11

Classes

Figure 11.1 credit: modification of work "Fresh, bright apples newly picked", by Colorado State University Extension/Flickr, Public Domain

Chapter Outline

Introduction

A programmer can model real-world entities as objects for better program design and organization. A class defines a type of object with attributes and methods. Many instances of a class type can be created to represent multiple objects in a program.

Classes promote reusability. Classes add benefits like data abstraction and encapsulation, which organize code for better usability and extensibility.

11.1 Object-oriented programming basics

Learning objectives

By the end of this section you should be able to

- Describe the paradigm of object-oriented programming (OOP).
- Describe the concepts of encapsulation and abstraction as they relate to OOP, and identify the value of each concept.

Grouping into objects

Object-oriented programming (OOP) is a style of programming that groups related fields, or data members,

and procedures into objects. Real-world entities are modeled as individual objects that interact with each other. Ex: A social media account can follow other accounts, and accounts can send messages to each other. An account can be modeled as an object in a program.

CHECKPOINT

Using objects to model social media

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-1-object-oriented-programming-basics)

CONCEPTS IN PRACTICE

OOP

1. Consider the example above. Which is a field in Ellis's 12/18/23 post?
   a. Ellis's followers list
   b. Ellis's username

2. What does an object typically model in object-oriented programming?
   a. program code
   b. real-world entity

Encapsulation

Encapsulation is a key concept in OOP that involves wrapping data and procedures that operate on that data into a single unit. Access to the unit's data is restricted to prevent other units from directly modifying the data. Ex: A ticket website manages all transactions for a concert, keeping track of tickets sold and tickets still available to avoid accidental overbooking.

CHECKPOINT

Encapsulating concert ticket information

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-1-object-oriented-programming-basics)

CONCEPTS IN PRACTICE

Encapsulation

3. Consider the example above. Suppose the venue has to be changed due to inclement weather. How should a programmer change the website object to allow changes to the Venue field?
   a. Add a procedure to change the field.
   b. Allow users direct access to the field.
   c. No valid way to allow changes to the field.

4. Which is a benefit of encapsulation when developing complex programs?

a. All objects can easily access and modify each other's data.
b. Each object's data is restricted for intentional access.

Abstraction

Abstraction is a key concept in OOP in which a unit's inner workings are hidden from users and other units that don't need to know the inner workings. Ex: A driver doesn't usually need to know their car engine's exact, numerical temperature. So the car has a gauge to display whether the engine temperature is within an appropriate range.

CHECKPOINT

Abstracting data in a countdown calculator

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-1-object-oriented-programming-basics)

CONCEPTS IN PRACTICE

Abstraction

5. Consider the car example in the paragraph above. Suppose the designer decided to remove the engine temperature indicator. Would this be a good use of abstraction?
   a. yes
   b. no

6. Which is a benefit of abstraction?
   a. improved view of information
   b. high visibility of information

11.2 Classes and instances

Learning objectives

By the end of this section you should be able to

- Create a class with instance attributes, class attributes, and the `__init__()` method.
- Use a class definition to create class instances to represent objects.

Classes and instances

In the previous section, a real-world entity, like a person's social media profile, was modeled as a single object. How could a programmer develop a software system that manages millions of profiles? A blueprint that defines the fields and procedures of a profile would be crucial.

In a Python program, a **class** defines a type of object with attributes (fields) and methods (procedures). A class is a blueprint for creating objects. Individual objects created of the class type are called **instances**.

