LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 7: OBJECT ORIENTED PROGRAMMING

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

7.

OBJECT-ORIENTED PROGRAMMING

BY FRED ANNEXSTEIN, PHD

CHAPTER 7 OVERVIEW

In this module we will be covering an introduction to the style of object oriented programming using Python. Object-oriented design and object-oriented programming is quite popular in Python since the language natively supports the concepts of "classes, objects, attributes, and methods". Object-oriented design is a methodology that supports code reuse and the long-term maintenance of computer code.

CHAPTER 7 OBJECTIVES

By the end of this chapter, you will be able to...

- Understand the need and objectives of OOP.
- Understand the syntax for classes, objects, and methods.
- Apply the syntax of the language to correctly execute OOP codes.
- Understand inheritance and polymorphism as popularly used in Python for OOP.
- Apply OOP methods to the implementation of linked lists and trees.

DEFINING OOP

Object-oriented programming (OOP) is a methodology for organizing programs into units

called classes and objects. OOP is an alternative design process for building data abstraction and abstract data types. OOP classes create abstraction layers or barriers between the use and implementation of data types. Like the dispatch dictionaries we discussed in Chapter 2, objects respond to behavioral requests. Like mutable data structures, objects have a local state that is not directly accessible from the global environment. The Python object system provides convenient syntax to promote the use of these techniques for organizing programs. Much of this syntax is shared among other object-oriented programming languages.

OBJECT-BASED PROGRAMMING

The vast majority of object-oriented programming and design involves creating and using objects of existing classes. You have been doing this already when using built-in types like int, float, str, list, tuple, dict and set. Over the years, the Python open-source community has crafted an enormous number of valuable classes and packaged them into class libraries. This makes it easy for you to reuse existing classes. The vast majority of the classes you will need in programming are likely to be freely available in open-source libraries.

However, every competent programmer must master the design and implementation of

custom classes. Unlike standard classes, these will be classes that you write yourselves.

The object system offers more than just convenience. It enables a new metaphor for designing programs in which independent agents interact within the computer. Each object bundles together local state and behavior in a way that abstracts the complexity of both.

Objects communicate with each other, and useful results are computed as a consequence of their interaction. Not only do objects pass messages, they also share behavior among other objects of the same type and inherit characteristics from related types. The paradigm of object-oriented programming has its own vocabulary that supports the object metaphor. We have seen that an object is a data value that has methods and attributes, accessible via dot notation. Every object also has a type, called its class. To create new types of data, we implement new classes.

CREATING YOUR OWN CUSTOM CLASSES

A class serves as a template for all objects whose type is that class. Every object is an instance of some particular class. The objects we have used so far all have built-in classes, but new user-defined classes can be created as well. A class definition specifies the attributes

and methods shared among objects of that class. Classes that you create are new data types. A class is a blueprint for a collection of objects. Classes can specify the use of data structures defined by the user. Users create new classes with the keyword class to define a block of code that keeps related things together.

Let's write a simple class starting with no attributes or methods. We instantiate an object or instance of the class Pokemon, and assign the object just created to the variable name poke.

```
class Pokemon:
    pass

poke = Pokemon()

>>> poke
<__main__.Pokemon at 0x7f9761346cd0>
```

Because our Python custom class is empty, it simply returns the address where the object is stored. In object-oriented programming, the properties of a custom object are defined by attributes, while its methods define its behavior.

In Python, the variable name *self* is used as a keyword to represent an instance of a class. It works as a handle to access the class members, such as attributes from the class methods. It is the first argument to a function called the constructor or the __init__() method

and is called automatically to initialize the class attributes with the values defined by the user. Let us look at an example:

```
class Pokemon:

    def __init__(self):
        print(“calling constructor...")

>>> poke = Pokemon()
```

INSTANCE AND CLASS ATTRIBUTES

Let's update the Pokemon class with an __init()__ constructor method that creates two attributes: name and attack. These attributes are called instance attributes.

```
class Pokemon:
    def __init__(self, name, attack):
        self.name = name
        self.attack = attack

>>> poke = Pokemon(‘charizard’,‘blast_burn’)

>>> poke.name, poke.attack
('charizard', 'blast_burn')
```

Instance methods are functions defined inside a class and can only be called from an instance of that class. Like __init__(), the first parameter of an instance method is always self.

Let's define some instance methods for our Python custom class Pokemon.

```
class Pokemon:
    def __init__(self, name, attack):
        self.name = name
        self.attack = attack

    def description(self):
        return f"{self.name} favorite attack
                          is {self.attack}"

    def speak(self, sound):
        return f"{self.name} says {sound}”

>>> pchu = Pokemon("Pikachu", “ElectroBall")

>>> pchu.description()
"Pikachu favorite attack is ElectroBall"

>>> pchu.speak("pikachu pikachu")
'Pikachu says pikachu pikachu'
```

CLASS ATTRIBUTES

Class attributes are the variables defined directly in the class that are shared by all objects of the class. Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are usually defined in the constructor.Class Attributes are defined directly inside a class, and shared across all objects of the class. Class attributes are accessed using class name as well as using

object with dot notation, for example using syntax *classname.class_attribute or object.class_attribute*. Changing the value of a class attribute by using *classname.class_attribute = value* will be reflected to all the objects. Changing value of instance attribute will not be reflected to any other objects.

