13 Inheritance

Figure 13.1 credit: modification of work "Don't stare...only sheep dogs stare...", by Bernard Spragg. NZ/Flickr, Public Domain

Chapter Outline

13.1 Inheritance basics
13.2 Attribute access
13.3 Methods
13.4 Hierarchical inheritance
13.5 Multiple inheritance and mixin classes
13.6 Chapter summary

Introduction

Real-world entities are often described in relation to other entities. Ex: A finch is a type of bird. Similarly, classes, which represent types of real-world entities, can be related to each other.

Inheritance describes the relationship in which one class is a type of another class. Classes within inheritance relationships can inherit attributes and methods from other classes without needing to redefine everything. Thus, inheritance in object-oriented programming reduces redundancy and promotes modularity.

13.1 Inheritance basics

Learning objectives

By the end of this section you should be able to

- Identify is-a and has-a relationships between classes.
- Differentiate between a subclass and a superclass.
- Create a superclass, subclass, and instances of each.

is-a vs has-a relationships

Classes are related to each other. An is-a relationship exists between a subclass and a superclass. Ex: A daffodil is a plant. A `Daffodil` class inherits from a superclass, `Plant`.

Is-a relationships can be confused with has-a relationships. A has-a relationship exists between a class that contains another class. Ex: An employee has a company-issued laptop. Note: The laptop is not an employee.

CHECKPOINT

is-a relationship between Employee and Developer

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-1-inheritance-basics)

CONCEPTS IN PRACTICE

Relationships between classes

1. What is the relationship between a `Doughnut` class and a `Pastry` class?
   a. is-a
   b. has-a

2. What is the relationship between a `Kitchen` class and a `Freezer` class?
   a. is-a
   b. has-a

3. A goalkeeper is a player. Goalkeeper is a _____class. Player is a _____class.
   a. super; sub
   b. sub; super

Inheritance in Python

Inheritance uses an is-a relationship to inherit a class from a superclass. The subclass inherits all the superclass's attributes and methods, and extends the superclass's functionality.

In Python, a subclass is created by including the superclass name in parentheses at the top of the subclass's definition:

```
class SuperClass:
    # SuperClass attributes and methods

class SubClass(SuperClass):
    # SubClass attributes and methods
```

CHECKPOINT

Using inheritance to create subclasses

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-1-inheritance-basics)

CONCEPTS IN PRACTICE

Creating subclasses

4. How is a `Daisy` class that inherits from the `Plant` class defined?
   a. `class Plant(Daisy):`
   b. `class Daisy(Plant):`
   c. `class Daisy:`
      `class Plant:`

5. Suppose a CarryOn class is inherited from a Luggage class. How is a CarryOn instance created?
   a. `small_bag = CarryOn()`
   b. `small_bag = Luggage(CarryOn)`
   c. `small_bag = CarryOn(Luggage)`

6. Given the following `SuperClass` and `SubClass`, which of the following can an instance of `SubClass` access?

```
class SuperClass():
  def func_1(self):
    print('Superclass function')

class SubClass(SuperClass):
  def func_2(self):
    print('Subclass function')
```

   a. `func_1()` only
   b. `func_2()` only
   c. `func_1()` and `func_2()`

7. Given the following `SuperClass` and `SubClass`, which of the following can an instance of SuperClass access?

```
class SuperClass():
  def func_1(self):
    print('Superclass function')

class SubClass(SuperClass):
  def func_2(self):
    print('Subclass function')
```

   a. `func_1()` only
   b. `func_2()` only
   c. `func_1()` and `func_2()`

ALTERNATIVE INHERITANCE TERMS

Python documentation for inheritance uses multiple terms to refer to the class that is inherited from and

the class that inherits. This book uses superclass/subclass throughout for consistency.