CHECKPOINT

Representing a coffee order with a class

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-2-classes-and-instances)

CONCEPTS IN PRACTICE

Classes and instances

Consider the example below:

```
class Cat:
  def __init__(self):
    self.name = 'Kitty'
    self.breed = 'domestic short hair'
    self.age = 1
  def print_info(self):
    print(self.name, 'is a ', self.age, 'yr old', self.breed)

pet_1 = Cat()
pet_2 = Cat()
```

1. What is the name of the class?
   a. `Cat`
   b. `class Cat`
   c. `self`

2. Which of the following is an instance of the `Cat` class?
   a. `__init__`
   b. `name`
   c. `pet_1`

3. Which of the following is an attribute?
   a. `breed`
   b. `pet_2`
   c. `print_info`

4. Suppose the programmer wanted to change the class to represent a pet cat. Which is the appropriate name that follows PEP 8 recommendations?
   a. `petcat`
   b. `pet_cat`
   c. `PetCat`

Creating instances with __init__()

`___init___()` is a special method that is called every time a new instance of a class is created. **self** refers to the instance of a class and is used in class methods to access the specific instance that called the method. `__init__()` uses `self` to define and initialize the instance's attributes.

CHECKPOINT

Creating multiple coffee orders and changing attributes

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-2-classes-and-instances)

CONCEPTS IN PRACTICE

instances and __init__()

Consider the example below:

```
 1 class Rectangle:
 2       def __init__(self):
 3         self.length = 1
 4         self.width = 1
 5       def area(self):
 6         return self.length * self.width
 7
 8     room_1 = Rectangle()
 9     room_1.length = 10
10     room_1.width = 15
11     print("Room 1's area:", room_1.area())
12     room_3 = Rectangle()
13     room_3.length = 12
14     room_3.width = 14
15     print("Room 3's area:", room_3.area())
```

5. How many times is `__init__()` called?
   a. 1
   b. 2
   c. 3

6. When line 11 executes, execution flow moves to line 5. What does `self` represent on line 5?
   a. the area of `room_1`
   b. the instance `room_1`
   c. the `Rectangle` class

7. Which line initializes the instance attribute length?
   a. 3
   b. 6
   c. 9

8. Suppose line 2 is changed to `def __init__():`. What would `room_1's` attributes be initialized to?
   a. `length = 0, width = 0`
   b. `length = 1, width = 1`
   c. `Error`

Instance attributes vs. class attributes

The attributes shown so far have been instance attributes. An **instance attribute** is a variable that is unique to each instance of a class and is accessed using the format `instance_name.attribute_name`. Another type of attribute, a **class attribute**, belongs to the class and is shared by all class instances. Class attributes are accessed using the format `class_name.attribute_name`.

CHECKPOINT

Using class attributes for shared coffee order information

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-2-classes-and-instances)

CONCEPTS IN PRACTICE

Instances and class attributes

Consider the example above.

9. Which is an instance attribute?
   a. `loc`
   b. `order_id`
   c. `order_3`

10. Suppose the line `order_1.cup_size = 8` is added before `order_3.print_order()`. What is the new output?
   a. `Cafe Coffee Order 3 : 8 oz`
   b. `Cafe Coffee Order 3 : 16 oz`
   c. `Error`

11. Suppose the line `CoffeeOrder.loc = 'Caffeine Cafe'` is added before `order_3.print_order()`. What is the new output?
   a. `Caffeine Cafe Order 3 : 16 oz`
   b. `Cafe Coffee Order 3 : 16 oz`
   c. `Error`

12. Suppose the line `self.cls_id = 5` is added to the end of `__init__()`'s definition. What is the new output?
   a. `Cafe Coffee Order 5 : 16 oz`
   b. `Cafe Coffee Order 3 : 16 oz`
   c. `Error`

TRY IT

Creating a class for an airline's flight tickets

Write a class, `FlightTicket`, as described below. Default values follow the attributes. Then create a flight ticket and assign each instance attribute with values read from input.

