Module 7

Python Object-Oriented Programming

CENGAGE

Lesson 7.1: A First Look at OOP

- Object-oriented programming (OOP): Programming paradigm based on concept of objects.
  - Objects: Capsules of properties and procedures/methods.
- Allows us to abstract actual code and think more about attributes of data and operations around data.
- Advantages:
  - Code reusable.
  - Easier to design software as you can model it in terms of real-world objects.
  - Easier to test, debug, and maintain.
  - Data is secure due to abstraction and data hiding.

Lesson 7.2: OOP in Python

- Classes are OOP building-blocks.
  - Blueprints for objects.
- Everything in Python is an object.
- Running `type` function on any object will reveal class.

Lesson 7.2.1: Defining a Class in Python

- Syntax is minimal.
  - `class ClassName`
  - `type` function shows class `type`.
  - In Python, `class` and `type` are synonymous.

Defining a Class in Python

- Class: blueprint
- Attributes: noun, parts of the object
- Methods: action, things the object does
- Instantiation: copies of the class

Defining a Class in Python

Declaring the Employee Class

- class Employee:
  def _ _ init _ _(self):

Developing the Attributes (nouns)

- Building the blueprint using the keyword class
- Employee is the class name
- def is used to create the function
- Init method is used to initialize the method
- When you create a method as a class, it receive the instance as the first argument automatically. By convention you may call the instance Self.

Defining a Class in Python

Declaring the Employee Class

- class Employee:
  def _ _ init _ _(self, fname, lname, age):
    self.fname = fname
    self.lname = lname
    self.age = age

Developing the Attributes (nouns)

- After “self” you may specify other arguments to accept
- Set instance variables
  - Could use self.first = fname
    - It’s best to keep things the same

Defining a Class in Python

Declaring the Employee Class

- class Employee:
  def _ _ init _ _(self, fname, lname, age):
    self.fname = fname
    self.lname = lname
    self.age = age

emp_1 = Employee('Eric', 'Clayborn', 23)

emp_2 = Employee('Andrew', 'Churchill', 25)

Developing the Attributes (nouns)

- After "self" you may specify other arguments to accept
- Set instance variables
  - Could use self.first = fname
    - It's best to keep things the same
- Pass in values

Defining a Class in Python

Declaring the Employee Class

- class Employee:
  def _ _ init _ _(self, fname, lname, age):
    self.fname = fname
    self.lname = lname
    self.age = age

emp_1 = Employee('Eric', 'Clayborn', 23)

emp_2 = Employee('Andrew', 'Churchill', 25)

Developing the Attributes (nouns)

- After "self" you may specify other arguments to accept
- Set instance variables
  - Could use self.first = fname
    - It's best to keep things the same
- Pass in values
- What happens:
  - emp_1 will be passed to self
  - Eric will be passed to fname
  - Clayborn will be passed to lname
  - 23 will be passed to age

Defining a Class in Python

Declaring the Employee Class

- class Employee:
  def _ _ init _ _(self, fname, lname, age):
    self.fname = fname
    self.lname = lname
    self.age = age

emp_1 = Employee('Eric', 'Clayborn', 23)

emp_2 = Employee('Andrew', 'Churchill', 25)

Developing the Attributes (nouns)

- After "self" you may specify other arguments to accept
- Set instance variables
  - Could use self.first = fname
    - It's best to keep things the same
- Pass in values
- What happens:
  - emp_2 will be passed to self
  - Andrew will be passed to fname
  - Churchill will be passed to lname
  - 25 will be passed to age

Defining a Class in Python

Declaring the Employee Class

- class Employee:

```
    def _ _ init _ _(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def fullname (self):
        print(f"Hello {self.fname} {self.lname},
    you are {self.age} years old.")

emp_1 = Employee('Eric', 'Clayborn', 23)
emp_2 = Employee('Andrew', 'Churchill', 25)
```

Developing the Methods (actions)

- Each method within a class automatically takes the instance as the first argument.
  - Always call it "self"
- Create method containing self using a print statement to display full name and age