Class inherited from	Class that inherits
superclass	subclass
base class	derived class
parent class	child class

Table 13.1

TRY IT

Employee and Developer classes

Given the `Employee` class, create a `Developer` class that inherits from `Employee`. The `Developer` class has one method, `update_codebase()`, which prints `"Employee has updated the codebase"`. Then, use the `Developer` instance, `python_dev`, to call `print_company()` and `update_codebase()`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-1-inheritance-basics)

TRY IT

Polygon classes

Define three classes: `Polygon`, `Rectangle`, and `Square`:

- `Polygon` has the method `p_disp()`, which prints `"object is a Polygon"`.
- `Rectangle` inherits from `Polygon` and has the method `r_disp()`, which prints `"object is a Rectangle"`.
- `Square` inherits from `Rectangle` and has the method `s_disp()`, which prints `"object is a Square"`.

Create an instance of each class. Then, for each instance, call all the methods the instance has access to.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-1-inheritance-basics)

13.2 Attribute access

Learning objectives

By the end of this section you should be able to

- Implement a subclass that accesses inherited attributes from the superclass.
- Write a subclass's `__init__()` that inherits superclass instance attributes and creates new instance attributes.

Creating a simple subclass

Subclasses have access to the attributes inherited from the superclass. When the subclass's `__init__()` isn't explicitly defined, the superclass's `__init__()` method is called. Accessing both types of attributes uses the same syntax.

CHECKPOINT

Defining a simple subclass

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-2-attribute-access)

CONCEPTS IN PRACTICE

Using simple subclasses

1. Consider the `Employee` and `Developer` example above. What is the value of `dev_1.e_id`?
   a. `1`
   b. `2`
   c. `Error`

2. Consider the following example. Line 13 executes and `SubClass()` is called. Which line does control flow move to?

```
1  class SuperClass:
2    def __init__(self):
3      self.feat_1 = 1
4      self.feat_2 = ""
5
6    def bc_display(self):
7      print(f"Superclass: {self.feat_2}")
8
9  class SubClass(SuperClass):
10   def dc_display(self):
11     print(f"Subclass: {self.feat_2}")
12
13 dc_1 = SubClass()
```

   a. 2
   b. 10
   c. Error

3. Consider the following example. Which instance attribute(s) does `dc_1` have?

```
class SuperClass:
  def __init__(self):
    self.feat_1 = 1
    self.feat_2 = ""
```

```
  def bc_display(self):
    print(f"Superclass: {self.feat_2}")

class SubClass(SuperClass):
  def dc_display(self):
    print(f"Subclass: {self.feat_2}")

dc_1 = SubClass()
```

a. `feat_2`
b. `feat_1` and `feat_2`
c. None

Using __init__() to create and inherit instance attributes

A programmer often wants a subclass to have new instance attributes as well as those inherited from the superclass. Explicitly defining a subclass's `__init__()` involves defining instance attributes and assigning instance attributes inherited from the superclass.

CHECKPOINT

Defining __init__() in a subclass

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-2-attribute-access)

CONCEPTS IN PRACTICE

Accessing a subclass's attributes

Consider the `Employee` and `Developer` example code:

```
class Employee:
  count = 0
  def __init__(self):
    Employee.count += 1
    self.e_id = Employee.count
    self.hire_year = 2023

  def emp_display(self):
    print(f"Employee {self.e_id} hired in {self.hire_year}")

class Developer(Employee):
  def __init__(self):
    Employee.count += 1
    self.e_id = Employee.count
```

```
        self.hire_year = 2023
        self.lang_xp = ["Python", "C++", "Java"]

      def dev_display(self):
        print(f"Proficient in {self.lang_xp}")

    emp_1 = Employee()
    dev_1 = Developer()
```

4. What would be the output of `dev_1.dev_display()`?
   a. `Employee 2 hired in 2023`
   b. `Proficient in ['Python', 'C++', 'Java']`
   c. `Error`

5. What would be the output of `emp_1.dev_display()`?
   a. `Employee 1 hired in 2023`
   b. `Proficient in ['Python', 'C++', 'Java']`
   c. `Error`