Instance attributes:

- flight_num: 1
- airport: JFK
- gate: T1-1
- time: 8:00
- seat: 1A
- passenger: unknown

Class attributes:

- airline: Oceanic Airlines
- airline_code: OA

Method:

- `__init__()`: initializes the instance attributes
- `print_info()`: prints ticket information (provided in template)

Given input:

```
2121
KEF
D22B
11:45
12B
Jules Laurent
```

The output is:

```
Passenger Jules Laurent departs on flight # 2121 at 11:45 from KEF D22B in seat
12B
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-2-classes-and-instances)

TRY IT

Creating a class for fantasy books

Write a class, Book, as described below. Then create two instances and assign each instance attribute with values read from input.

Instance attributes:

- title: ''
- author: ''
- year: 0
- pages: 0

Class attribute:

- imprint: Fantasy Tomes

Method:

- `__init__()`: initializes the instance attributes
- `print_info()`: prints book information (provided in template)

Given input:

```
Lord of the Bracelets
Blake R. R. Brown
1999
423
A Match of Thrones
Terry R. R. Thomas
2020
761
```

The output is:

```
Lord of the Bracelets by Blake R. R. Brown published by Fantasy Tomes
in 1999 with 423 pages
A Match of Thrones by Terry R. R. Thomas published by Fantasy Tomes
in 2020 with 761 pages
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-2-classes-and-instances)

11.3 Instance methods

Learning objectives

By the end of this section you should be able to

- Create and implement `__init__()` with multiple parameters including default parameter values.
- Describe what information an instance method has access to and can modify.

More about __init__()

In Python, `__init__()` is the special method that creates instances. `__init__()` must have the calling instance, `self`, as the first parameter and can have any number of other parameters with or without default parameter values.

CHECKPOINT

Creating patient vital signs instances

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-3-instance-methods)

CONCEPTS IN PRACTICE

Defining and using __init__() with parameters

Consider the example above.

1. Suppose the programmer wanted to make blood pressure a required parameter in `__init__()`. Which is the correct `__init__()` method header?
   a. `def __init__(p_id, bp, self, tmp=-1.0, hr=-1, rr=-1):`
   b. `def __init__(self, p_id, bp, tmp=-1.0, hr=-1, rr=-1):`
   c. `def __init__(self, p_id, bp=[-1,-1], tmp=-1.0, hr=-1, rr=-1):`

2. Which is a correct call to `__init__()` to create an instance with `p_id=5241`?
   a. `patient_10 = Vitals(self, 5241)`
   b. `patient_10 = Vitals(5241)`
   c. both

3. Suppose another `__init__()` definition is added after the first with the header as follows:

   ```
   def __init__(self, p_id, bp, tmp, hr, rr)
   ```

   What is the impact on the program?
   a. no change.
   b. Second `__init__()` definition produces an error.
   c. First two `__init__()` calls produce an error.

Instance methods

An **instance method** is used to access and modify instance attributes as well as class attributes. All methods shown so far, and most methods defined in a class definition, are instance methods.

EXAMPLE 11.1

Instance methods are often used to get and set instance information

```
class ProductionCar:
  def __init__(self, make, model, year, max_mph = 0.0):
    self.make = make
    self.model = model
    self.year = year
```

```
      self.max_mph = max_mph

    def max_kmh(self):
      return self.max_mph * 1.609344

    def update_max(self, speed):
      self.max_mph = speed

  car_1 = ProductionCar('McLaren', 'Speedtail', 2020) # car_1.max_mph is 0.0
  car_1.update_max(250.0) # car_1.max_mph is 250.0
  print(car_1.make, car_1.model, 'reaches', car_1.max_mph, 'mph (',
    car_1.max_kmh(), 'km/h)') # Prints McLaren Speedtail reaches 250.0 mph (402.336
km/h)
```

CONCEPTS IN PRACTICE

CoffeeOrder instance methods

Consider the example below:

```
  class CoffeeOrder:
    loc = 'Cafe Coffee'
    cls_id = 1

    def __init__(self, size=16, milk=False, sugar=False):
      self.order_id = CoffeeOrder.cls_id
      self.cup_size = size
      self.with_milk = milk
      self.with_sugar = sugar
      CoffeeOrder.cls_id += 1

    def change(self, milk, sugar):
      self.with_milk = milk
      self.with_sugar = sugar

    def print_order(self):
      print(CoffeeOrder.loc,'Order', self.order_id, ':', self.cup_size, 'oz')
      if self.with_milk:
        print('\twith milk')
      if self.with_sugar:
        print('\twith sugar')

  order_1 = CoffeeOrder(8)
  order_2 = CoffeeOrder(8, True, False)
```

```
order_1.change(False, True)
```

