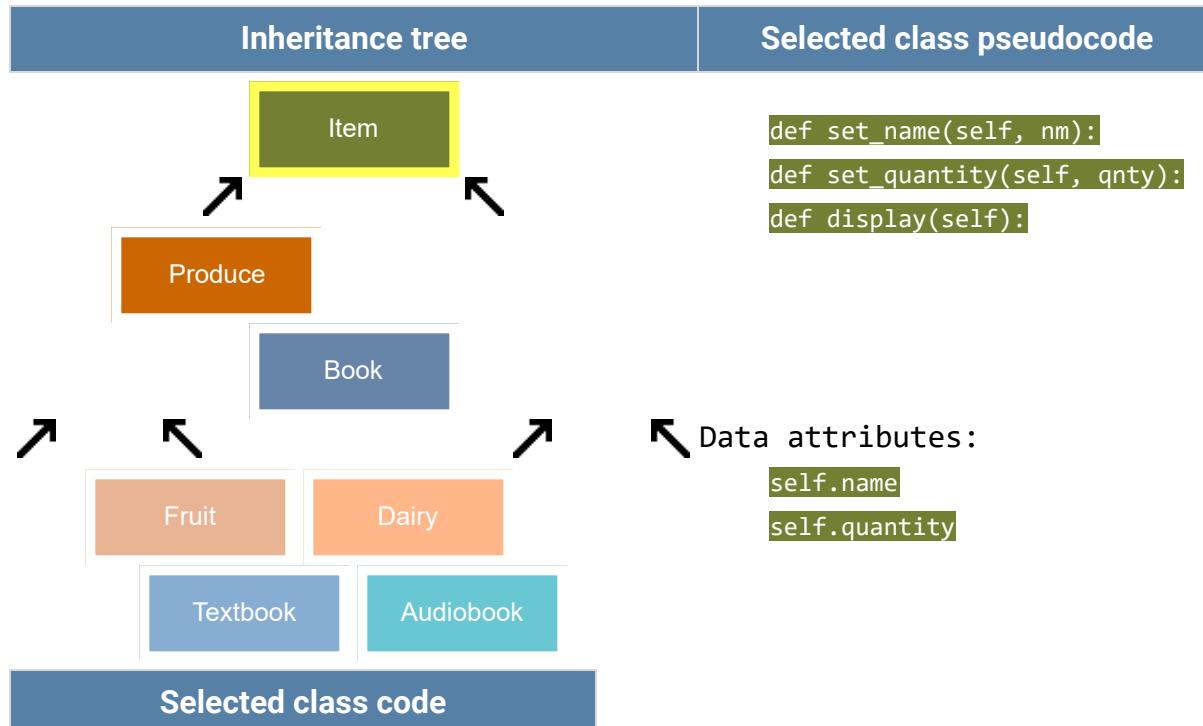
Various class derivation variations are possible:

- A derived class can itself serve as a base class for another class. In the earlier example, "class Fruit(Produce):" could be added.
- A class can serve as a base class for multiple derived classes. In the earlier example, "class Book(Item):" could be added.
- A class may be derived from multiple classes. For example, "class House(Dwelling, Property):" could be defined.

**PARTICIPATION ACTIVITY****13.1.4: Interactive inheritance tree.**

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

Click a class to see available methods and data for that class.



```
class Item:
    def __init__(self):
        self.name = ''
        self.quantity = 0

    def set_name(self, nm):
        self.name = nm

    def set_quantity(self, qnty):
        self.quantity = qnty

    def display(self):
        # print name, quantity
```

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

**PARTICIPATION  
ACTIVITY**

13.1.5: Derived classes basics.



- 1) A class that can serve as the basis for another class is called a \_\_\_\_\_ class.

**Check****Show answer**

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

- 2) Class "Dwelling" has the method open\_door(). Class "House" is derived from Dwelling and has the methods open\_window() and open\_basement(). After `h = House()` executes, how many different methods can `h` call, ignoring constructors?

**Check****Show answer****CHALLENGE  
ACTIVITY**

13.1.1: Basic inheritance.