6. Suppose `dev_display()` should be modified to display the developer's ID along with their proficiencies. Ex: `dev_1.dev_display()` would output `Employee 2 proficient in ['Python', 'C++', 'Java']`. Which is the appropriate new `print()` call in `dev_display()`?
   a. `print(f"Employee {self.e_id} proficient in {self.lang_xp}")`
   b. `print(f"Employee {self.Employee.e_id} proficient in {self.lang_xp}")`
   c. `print(f"Employee 2 proficient in {self.lang_xp}")`

TRY IT

Creating a subclass with an instance attribute

Given a class `Dessert`, create a class, `Cupcake`, inherited from `Dessert`. `Cupcake` class methods:

- `__init__(self)`: initializes inherited instance attribute ingredients with ["butter", "sugar", "eggs", "flour"], and initializes instance attribute frosting with "buttercream"
- `display(self)`: prints a cupcake's ingredients and frosting

Then call the `display()` method on a new `Cupcake` object. The output should match:

```
    Made with ["butter", "sugar", "eggs", "flour"] and topped with buttercream
frosting
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-2-attribute-access)

13.3 Methods

Learning objectives

By the end of this section you should be able to

- Write overridden methods to change behavior of inherited methods.
- Use `super()` to access superclass methods.
- Identify applications of polymorphism in method and function use.

Overriding methods

Sometimes a programmer wants to change the functions a subclass inherits. `Mint` is a subclass that has the same functionality as `Plant`, except for one function. A subclass can **override** a superclass method by defining a method with the same name as the superclass method.

CHECKPOINT

Overriding a superclass method

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-3-methods)

CONCEPTS IN PRACTICE

Overriding methods

1. Suppose a programmer inherits the `ContractTax` class from class `Tax` and wants to override `Tax`'s `calc_tax()`. What should the programmer do?
   a. Define another `calc_tax()` method in `Tax`.
   b. Define a `calc_tax()` method in `ContractTax`.
   c. Define a function that takes a `ContractTax` instance.

2. Which is the error in the program that attempts to override `calc_tax()` for `ContractTaxDE`?

```
class Tax:
  def calc_tax(self):
    print('Calculating tax')

class ContractTax(Tax):
  def calc_tax(self):
    print('Calculating contract tax')

class ContractTaxDE(ContractTax):
  def calc_tax():
    print('Calculating German contract tax')

my_tax = ContractTaxDE()
my_tax.calc_tax()
```

   a. `ContractTaxDE` must inherit from `Tax`, not `ContractTax`.

b. `ContractTaxDE's` definition of `calc_tax()` is missing a parameter.
c. `ContractTaxDE` can't override `calc_tax()` since `ContractTax` already has.

3. The following program doesn't override `calc_tax()`. Why?

```
class Tax:
  def calc_tax(self):
    print('Calculating tax')
    total = 0 # To replace with calculation
    return total

class ContractTax:
  def calc_tax(self):
    print('Calculating contract tax')
    total = 0 # To replace with calculation
    return total

my_tax = ContractTax()
my_tax.calc_tax()
```

a. An overridden method cannot return a value.
b. Tax doesn't specify `calc_tax()` can be overridden.
c. `ContractTax` isn't inherited from Tax.

super()

`super()` is a special method that provides a temporary superclass object instance for a subclass to use. `super()` is commonly used to call superclass methods from a subclass. `super()` is commonly used in a subclass's `__init__()` to assign inherited instance attributes. Ex: `super().__init__()`.

CHECKPOINT

Using super() to call the superclass __init__() method

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-3-methods)

CONCEPTS IN PRACTICE

Using super()

Consider the following program.