4. What does `order_1` represent at the end of the program?
   a. 8 oz coffee
   b. 8 oz coffee with milk
   c. 8 oz coffee with sugar

5. Why is `change()` an instance method?
   a. can change attributes
   b. has multiple parameters
   c. can change an instance

6. Can `print_order()` be unindented?
   a. yes
   b. no

TRY IT

Creating a class for vending machines

Write a class, `VendingMachine`, as described below. Default values follow the attributes. Ex: `count`'s default value is `0`. Create a vending machine using a value read from input and call instance methods.

Instance attributes:

- count: 0
- max: 0

Methods:

- `__init__(num)`: initializes count and max with `num` parameter
- `refill()`: assigns `count` with `max`'s value and prints `"Refilled"`
- `sell(order)`: assigns `count` with the value of `count` minus `order` and prints `"Sold: [order]"`
- `print_stock()`: prints `"Current stock: [count]"`

Given input:

```
100
25
```

The output is:

```
Current stock: 100
Sold: 25
Current stock: 75
```

```
Refilled
Current stock: 100
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-3-instance-methods)

11.4 Overloading operators

Learning objectives

By the end of this section you should be able to

- Identify magic methods and describe their purpose.
- Develop overloaded arithmetic and comparison operators for user-defined classes.

Magic methods and customizing

Magic methods are special methods that perform actions for users, typically out of view of users. Magic methods are also called dunder methods, since the methods must start and end with double underscores (__). Ex: `__init__()` is a magic method used alongside __new__() to create a new instance and initialize attributes with a simple line like `eng = Engineer()`. A programmer can explicitly define a magic method in a user-defined class to customize the method's behavior.

CHECKPOINT

Customizing __str__() in a user-defined class, Engineer

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-4-overloading-operators)

CONCEPTS IN PRACTICE

Magic methods

1. Which of the following is a magic method?
   a. `add()`
   b. `_add_()`
   c. `__add__()`

2. Why are magic methods special?
   a. can't be called by the user
   b. perform internal actions
   c. have fixed definitions

3. Consider the example above, and identify the magic method(s) in the updated program.
   a. `__init__()`
   b. `__str__()`
   c. `__init__()`, `__str__()`

Overloading arithmetic operators

Operator overloading refers to customizing the function of a built-in operator. Arithmetic operators are commonly overloaded to allow for easy changes to instances of user-defined classes.

CHECKPOINT

Overloading __add__() for a user-defined class, Account

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-4-overloading-operators)

Arithmetic operator (Operation)	Magic method
+ (Addition)	`__add__(self, other)`
- (Subtraction)	`__sub__(self, other)`
* (Multiplication)	`__mul__(self, other)`
/ (Division)	`__truediv__(self, other)`
% (Modulo)	`__mod__(self, other)`
** (Power)	`__pow__(self, other)`

Table 11.1 Arithmetic operators and magic methods.

CONCEPTS IN PRACTICE

Arithmetic operator overloading

4. A programmer explicitly defining the modulo operator for their user-defined class is _____ the operator.

a. overloading
b. rewriting
c. implementing

5. Given the code below, which argument maps to the parameter `other` in the call to `Account.__add__()`?

```
acct_a = Account("Ashe", 6492)
acct_b = Account("Bevins", 5210)

acct_ab = acct_a + acct_b
```

a. `acct_a`
b. `acct_b`
c. `acct_ab`

6. Which `__sub__()` definition overloads the - operator for the code below to work?