Defining a Class in Python

Declaring the Employee Class

- class Employee:
  def _ _ init _ _(self, fname, lname, age):
  self.fname = fname
  self.lname = lname
  self.age = age

  def fullname (self):
  print(f"Hello {self.fname} {self.lname}, you are {self.age} years old.")

emp_1 = Employee('Eric', 'Clayborn', 23)

emp_1.fullname()

emp_2 = Employee('Andrew', 'Churchill', 25)

emp_2.fullname()

Developing the Methods (actions)

- Each method within a class automatically takes the instance as the first argument.
  - Always call it "self"
- Create method containing self using a print statement to display full name and age
- Call the methods in order to display the actions or results
- Notice the ( ) located at the end, because it is a method being called rather than an attribute being passed
  - This displays the return value of the method

Defining a Class in Python

```
class Employee:
    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def fullname(self):
        print(f"Hello {self.fname} {self.lname}, you are {self.age} years old")

emp_1 = Employee("Eric", "Clayborn", 23)
emp_1.fullname()
emp_2 = Employee("Andrew", "Churchill", 25)
emp_2.fullname()
```

Defining a Class in Python

```
class Employee:
    def __init__(self, fname, lname, age): #objects of class Employee
        self.fname = fname
        self.lname = lname
        self.age = age

    def fullname(self): #methods of class Employee
        print(f"Hello {self.fname.capitalize()} {self.lname.capitalize()}, you are {self.age} years old")

    def rental(self): #methods of class Employee
        min_age = 25
        if self.age >= min_age:
            print(f"You are old enough to rent a car")
        else:
            older = min_age - self.age
            print(f"You may rent a car when you are {older} year(s) older")

emp_1 = Employee("eric", "clayborn", 23)
emp_1.fullname()
emp_1.rental()
print("")
emp_2 = Employee("andrew", "churchill", 25)
emp_2.fullname()
emp_2.rental()
```

```
Hello Eric Clayborn, you are 23 years old
You may rent a car when you are 2 year(s) older

Hello Andrew Churchill, you are 25 years old
You are old enough to rent a car
```

Lesson 7.2.2: Instantiating an Object

- Example: `jack = Person()`
- Each new object instantiated points to different objects.
  - If we declare a second `Person()` such as, `jill = Person()`
    - `jack is jill` will be `False`

Lesson 7.2.3: Adding Attributes to an Object

- Can add attributes dynamically.
  - Type name of object followed by dot (.) and then name of attribute.
  - Example: `jack.name = "Jack Smith"`
  - Try to avoid; this is bad practice.
- Every object has built-in attributes.
  - Such as `__dict__`
    - Dictionary that holds all attributes of object.

Lesson 7.2.4: The __init__ Method (1 of 2)

- Preferred way to add attributes is by defining them in object's constructor method.
- In Python constructor is `__init__`
  - `class ClassName`
- `hasattr()` function checks whether object has specific attribute or method.
  - In Python, `class` and `type` are synonymous.
- Define `__init__` just like function and specify attributes that need to be passed when instantiated object.
  - Snippet 7.13 shows example.
  - Note: `self` refers to object we're in process of creating.
  - If `__init__` defines parameters and you instantiate object without passing argument, you get `TypeError`.

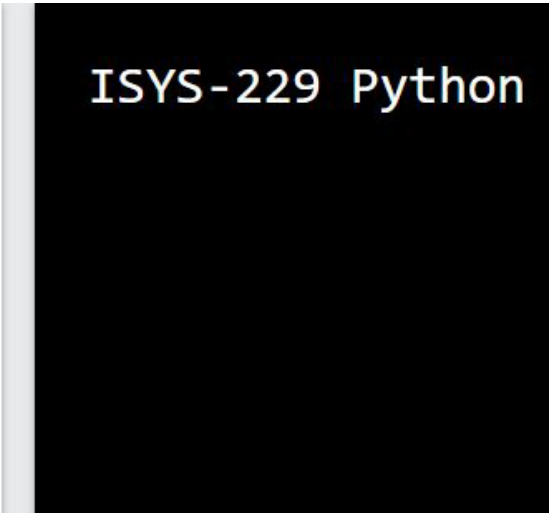
Lesson 7.2.4: The __init__ Method

- The __init__() function is always executed when the class is being initialized.
  - Use this function to assign values to objects properties

```
class Course:
  def __init__(self, course, section):
    self.course = course
    self.section = section

p1 = Course("ISYS-229", "Python")

print(p1.course, p1.section)
```