```
1 class Polygon:
2       def __init__(self, num_sides=3):
3         self.num_sides = num_sides
4
5     class Rectangle(Polygon):
```

```
 6       def __init__(self, ln=1, wd=1):
 7         super().__init__(4)
 8         self.length = ln
 9         self.width = wd
10
11     class Square(Rectangle):
12       def __init__(self, side=1):
13         super().__init__(side, side)
14         self.side = side
15
16     sq_1 = Square(5)
17     print(sq_1.num_sides)
```

4. Line 16 executes and `Square`'s `__init__()` is called on line 12. Line 13 executes and the superclass's `__init__()` is called. Which line does control flow move to next?
   a. 2
   b. 6
   c. 14

5. The next line executes. Which line does control flow move to next?
   a. 2
   b. 6
   c. 14

6. The method call returns. Lines 8 and 9 execute to initialize length and width, and `Rectangle's __init__()` returns. Which line does control flow move to next?
   a. 12
   b. 14
   c. 17

7. `Square's __init__()` returns and control flow moves to line 17. What is the output?
   a. 3
   b. 4
   c. 5

Polymorphism

Polymorphism is the concept of having many forms. In programming, a single name can be used for multiple functionalities. Within inheritance, polymorphism is the basis of method overriding, as multiple methods have the same name.

EXAMPLE 13.1

Polymorphism and inheritance

The name `display()` maps to multiple methods. The class type of the calling object determines which `display()` method is executed.

```
class Plant:
  def display(self):
    print("I'm a plant")

class Mint(Plant):
  def display(self):
    print("I'm a mint")

class Lavender(Mint):
  def display(self):
    print("I'm a lavender")

mint_1 = Mint()
mint_1.display()

lavender_1 = Lavender()
lavender_1.display()
```

The code's output is:

```
I'm a mint
I'm a lavender
```

Polymorphism can also be used with methods of unrelated classes. The class type of the calling object determines the method executed.

EXAMPLE 13.2

Polymorphism and methods

`Tax` and `ContractTax` are unrelated classes that each define `calc_tax()`. `calc_tax()` isn't overridden as `ContractTax` isn't inherited from `Tax`.

```
class Tax:
  def __init__(self, value):
    self.value = value

  def calc_tax(self):
    print("Calculating tax")
    total = 0.10 * self.value  # To replace with calculation
    return total
```

```
class ContractTax:
  def __init__(self, value):
    self.value = value

  def calc_tax(self):
      print("Calculating contracts tax")
      total = 0.15 * self.value  # To replace with calculation
      return total

my_tax = ContractTax(value=1000)  # Replace 1000 with any value
result = my_tax.calc_tax()
print(f"Total tax: ${result}")
```

The code's output is:

```
Calculating contracts tax
Total tax: $150.00
```

Polymorphism allows methods to be called with different parameter types. Many built-in operators and functions have this utility.

EXAMPLE 13.3

Polymorphism and functions

`len()` can be called with multiple types, including lists and strings.

```
tree_list = ["ash", "hazel", "oak", "yew"]
tree_1 = tree_list[0]
print(len(tree_list))
print(len(tree_1))
```

The code's output is:

```
4
3
```

CONCEPTS IN PRACTICE

Polymorphism in practice

8. Consider the example. What is the output?

```
class Polygon:
  def print_type(self):
    print("Polygon type")

class Rectangle(Polygon):
  def print_type(self):
    print("Rectangle type")

class Square(Rectangle):
  def print_type(self):
    print("Square type")

sq_1 = Square()
sq_1.print_type()
```

a. `Polygon type`
b. `Rectangle type`
c. `Square type`

9. Which is an example of polymorphism for the multiplication operator?

a.
```
x = 3 * 4
y = 3 * "la"
```
b.
```
x = 3 * 4
y = str(x)
```
c.
```
x = 3 * 4
y = x * x
```

TRY IT

Overriding methods

Given the `Pet` class, create a `Bird` class inherited from `Pet` and a `Finch` class inherited from `Bird`. Override the `display()` method in `Bird` and `Finch` such that the program output is:

```
Pet type: Bird
Bird type: Finch
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-3-methods)

TRY IT

Using super()

Given the `Employee` class, create a `Developer` class inherited from `Employee` with methods `__init__()` and `print_info()` such that:

`__init__()`

- Uses `super()` to call `Employee`'s `__init__()` to initialize `e_id` and `hire_year`.
- Assigns `lang_xp` with list parameter `lang_xp`.