```
class Pair:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
  # Define __sub__()

p1 = Pair(10, 2)
p2 = Pair(8, 4)
p3 = p1 - p2
print(p3.x, p3.y)
```

a.
```
def __sub__(self):
  return Pair(self.x - other.x, self.y - other.y)
```

b.
```
def __sub__(self, other):
  return self.x - other.x, self.y - other.y
```

c.
```
def __sub__(self, other):
  return Pair(self.x - other.x, self.y - other.y)
```

Overloading comparison operators

Comparison operators can also be overloaded like arithmetic operators.

Comparison operator (Operation)	Magic method
< (Less than)	`__lt__(self, other)`
> (Greater than)	`__gt__(self, other)`
<= (Less than or equal to)	`__le__(self, other)`
>= (Greater than or equal to)	`__ge__(self, other)`
== (Equal)	`__eq__(self, other)`
!= (Not equal)	`__ne__(self, other)`

Table 11.2 Comparison operators and magic methods.

EXAMPLE 11.2

Overloading comparison operators for the Account class

Code	Output

`class Account:`
`  def __init__(self, name="", amount=0):`
`  self.name = name`
`  self.amount = amount`
 	
`  def __str__(self):`
`  return f"{self.name}: ${self.amount}"`
 	
`  def __lt__(self, other):`
`  return self.amount < other.amount`
 	
`  def __gt__(self, other):`
`  return self.amount > other.amount`
 	
`  def __eq__(self, other):`
`  return self.amount == other.amount`
 	
`acct_a = Account("Ashe", 6492)`
`acct_b = Account("Bevins", 5210)`
 	
`print(acct_a < acct_b)`
`print(acct_a > acct_b)`
`acct_a.amount = 5210`
`print(acct_a == acct_b)`
	`False`
	`True`
	`True`

Table 11.3

CONCEPTS IN PRACTICE

Comparison operator overloading

Consider the example above.

7. How many operators are overloaded?

a. 3
b. 4
c. 5

8. Which code appropriately overloads the <= operator?

a.
```
def __le__(other):
  return self.amount <= other.amount
```
b.
```
def __le__(self, other):
  return self.amount <= other.amount
```
c.
```
def __le__(self, other):
  return other.amount <= self.amount
```

9. Which type of value does `__gt__()` return?
   a. account instance
   b. Boolean
   c. integer

TRY IT

Combining exercise logs

The `ExerciseLog` class has two instance attributes: `e_type`, the type of exercise, and `duration`, the time spent exercising.

Overload the + operator to combine two `ExerciseLogs` such that:

- If the exercise types are different, combine them with `" and "` in between. Else, use the same type and don't duplicate.
- Add durations together.

Given input:

```
walk
5
run
30
```

The output is:

```
walk and run: 35 minutes
```

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-4-overloading-operators)

TRY IT

Expanding the Account class

Using `isinstance()` allows the programmer to define different behaviors depending on an object's type. The first parameter is the object, and the second parameter is the type or class. Ex: `isinstance(my_var,`

`int`) returns `True` if `my_var` is an integer.

Expand the existing `Account` example so that the addition operator can also be used to add an integer value to an `Account`'s amount attribute.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-4-overloading-operators)

11.5 Using modules with classes

Learning objectives

By the end of this section you should be able to

- Construct an import statement that selectively imports classes.
- Create an alias to import a module.

Importing classes from modules

Import statements, as discussed in the Modules chapter, allow code from other files, including classes, to be imported into a program. Accessing an imported class depends on whether the whole module is imported or only selected parts.

Multiple classes can be grouped in a module. For good organization, classes should be grouped in a module only if the grouping enables module reuse, as a key benefit of modules is reusability.

CHECKPOINT

Importing classes from files

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-5-using-modules-with-classes)

CONCEPTS IN PRACTICE

Importing classes

1. Consider the example above. How many classes are imported in the second *main.py*?
   a. 1
   b. 2
   c. 3

2. Which line imports only the `Level` class from character?
   a. `import Level from character`
   b. `from character import Level`
   c. `import Level`

3. How many classes can be in a module?
   a. none
   b. only 1
   c. any number

Using aliases

Aliasing allows the programmer to use an alternative name for imported items. Ex: `import triangle as tri` allows the program to refer to the triangle module as tri. Aliasing is useful to avoid name collisions but should be used carefully to avoid confusion.

EXAMPLE 11.3

Using aliasing to import character

character.py