Set `course_student.last_name` to Smith, `age_years` to 20, and `id_num` to 9999. Sample output for the given program:

Name: Smith, Age: 20, ID: 9999

```
1 class PersonData:  
2     def __init__(self):  
3         self.last_name = ''  
4         self.age_years = 0  
5     def set_name(self, user_name):  
6         self.last_name = user_name  
7     def set_age(self, num_years):  
8         self.age_years = num_years  
9     # Other parts omitted  
10    def print_all(self):  
11        output_str = 'Name: ' + self.last_name + ', Age: ' + str(self.age_years)  
12        return output_str
```

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

```
17  
18  
19 class StudentData(PersonData):  
20     def __init__(self):  
21         PersonData.__init__(self) # Call base class constructor
```

Run

(\*interfaces) For maximal simplicity and brevity in the example, we have used a set of methods that either set or return the value of an attribute. Such an interface to a class is commonly known as a getter/setter design pattern. In Python, the getter/setter interface is better replaced with simple attribute reference operations; e.g., instead of item1.set\_name('Hot Pockets'), use item1.name = 'Hot Pockets'.

(\*privacy) Python does have a way to enforce private variables through name mangling using double underscores in front of an identifier, e.g., self.\_\_data. A private variable is mostly used as a way to prevent name collisions in inheritance trees, instead of as a form of information hiding.

## 13.2 Accessing base class attributes

A derived class can access the attributes of all of its base classes via normal attribute reference operations. For example, item1.set\_name() might refer to the set\_name method attribute of a class from which item1 is derived. An attribute reference is resolved using a search procedure that first checks the instance's namespace, then the classes' namespace, then the namespaces of any base classes.

The search for an attribute continues all the way up the **inheritance tree**, which is the hierarchy of classes from a derived class to the final base class. Ex: Consider the following class structure in which Motorcycle is derived from MotorVehicle, which itself is derived from TransportMode.

Figure 13.2.1: Searching the inheritance tree for an attribute.

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

```

class TransportMode:
    def __init__(self, name, speed):
        self.name = name
        self.speed = speed

    def info(self):
        print('%s can go %d mph.' %
        (self.name, self.speed))

class MotorVehicle(TransportMode):
    def __init__(self, name, speed, mpg):
        TransportMode.__init__(self, name, speed)
        self.mpg = mpg
        self.fuel_gal = 0

    def add_fuel(self, amount):
        self.fuel_gal += amount

    def drive(self, distance):
        required_fuel = distance / self.mpg
        if self.fuel_gal < required_fuel:
            print('Not enough gas.')
        else:
            self.fuel_gal -= required_fuel
            print('%f gallons remaining.' %
            self.fuel_gal)

class MotorCycle(MotorVehicle):
    def __init__(self, name, speed, mpg):
        MotorVehicle.__init__(self, name, speed, mpg)

    def wheelie(self):
        print('That is too dangerous.')

scooter = MotorCycle('Vespa', 55, 40)
dirtbike = MotorCycle('KX450F', 80, 25)

scooter.info()
dirtbike.info()
choice = input('Select scooter (s) or dirtbike (d): ')
bike = scooter if (choice == 's') else
dirtbike

menu = '\nSelect add fuel(f), go(g),
wheelie(w), quit(q): '
command = input(menu)
while command != 'q':
    if command == 'f':
        fuel = int(input('Enter amount: '))
        bike.add_fuel(fuel)
    elif command == 'g':
        distance = int(input('Enter distance: '))
        bike.drive(distance)
    elif command == 'w':
        bike.wheelie()
    elif command == 'q':
        break
    else:
        print('Invalid command.')

```

Vespa can go 55 mph.  
KX450F can go 80 mph.  
Select scooter (s) or dirtbike (d): d

Select add fuel(f), go(g),
wheelie(w), quit(q): f  
Enter amount: 3

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
Select add fuel(f), go(g),
wheelie(w), quit(q): g  
Enter distance: 60  
0.600000 gallons remaining.