`print_info()`

- Uses `super()` to print `e_id` and `hire_year`.
- Prints `"Language(s):"` followed by `lang_xp`.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-3-methods)

13.4 Hierarchical inheritance

Learning objectives

By the end of this section you should be able to

- Label relationships between classes as types of inheritance.
- Construct classes that form hierarchical inheritance.

Hierarchical inheritance basics

Hierarchical inheritance is a type of inheritance in which multiple classes inherit from a single superclass. **Multilevel inheritance** is a type of inheritance in which a subclass becomes the superclass for another class. Combining hierarchical and multilevel inheritance creates a tree-like organization of classes.

CHECKPOINT

Hierarchical organization and types of inheritance

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-4-hierarchical-inheritance)

CONCEPTS IN PRACTICE

Hierarchical organization

1. Which is an example of hierarchical inheritance?
   a. Class B inherits from Class A
      Class C inherits from Class A
   b. Class B inherits from Class A
      Class C inherits from Class B
   c. Class B inherits from Class A

Class C inherits from Class D

2. Which group of classes is hierarchical inheritance appropriate for?
   a. `Cat`, `Dog`, `Bird`
   b. `Employee`, `Developer`, `SalesRep`
   c. `Dessert`, `BakedGood`, `ApplePie`

Implementing hierarchical inheritance

Multiple classes can inherit from a single class by simply including the superclass name in each subclass definition.

EXAMPLE 13.4

Choir members

```
class ChoirMember:
  def display(self):
    print("Current choir member")

class Soprano(ChoirMember):
  def display(self):
    super().display()
    print("Part: Soprano")

class Soprano1(Soprano):
  def display(self):
    super().display()
    print("Division: Soprano 1")

class Alto(ChoirMember):
  def display(self):
    super().display()
    print("Part: Alto")

class Tenor(ChoirMember):
  def display(self):
    super().display()
    print("Part: Tenor")

class Bass(ChoirMember):
  def display(self):
    super().display()
    print("Part: Bass")

mem_10 = Alto()
```

```
mem_13 = Tenor()
mem_15 = Soprano1()

mem_10.display()
print()
mem_13.display()
print()
mem_15.display()
```

The code's output is:

```
Current choir member
Part: Alto

Current choir member
Part: Tenor

Current choir member
Part: Soprano
Division: Soprano 1
```

CONCEPTS IN PRACTICE

Implementing hierarchical inheritance

Consider the program:

```
class A:
  def __init__(self, a_attr=0):
    self.a_attr = a_attr

class B(A):
  def __init__(self, a_attr=0, b_attr=0):
    super().__init__(a_attr)
    self.b_attr = b_attr

class C(A):
  def __init__(self, a_attr=0, c_attr=0):
    super().__init__(a_attr)
    self.c_attr = c_attr

class D(B):
  def __init__(self, a_attr=0, b_attr=0, d_attr=0):
```

```
        super().__init__(a_attr, b_attr)
        self.d_attr = d_attr

    b_inst = B(2)
    c_inst = C(c_attr=4)
    d_inst = D(6, 7)
```

3. What is the value of `b_inst.b_attr`?
   a. `0`
   b. `2`
   c. `Error`

4. Which attributes does `c_inst` have access to?
   a. `a_attr`
   b. `a_attr`, `c_attr`
   c. `c_attr`

5. Which attributes does `d_inst` have access to?
   a. `b_attr`, `d_attr`
   b. `a_attr`, `b_attr`, `d_attr`
   c. `d_attr`

TRY IT

Overriding methods

Define three classes: `Instrument`, `Woodwind`, and `String`.