Class attributes are created by assignment statements in the suite of a `class` statement, outside of any method definition. In the broader developer community, class attributes may also be called class variables or static variables. The following class statement creates a collection of three class attributes for the `Pokemon` class.

```
class Pokemon():
    attack = 12
    defense = 10
    health = 15

    def __init__(self, name, level = 5):
        self.name = name
        self.level = level
```

Be aware of the difference between class attributes and instance attributes. In our example above, attack, defense, and health are class attributes, where name and level are instance attributes.

INHERITANCE

We extend the Pokemon class to include attributes for leveling and evolving characteristics. We use inheritance to define a new class Grass_Pokemon as a subclass that inherits attributes and methods from Pokemon, but changes some aspects. In our example, the boost values are different. For the subclass Grass_Pokemon, we show how to add an additional method, called action, which returns the string "[name of pokemon] knows a lot of different moves!".

.

```
class Pokemon():
    attack = 12
    defense = 10
    health = 15

    def __init__(self, name, level = 5):
        self.name = name
        self.level = level

    def train(self):
        self.update()
        self.attack_up()
        self.defense_up()
        self.health_up()
        self.level = self.level + 1
        if self.level%self.evolve == 0:
            return self.level, "Evolved!"
        else:
            return self.level

    def attack_up(self):
```

```
        self.attack = self.attack +
                          self.attack_boost
        return self.attack

    def defense_up(self):
        self.defense = self.defense +
                          self.defense_boost
        return self.defense

    def health_up(self):
        self.health = self.health +
                          self.health_boost
        return self.health

    def update(self):
        self.health_boost = 5
        self.attack_boost = 3
        self.defense_boost = 2
        self.evolve = 10

    def __str__(self):
        self.update()
        return "Pokemon name: {}, Level:
          {}".format(self.name, self.level)

class Grass_Pokemon(Pokemon):
    attack = 15
    defense = 14
    health = 12

    def update(self):
        self.health_boost = 6
        self.attack_boost = 2
        self.defense_boost = 3
        self.evolve = 12

    def action(self):
        return "{} knows a lot of different
          moves!".format(self.name)

>>> p0 = Pokemon("Abe")
>>> p1 = Grass_Pokemon("Bella")
```

```
>>> print(p0)
Pokemon name: Abe Level: 5

>>> print(p1)
Pokemon name: Bella Level: 5

>> p1.train()
6

>>> p0.train()
6

>>> p1.attack
17

>>> p0.attack
15

>> p2 = Pokemon("Carlos",9)

>>> p2.train()
(10, 'Evolved!')
```

METHODS FOR COMPARING OBJECTS

We now write a function method for the Pokemon class that compares two health values, and returns whether the first object's health is strictly greater than the second, as follows:

```
>>> healthier(p0,p1)
True
```

This method is slightly more complicated because it operates on two `Pokemon` objects, not

just one. There are two approaches. We write an auxiliary function outside the class, which calls a class attribute as a method bound to the first argument.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class, as you could equivalently, assign a function object to a class variable.

```
class Pokemon():

…
    # function is class attribute
    def healthier (self, poke2):
         if self.health > poke2.health
         return True
         else: return False

# auxiliary function, not a class attribute

def healthier(poke1,poke2):
    return poke1.healthier(poke2)


>>> p0.healthier(p1)
True

>>> p1.healthier(p0)
False

>>> healthier(p0,p1)
True
```

AN OOP IMPLEMENTATION OF LINKED LIST ADT

In Chapter 6 we presented a functional definition of a linked list data structure. The definition we provided was functional in nature and recursive applying a pair abstraction consisting of a first data element and rest element that was itself a linked list. Here we present an OOP design and implementation of an analogous linked list data structure.

You will see in the implementation a new type of method called a repr method. The __repr__ method defines behavior which returns the string representation of an object. Typically, the __repr__ method is defined to return a string that can be executed by the interpreter and yield the same value as the given object. In other words, if you pass the returned string from the *object_name.__repr__()* method to the *eval()* function, then you will get back the same value as the value of the original object. Let us take a look at an example.

```
class Link:
    def __init__(self, first, rest=()):
        self.first = first
        self.rest = rest

    def first(self):
        return self.first

    def rest(self):
        return self.rest()
```

```
    def __repr__(self):
        if self.rest:
            return 'Link({0},
               {1})'.format(self.first,
                            repr(self.rest))
        else:
            return
     'Link({0})'.format(repr(self.first))
```

Let us write a function that returns a linked list of size n, where n is provided as an argument. We test our repr method to show that it produces a string that can be evaluated using eval to create an object identical to the original linked list.

```
def mylist(n):
    if n == 1:
        return Link(1)
    else:
        return Link(n, mylist(n-1) )

>>> five = list(5)

>>> five
Link(5,Link(4,Link(3,Link(2,Link(1)))))

>>> repr(five)
'Link(5,Link(4,Link(3,Link(2,Link(1)))))'

>>> y  = eval(repr(five))
>>> y
Link(5,Link(4,Link(3,Link(2,Link(1)))))
```