Select add fuel(f), go(g),
wheelie(w), quit(q): g  
Enter distance: 10  
0.200000 gallons remaining.

Select add fuel(f), go(g),
wheelie(w), quit(q): g  
Enter distance: 25  
Not enough gas.

Select add fuel(f), go(g),
wheelie(w), quit(q): w  
That is too dangerous.

Select add fuel(f), go(g),
wheelie(w), quit(q): q

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

```
command = input(menu)
```

The above illustrates a program with three levels of inheritance. The scooter and dirt bike variables are instances of the Motorcycle class at the bottom of the inheritance tree. Calling the add\_fuel() or drive() methods initiates a search, first in MotorCycle, and then in MotorVehicle. Calling the info() method defined at the top of the inheritance tree, as in **scooter.info()**, results in searching MotorCycle first, then MotorVehicle, and finally TransportMode.

zyDE 13.2.1: Extending the transportation modes class hierarchy.

Extend the above example with the following additional modes of transportation:

- Implement an Airplane class that is derived from TransportMode. Airplane should have the methods add\_fuel(), and fly(), and a data attribute num\_passengers.
  - Implement a JetPlane class that is derived from Airplane. Add some method to JetPlane of your own choosing, such as barrel\_roll() or immelman().

## Load default template...

```
1 class TransportMode:
2     def __init__(self, name, speed):
3         self.name = name
4         self.speed = speed
5
6
7     def info(self):
8         print('{} can go {} mph.'.format(se
9
10    class MotorVehicle(TransportMode):
11        def __init__(self, name, speed, mpg):
12            TransportMode.__init__(self, name,
13            self.mpg = mpg
14            self.fuel_gal = 0
15
16        def add_fuel(self, amount):
17            self.fuel_gal += amount
18
19        def drive(self, distance):
20            required_fuel = distance / self.mpg
```

Pre-enter any input for program run.

Run

## PARTICIPATION ACTIVITY

### 13.2.1: Searching for attributes in the inheritance tree.

- 1) "Inheritance tree" describes the hierarchy between base and derived classes.

Q True

False

- 2) Evaluating bike.wheelie() searches TransportMode, then MotorVehicle, then finally MotorCycle for the wheelie() method.

 True False

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

- 3) When adding a new derived class, a programmer has to change the base class as well.

 True False

## 13.3 Overriding class methods

A derived class may define a method having the same name as a method in the base class. Such a member function **overrides** the method of the base class. The following example shows the earlier Item/Produce example where the Produce class has its own display() method that overrides the display() method of the Item class.

Figure 13.3.1: Produce's display() function overrides Item's display() function.

Smith Cereal 9  
Apples 40 (Expires: May 5, 2012)

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

```

class Item:
    def __init__(self):
        self.name = ''
        self.quantity = 0

    def set_name(self, nm):
        self.name = nm

    def set_quantity(self, qnty):
        self.quantity = qnty

    def display(self):
        print(self.name, self.quantity)

class Produce(Item): # Derived from Item
    def __init__(self):
        Item.__init__(self) # Call base class
constructor
        self.expiration = ''

    def set_expiration(self, expir):
        self.expiration = expir

    def get_expiration(self):
        return self.expiration

    def display(self):
        print(self.name, self.quantity, end=' ')
        print(' (Expires: %s)' % self.expiration)

item1 = Item()
item1.set_name('Smith Cereal')
item1.set_quantity(9)
item1.display() # Will call Item's display()

item2 = Produce()
item2.set_name('Apples')
item2.set_quantity(40)
item2.set_expiration('May 5, 2012')
item2.display() # Will call Produce's display()

```

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

When the derived class defines the method being overwritten, that method is placed in the class's namespace. Because attribute references search the inheritance tree by starting with the derived class and then recursively searching base classes, the method called will always be the method defined in the instance's class.