- `Instrument` has instance attribute `owner`, with default value of "unknown".
- `Woodwind` inherits from `Instrument` and has instance attribute `material` with default value of "wood".
- `String` inherits from `Instrument` and has instance attribute `num_strings`, with default value of 4.

The output should match:

```
This flute belongs to unknown and is made of silver
This cello belongs to Bea and has 4 strings
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-4-hierarchical-inheritance)

13.5 Multiple inheritance and mixin classes

Learning objectives

By the end of this section you should be able to

- Construct a class that inherits from multiple superclasses.
- Identify the diamond problem within multiple inheritance.
- Construct a mixin class to add functionality to a subclass.

Multiple inheritance basics

Multiple inheritance is a type of inheritance in which one class inherits from multiple classes. A class inherited from multiple classes has all superclasses listed in the class definition inheritance list. Ex: `class SubClass(SuperClass_1, SuperClass_2)`.

CHECKPOINT

Multiple inheritance organization

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-5-multiple-inheritance-and-mixin-classes)

CONCEPTS IN PRACTICE

Implementing multiple inheritance

Consider the program:

```
class A:
  def __init__(self, a_attr=2):
    self.a_attr = a_attr

class B:
  def __init__(self, b_attr=4):
    self.b_attr = b_attr

class C(A, B):
  def __init__(self, a_attr=5, b_attr=10, c_attr=20):
    A.__init__(self, a_attr)
    B.__init__(self, b_attr)
    self.c_attr = c_attr

b_inst = B(2)
c_inst = C(1, 2)
```

1. What is the value of `c_inst.a_attr`?
   a. 1
   b. 5

c. `Error`

2. What is the value of `c_inst.c_attr`?
   a. `2`
   b. `20`
   c. `Error`

3. What is the value of `b_inst.a_attr`?
   a. `2`
   b. `4`
   c. `Error`

The diamond problem and mixin classes

Multiple inheritance should be implemented with care. The **diamond problem** occurs when a class inherits from multiple classes that share a common superclass. Ex: `Dessert` and `BakedGood` both inherit from `Food`, and `ApplePie` inherits from `Dessert` and `BakedGood`.

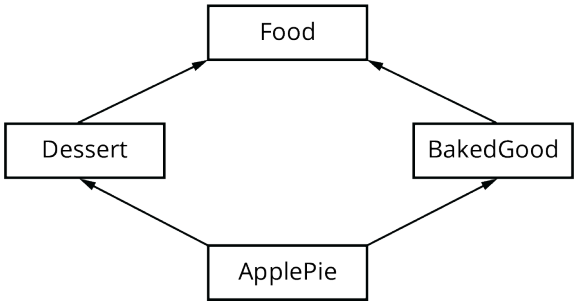

Figure 13.2 The diamond problem. If both `Dessert` and BakedGood override a `Food` method, the overridden `Food` method that `ApplePie` inherits is ambiguous. Thus, diamond shaped inheritance should be avoided.

Mixin classes promote modularity and can remove the diamond problem. A mixin class:

- Has no superclass
- Has attributes and methods to be added to a subclass
- Isn't instantiated (Ex: Given `MyMixin` class, `my_inst = MyMixin()` should never be used.)

Mixin classes are used in multiple inheritance to add functionality to a subclass without adding inheritance concerns.

CONCEPTS IN PRACTICE

Creating a mixin class

The following code isn't correct, as not all plants are carnivorous. Follow the programmer through the programmer's edits to improve the program. Note: `Rose`, `Pitcher`, and `VenusFlyTrap` represent plants that all photosynthesize. `Pitcher` and `VenusFlyTrap` represent plants that are also carnivorous and can eat.

A pass statement is used in Python to indicate that code is to be written later and prevents certain errors that would result if no code was written or a comment was used instead.