Lesson 7.2.4: The __init__ Method (2 of 2)

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
...
>>>
```

Snippet 7.13

Lesson 7.3.1: Defining Methods in a Class (1 of 2)

- Define method in class just like you would `__init__`.
- Can access instance attributes within instance methods.
- Snippet 7.22 shows example of creating method, accessing instance attributes within method, and calling method.

Lesson 7.3.1: Defining Methods in a Class (2 of 2)

```
>>> class Person:
...     def __init__(self, name, age, height_in_cm):
...         self.name = name
...         self.age = age
...         self.height_in_cm = height_in_cm
...     def speak(self):
...         print(f"Hello! My name is {self.name}. I am {self.age}
years old.")
...
>>> adam = Person("Adam", 47, 193)
>>> lovelace = Person("Lovelace", 24, 178)
>>> lucre = Person("Lucre", 13, 154)
>>> adam.speak()
Hello! My name is Adam. I am 47 years old.
>>> lovelace.speak()
Hello! My name is Lovelace. I am 24 years old.
>>> lucre.speak()
Hello! My name is Lucre. I am 13 years old.
>>>
```

Snippet 7.22

CENGAGE

Lesson 7.3.2: Passing Arguments to Instance Methods

- Can pass arguments to instance methods, just like normal functions.
- Can add condition statements to instance methods.

Lesson 7.3.3: Setting Instance Attributes within Instance Methods

- Can add instance attributes to instance methods.
  - Same process as adding attributes to `__init__`.

Lesson 7.4: Class versus Instance Attributes

- Instance attributes are bound to specific instance of the class.

Lesson 7.4.1: Class Attributes (1 of 2)

- Class attributes are bound the class and shared by all instances.
- Syntax is just like defining variable, only you do it in the class body.
  - Snippet 7.33 shows example.
- Access class attribute via class itself.
- Changing class attribute through class will reflect on all existing instances.
- Changing class attribute through instance will create instance attribute.
- Can access and change class attributes in instance methods.

Lesson 7.4.1: Class Attributes (2 of 2)

```
class WebBrowser:
    connected = True
    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False
```

Snippet 7.33

Lesson 7.5.1: Creating Instance Methods (1 of 2)

- Instance methods must receive `self` as first argument.
  - Don’t need to explicitly pass in value for `self`.
- Bound method: One that takes an instance (`self`) as first parameter.
  - Every instance of class has its own copy of method.
- Snippet 7.49 shows instance method examples.

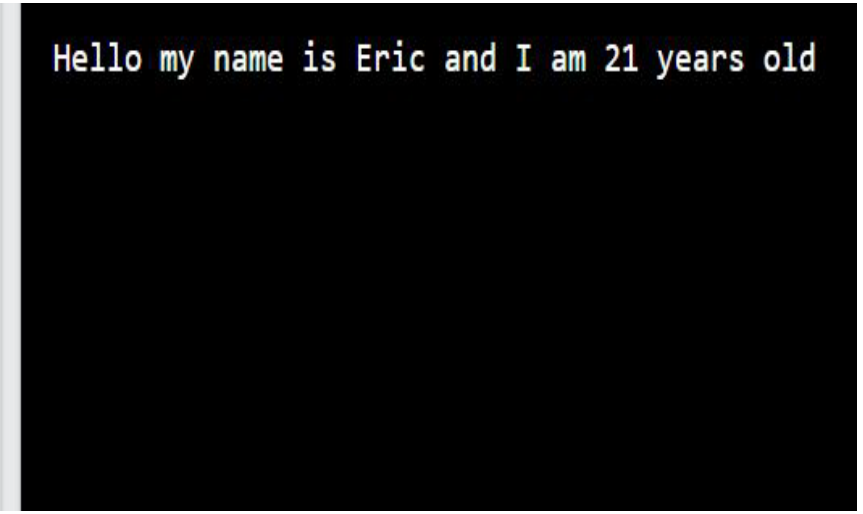
Lesson 7.5.1: Creating Instance Methods

- The Self parameter is a reference to the current instance of the class and is used to access variable that belongs to the class.