A programmer will often want to *extend*, rather than replace, the base class method. The base class method can be explicitly called at the start of the method, with the derived class then performing additional operations:

Figure 13.3.2: Method calling overridden

method of base class.

```
class Produce(Item):
    # ...
    def display(self):
        Item.display(self)
        print(' (Expires: %s)' % self.expiration)
    # ...
```

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

Above, the `display()` method of `Produce` calls the `display()` method of `Item`, passing `self` as the first argument. Thus, when `Item`'s `display()` executes, the name and quantity instance attributes from the `Produce` instance are retrieved and printed.

PARTICIPATION  
ACTIVITY

13.3.1: Overriding base class methods.



Assume `my_item` is an instance of `Item`, and `my_produce` is an instance of `Produce`, with classes `Item` and `Produce` defined as above.

- 1) Will `my_item.display()` call the `display()` function for `Item` or for `Produce`?

**Check**

[Show answer](#)



- 2) Will `my_produce.display()` call the `display()` function for `Item` or for `Produce`?

**Check**

[Show answer](#)



- 3) Provide a statement within the `display()` method of the `Produce` class to call the `display()` method of `Produce`'s base class.

**Check**

[Show answer](#)

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019





- 4) If Produce did NOT have its own display() method defined, the display method of which class would be called in the following code? Type "ERROR" if appropriate.

```
p = Produce()  
p.display()
```

**Check****Show answer**

©zyBooks 11/11/19 21:02 569464

Diep Vu

UNIONCSC103OrhanFall2019

**CHALLENGE ACTIVITY****13.3.1: Basic derived class member override.**

Define a member method print\_all() for class PetData. Make use of the base class' print\_all() method. Sample output for the given program:

**Name: Fluffy****Age: 5****ID: 4444**

```
1 class AnimalData:  
2     def __init__(self):  
3         self.full_name = ''  
4         self.age_years = 0  
5  
6     def set_name(self, given_name):  
7         self.full_name = given_name  
8  
9     def set_age(self, num_years):  
10        self.age_years = num_years  
11  
12    # Other parts omitted  
13  
14    def print_all(self):  
15        print('Name:', self.full_name)  
16        print('Age:', self.age_years)  
17  
18  
19 class PetData(AnimalData):  
20     def __init__(self):  
21         AnimalData.__init__(self)
```

©zyBooks 11/11/19 21:02 569464

Diep Vu

UNIONCSC103OrhanFall2019

**Run**

## 13.4 Is-a versus has-a relationships

The concept of *inheritance* is often confused with *composition*. Composition is the idea that one object may be made up of other objects. For instance, a "mother" class can be made up of objects like "name" (possibly a string object), "children" (which may be a list of Child objects), etc. Defining that "mother" class does *not* involve inheritance, but rather just composing the sub-objects in the class.

Figure 13.4.1: Composition.

The 'has-a' relationship. A Mother object 'has-a' string object and 'has' child objects, but no inheritance is involved.

```
class Child:
    def __init__(self):
        self.name = ''
        self.birthdate = ''
        self.schoolname = ''
    # ...

class Mother:
    def __init__(self):
        self.name = ''
        self.birthdate = ''
        self.spouse_name = ''
        self.children = []
    # ...
```

In contrast, a programmer may note that a mother and a child are both a kind of person, and all persons have a name and birthdate. So the programmer may decide to better organize the program by defining a Person class, and then by creating the Mother and Child classes as derived from Person.

Figure 13.4.2: Inheritance.

The 'is-a' relationship. A Mother object 'is a' kind of Person. The Mother class thus inherits from the Person class. Likewise for the Child class.

```

class Person:
    def __init__(self):
        self.name = ''
        self.birthdate = ''
    # ...

class Child(Person):
    def __init__(self):
        Person.__init__(self)
        self.schoolname = ''
    # ...

class Mother(Person):
    def __init__(self):
        Person.__init__(self)
        self.spousename = ''
        self.children = []
    # ...

```

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

**PARTICIPATION  
ACTIVITY**

13.4.1: Is-a vs. has-a relationships.



Indicate whether the relationship of the everyday items is an is-a or has-a relationship.  
Derived classes and inheritance are related to is-a relationships, not has-a relationships.