```
class Plant:
  def photosynth(self):
    print("Photosynthesizing")

  def eat(self):
    print("Eating")

class Rose(Plant):
  pass

class Pitcher(Plant):
  pass

class VenusFlyTrap(Plant):
  pass
```

4. Which edit is appropriate?

a. Move `eat()` to a different class.
b. Remove `Plant` as a superclass.

5. The programmer edits the code as follows. How can the program be improved?

```
class Plant:
  def photosynth(self):
    print("Photosynthesizing")

class Rose(Plant):
  pass

class Pitcher(Plant):
  def eat(self):
    print("Eating")

class VenusFlyTrap(Plant)
  def eat(self):
    print("Eating")
```

a. Move `photosynth()` into `Rose`.
b. Remove redundancy of `eat()`.

6. The programmer edits the code to create a mixin class containing `eat()`. Which class name, replacing X, is most appropriate for the mixin?

```
class Plant:
  def photosynth(self):
    print("Photosynthesizing")
```

```
class X:
  def eat(self):
    print("Eating")

class Rose(Plant):
  pass

class Pitcher(Plant, X):
  pass

class VenusFlyTrap(Plant, X)
  pass
```

a. `CarnivClass`
b. `CarnivMixin`

TRY IT

Fixing the diamond problem with a mixin class

The program below represents multiple inheritance and has the diamond problem. Edit the program to use a mixin class, `OnlineMixin`. `OnlineMixin`:

- Replaces `OnlineShop`.
- Is a superclass of `FoodFast`.
- Has one method, `process_online()`, which prints `"Connecting to server"` and is called by `FoodFast`'s `process_order()`.

The output should match:

```
Processing meal order
Connecting to server
Enjoy your FoodFast order
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/13-5-multiple-inheritance-and-mixin-classes)

13.6 Chapter summary

Highlights from this chapter include:

- Inheritance describes an is-a relationship between classes. One class, the subclass, inherits from another class, the superclass.
- Subclasses can access inherited attributes and methods directly.
- Subclasses can override superclass methods to change or add functionality.
- `super()` allows a subclass to access the methods of the superclass.

- Polymorphism describes a single representation for multiple forms and is applied in Python to define multiple methods with the same name and allow the same method to take different arguments.
- Hierarchical inheritance is a type of inheritance in which multiple classes inherit from a single superclass.
- Multiple inheritance is a type of inheritance in which one class inherits from multiple classes.
- Mixin classes are used in multiple inheritance to add functionality to a subclass without adding inheritance concerns.

At this point, you should be able to write subclasses that inherit instance attributes and methods, and subclasses that have unique attributes and overridden methods. You should also be able to create hierarchical inheritance relationships and multiple inheritance relationships between classes.

Task	Example

Define a subclass	`class SuperClass:`
	`  pass`
	
	`class SubClass(SuperClass):`
	`  pass`

Define a subclass's `__init__()` using `super()`	`class SubClass(SuperClass):`
	`  def __init__(self):`
	`  super().__init__() # Calls superclass __init__()`
	`  # Initialize subclass instance attributes`

Override a superclass method	`class SuperClass:`
	`  def display(self):`
	`  print('Superclass method')`
	
	`class SubClass(SuperClass):`
	`  def display(self): # Same name as superclass method`
	`  print('Subclass method')`

Table 13.2 Chapter 13 reference.

Task	Example

Implement hierarchical inheritance	`class SuperClass:`
	`  def display(self):`
	`  print('Superclass method')`
	
	`class SubClass1(SuperClass):`
	`  def display(self):`
	`  print('Subclass 1 method ')`
	`class SubClass2(SuperClass):`
	`  def display(self):`
	`  print('Subclass 2 method')`
	
	`class SubClass3(SuperClass):`
	`  def display(self):`
	`  print('Subclass 3 method')`

Implement multiple inheritance	`class SuperClass1:`
	`  pass`
	
	`class SuperClass2:`
	`  pass`
	
	`class SubClass(SuperClass1, SuperClass2):`
	`  pass`

Table 13.2 Chapter 13 reference.