```
class Person:
  def __init__(self, name, age): #declaring attributes
    self.name = name
    self.age = age

  def myfunc(self): #declaring method
    print(f"Hello my name is {self.name} and I am {self.age} years old")

p1 = Person("Eric", 21)
p1.myfunc()
```



Lesson 7.5.1: Creating Instance Methods (2 of 2)

```
class WebBrowser:
    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False

    def navigate(self, new_page):
        self.current_page = new_page
        if not self.is_incognito:
            self.history.append(new_page)

    def clear_history(self):
        self.history[:-1] = []
```

Snippet 7.49

Lesson 7.5.2: Class Methods (1 of 3)

- Class methods are bound to class itself and not instance.
- Do not have access to instance attributes.
- Called through class; don't require creation on instance.
- First parameter is always class itself.
  - First argument is reserved.
  - Convention is to use name `cls`.

Lesson 7.5.2: Class Methods (2 of 3)

- Common use is for making factory methods.
  - Factory methods are ones that return objects.
  - Can be used to return objects of different types.
- See Snippet 7.52 for class method example.
  - Function definition starts with `@classmethod`.

Lesson 7.5.2: Class Methods (3 of 3)

```
class WebBrowser:
    def __init__(self, page):
        self.history = [page]
        self.current_page = page
        self.is_incognito = False

    def navigate(self, new_page):
        self.current_page = new_page
        if not self.is_incognito:
            self.history.append(new_page)

    def clear_history(self):
        self.history[:-1] = []

    @classmethod
    def with_incognito(cls, page):
        instance = cls(page)
        instance.is_incognito = True
        instance.history = []
        return instance
```

Snippet 7.52

Lesson 7.5.3: Encapsulation and Information Hiding

- Encapsulation: Bundling of data with methods that operate on that data.
  - Used to hide internal state of object.
- Information hiding: Hiding of internal state of object.
  - Used to abstract away irrelevant details about class to prevent users from changing them.
  - In Python, accomplished by marking attributes `private` or `protected`.
    - `private`: only used inside class definition and not accessed externally.
      - Prefix attribute name with double underscore.
      - Attribute inaccessible outside of class.
    - `protected`: similar to `private`, but only used in very specific contexts.
      - Prefix attribute name with underscore.
      - Interpreter doesn't enforce restrictions, this is just marker letting users know not to access outside of class.

Lesson 7.6: Class Inheritance

- Inheritance: Allows for class's implementation be derived from another class's implementation.
  - Subclass/derived/child class inherits all attributes and methods of superclass/base/parent class.
- Benefits:
  - Makes code more reusable.
  - Makes it easier to extend functionality.
  - Adds flexibility.
- In Python define class as usual but pass the base class as parameter.
  - Syntax: `class Subclass(Superclass):`

Lesson 7.6.1: Overriding __init__()

- Overriding: Redefining the implementation of a method defined in superclass to add or change subclass's functionality.
- Can override __init__ to add attribute to subclass.
  - Use `super` method to access inherited methods from parent class that have been overwritten in child.

Lesson 7.6.2: Commonly Overridden Methods

- Dunder or magic methods: Special methods prefixed and suffixed with double underscores.
  - Examples:
    - `__init__`
    - `__str__`
    - `__del__`
- Commonly overridden to customize class.

Lesson 7.6.3: The __str__() Method

- Every object has `__str__` method by default.
- Called every time `print` is called.
- Override to customize string containing readable representation of object.

Lesson 7.6.4: The __del__() Method

- `__del__` is destructor method.
- Called whenever object gets destroyed.
- Might want to override to print message to user.

Lesson 7.7: Multiple Inheritance

- Allows you to inherit attributes and methods from more than one class.
- Commonly used for mixins.
  - Mixins: Classes that have methods/attributes that are meant to be used by other functions.
    - Example: `Logger` class has `log` method that writes to logfile, and when added to your class as mixin gives it that capability.,