1) Pear / Fruit



- Is-a
- Has-a

2) House / Door



- Is-a
- Has-a

3) Dog / Owner



- Is-an
- Has-an

4) Mug / Cup



- Is-a
- Has-a

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

# 13.5 Mixin classes and multiple inheritance

A class can inherit from more than one base class, a concept known as multiple inheritance. The derived class inherits all of the class attributes and methods of every base class.

PARTICIPATION  
ACTIVITY

13.5.1: Multiple inheritance.

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

## Animation captions:

1. Vampire bats are both winged animals and mammals.
2. VampireBat can access methods of Bird and Mammal.

A class can inherit from multiple base classes by specifying multiple items in the inheritance list:

Figure 13.5.1: Inheriting from multiple base classes.

```
class VampireBat(Bird, Mammal): # Inherit from Bird, Mammal classes  
    # ...
```

A common usage of multiple inheritance is extending the functionality of a class using mixins. Mixins are classes that provide some additional behavior, by "mixin in" new methods, but are not themselves meant to be instantiated.

Figure 13.5.2: Using mixins to extend a class' functionality with new methods.

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

```

class DrivingMixin(object):
    def drive(self, distance):
        # ...

    def change_tire(self):
        # ...

    def check_oil(self):
        # ...

class FlyingMixin(object):
    def fly(self, distance, altitude):
        # ...

    def roll(self):
        # ...

    def eject(self):
        # ...

class TransportMode(object):
    def __init__(self, name, speed):
        self.name = name
        self.speed = speed

    def display(self):
        print('%s can go %d mpg' % (self.name, self.speed))

class SemiTruck(TransportMode, DrivingMixin):
    def __init__(self, name, speed, cargo):
        TransportMode.__init__(self, name, speed)
        self.cargo = cargo

    def go(self, distance):
        self.drive(distance)
        # ...

class FlyingCar(TransportMode, FlyingMixin, DrivingMixin):
    def __init__(self, name, speed, max_altitude):
        TransportMode.__init__(self, name, speed)
        self.max_altitude = max_altitude

    def go(self, distance):
        self.fly(distance / 2, self.max_altitude)
        # ...
        self.drive(distance / 2)

s = SemiTruck('MacTruck', 85, 'Frozen beans')
f = FlyingCar('Jetson35K', 325, 15000)

s.go()
f.go()

```

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

Above, the DrivingMixin and FlyingMixin classes each define a set of methods. Any class can be derived from one or both of the mixins. Note that the resolution order by which the base classes are searched for an attribute is related to the order in which classes appear in the inheritance list parenthesis. The resolution order is from left to right, so in the FlyingCar class, TransportMode is

searched first, then FlyingMixin, and finally DrivingMixin. When using a mixin class, a programmer should be careful to either avoid clashing names, or carefully choose the order of classes in the inheritance list.

**PARTICIPATION  
ACTIVITY****13.5.2: Mixin classes and multiple inheritance.**

Consider the above program and class inheritance tree. Match the new class definitions with methods that would be inherited by instances of that class.

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

**class HoverCraft(DrivingMixin, FlyingMixin, TransportMode):****class Camel(TransportMode):****class Jet(TransportMode, FlyingMixin):****class Motorcycle(DrivingMixin, TransportMode):**

display()

display(), fly(), roll(), eject()

display(), drive(), change\_tire(),  
check\_oil()display(), drive(), fly(),  
change\_tire(), roll(), eject(),  
check\_oil()**Reset**

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

## 13.6 Testing your code: The unittest module

A critical part of software development is testing that a program behaves correctly. For large projects, changing code in one file or class may create new bugs in other parts of the program

that import or inherit from the changed code. Maintaining a **test suite**, or a set of repeatable tests, that can be run after changing the source code of a program is critical.

A programmer commonly performs **unit testing**, testing the individual components of a program, such as specific methods, class interfaces, data structures, and so on. The Python standard library **unittest** module implements unit testing functionality:

Figure 13.6.1: Unit testing with the unittest module.