```
class Level:
  def __init__(self, level=1):
    self.level = level
    ...
  def level_up(self):
    ...
class XP:
  def __init__(self, XP=0):
    self.XP = XP
    ...
  def add_xp(self, num):
    ...
```

main.py

```
import character as c

bard_level = c.Level(1)
bard_XP = c.XP(0)
bard_XP.add_xp(300)
bard_level.level_up()
...
```

CONCEPTS IN PRACTICE

Using an alias

Consider the example above. Suppose a program imports the character module using `import character as game_char`.

4. How would the program create a `Level` instance?
   a. `rogue_level = character.Level()`
   b. `rogue_level = game_char.Level()`

```
    c. rogue_level = character.game_char.Level()
```

5. Which code creates a name collision?
    a. `character = 'Ranger'`
    b. `Game_char = 'Ranger'`
    c. `game_char = 'Ranger'`

TRY IT

Missing import statement

Add the missing import statement to the top of the file. Do not make any changes to the rest of the code. In the end, the program should run without errors.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-5-using-modules-with-classes)

TRY IT

Missing class import statement

Add the missing class import statement to the top of the file. Import only a class, not a full module. Do not make any changes to the rest of the code. In the end, the program should run without errors.

Access multimedia content (https://openstax.org/books/introduction-python-programming/pages/11-5-using-modules-with-classes)

11.6 Chapter summary

Highlights from this chapter include:

- Object-oriented programming involves grouping related fields and procedures into objects using data abstraction and encapsulation.
- Classes define a type of object with attributes and methods.
- Instances of a class type represent objects and are created using `__init__()`.
- Attributes may belong to the instance (unique for each instance) or the class (shared by all instances).
- Instance methods have the first parameter `self` to access the specific instance.
- Python uses magic methods to perform "under-the-hood" actions for users. Magic methods always start and end with double underscores.
- Python allows overloading of existing operators for user-defined classes.
- Classes can be imported from modules by name or can be renamed using aliases.

At this point, you should be able to write classes that have instance attributes, class attributes, and methods, and import classes from modules. You should also be able to overload operators when defining a class.

Construct	Description
Class definition	`class ClassName:` `  """Docstring"""`

`__init__()`	`class ClassName:`
	`  def __init__(self):`
	`  # Initialize attributes`

Attributes	`class ClassName:`
	`  class_attr_1 = [value]`
	`  class_attr_2 = [value]`
	
	`  def __init__(self):`
	`  instance_attr_1 = [value]`
	`  instance_attr_2 = [value]`
	`  instance_attr_3 = [value]`

Methods (instance)	`class ClassName:`
	
	`  def __init__(self):`
	`  instance_attr_1 = [value]`
	`  instance_attr_2 = [value]`
	`  instance_attr_3 = [value]`
	
	`  def method_1(self, val1, val2):`
	`  # Access/change attributes`
	
	`  def method_2(self):`
	`  # Access/change attributes`

Instances	`instance_1 = ClassName()`
	`instance_1.instance_attr_1 = [new value]`
	`instance_2 = ClassName()`
	`instance_2.method_2()`

Table 11.4 Chapter 11 reference.

Construct	Description

Overloaded multiplication operator	`class ClassName:`
	`  # Other methods`
	
	`  def __mul__(self, other):`
	`  return ClassName(self.instance_attr_3 * other.instance_attr_3)`

Import class from module with alias	`from class_module import ClassName as ClassAlias`

Table 11.4 Chapter 11 reference.