©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

```
import unittest

# User-defined class
class Circle(object):
    def __init__(self, radius):
        self.radius = radius

    def compute_area(self):
        return 3.14 * self.radius**2

# Class to test Circle
class TestCircle(unittest.TestCase):
    def test_compute_area(self):
        c = Circle(0)

        self.assertEqual(c.compute_area(), 0.0)

        c = Circle(5)

        self.assertEqual(c.compute_area(), 78.5)

    def test_will_fail(self):
        c = Circle(5)

        self.assertLess(c.compute_area(), 0)

if __name__ == "__main__":
    unittest.main()
```

```
.F
=====
FAIL: test_will_fail (__main__.TestCircle)
-----
Traceback (most recent call last):
  File "area.py", line 23, in test_will_fail
    self.assertLess(c.compute_area(), 0)
AssertionError: 78.5 not less than 0

-----
Ran 2 tests in 0.000s
FAILED (failures=1)
```

The program above implements a unit test for the `Circle.compute_area()` method. A new class `TestCircle` is defined that inherits from `unittest.TestCase`. Methods within the `TestCircle` class that begin with "test\_" are the unit tests to be run. A unit test performs **assertions** to check if a computed value meets certain requirements. Above,

`self.assertEqual( c.compute_area(), 78.5 )` asserts that the result of `c.compute_area()` is equal to 78.5. If the assertion is not true, then an `AssertionError` will be raised and the current test will report as a failure. Executing the `unittest.main()` function begins the test process. After all tests have completed, a report is automatically printed.

Assertions for many types of relationships exist, for example `assertEqual()` tests equality, `assertIn` tests if a value is in a container, etc. The below table (from [docs.python.org](https://docs.python.org)) lists common assertions.

Table 13.6.1: Assertion methods.

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a,b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assert IsNot(a,b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assert IsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertAlmostEqual(a, b)</code>	<code>round(a - b, 7) == 0</code>
<code>assertGreater(a, b)</code>	<code>a &gt; b</code>
<code>assertGreaterEqual(a, b)</code>	<code>a &gt;= b</code>
<code>assertLess(a, b)</code>	<code>a &lt; b</code>
<code>assertLessEqual(a, b)</code>	<code>a &lt;= b</code>

©zyBooks 11/11/19 21:02 569464

Diep Vu

UNIONCSC103OrhanFall2019

©zyBooks 11/11/19 21:02 569464

Diep Vu

UNIONCSC103OrhanFall2019

## zyDE 13.6.1: Writing unit tests.

Complete the unit tests for testing the `evens()` and `odds()` methods. Each unit test either `odds()` or `evens()`, passing in a known array of values, and then testing the result to ensure only the correct values are in the array. (Hint: Use the `self.assertIn()` method.)

Load default template...Run

```
1 import unittest
2
3
4 def evens(numbers):
5     """Return the even values in numbers"""
6     return [i for i in numbers if (i%2 == 0)
7
8 def odds(numbers):
9     """Return the odd values in numbers"""
10    return [i for i in numbers if (i%2 == 1)
11
12
13 class TestNumbers(unittest.TestCase):
14     test_nums = [1, 3, 5, 6, 8, 2, 1]
15
16     def test_evens(self):
17         # Fill in the unit test.
18
19     def test_odds(self):
20         # Fill in the unit test
21
```

**PARTICIPATION ACTIVITY**

13.6.1: Unit testing.



- 1) What is the Python standard library module that allows the definition of unit tests?

**Check****Show answer**

- 2) Write an assertion that checks if c.valid is True.

```
def test_a(self):
    c = Widget()
    self.
```



- 3) Write an assertion that checks if c.sprockets is less than 5.



©zyBooks 11/11/19 21:02 569464  
Diep Vu  
UNIONCSC103OrhanFall2019

```
def test_b(self):  
    c = Widget()  
    self.
```

©zyBooks 11/11/19 21:02 569464

Diep Vu

UNIONCSC103OrhanFall2019

©zyBooks 11/11/19 21:02 569464

Diep Vu

UNIONCSC103OrhanFall2019