Susanne Anya Pryor

Instructors: Randall Root and Grace Nam

Python Module 8

Introduction

In this module we explore using classes in more detail, seeing how we can create within classes properties and attributes and how to control access to them. We explore the peculiar use of the keyword self to create objects within a class. We also look at GitHub desktop.

Constructors

A constructor acts inside a class to set the initial values for fields. It is called using double underscores and the command **init** for initialize. We also use the keyword **self** in the constructor. The first parameter following the initialize command is self.

```
# ------------------------------------------------ #
# Title: Listing02
# Description: A class with a constructor
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ------------------------------------------------ #

class Person:
    # --Fields--
    strFirstName = ""

    # -- Constructor --
    def __init__(self, first_name=''):
        #Attributes
        self.strFirstName = first_name

# --- Use the class ----
objP1 = Person()  # with no argument
objP2 = Person(first_name="Sue")  # with the parameter and argument

print(objP1.strFirstName)
objP1.strFirstName = "Bob"
print(objP1.strFirstName)
print("-------------")
print(objP2.strFirstName)
```

We could also have set the value of first name in the constructor by writing def__init__(self,first_name='Bob'):

There is a parallel function called a destructor that is called by double underscores and the command del. It is not used as frequently as memory is better managed now.

Self

The objects created in a class have specific locations in the computer memory, while they have the same field name they are separate memory locations. Without the keyword self the program will not be able to call the field value. It does appear as a parameter but it does not act like one, which can be confusing.

```
# ------------------------------------------------ #
# Title: Lab 8-2
# Description: A class with fields and a constructor
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ------------------------------------------------ #
```

```
# --- Make the class ---
class Person():
    # --Fields--
    strFirstName = ""
    strLastName = ""
```

```
    # -- Constructor --
    def __init__(self, first_name='', last_name=''):
        #Attributes
        self.strFirstName = first_name
        self.strLastName = last_name
    # -- Properties --
    # -- Methods --
```

```
# --End of class--

# --- Use the class ----
objP1 = Person()
objP1.strFirstName = "Bob"
objP1.strLastName = "Smith"
print(objP1.strFirstName, objP1.strLastName)
```

Attributes

As Python often performs functions in the background, attributes are invisible fields created by the constructor command. You can call them instead of fields:

```
# --- Make the class ---
class Person():
    # --Fields--
   # strFirstName = "" #commented out to try using
attributes instead of fileds
    #strLastName = "" #commented out to try using
attributes instead of fileds

    # -- Constructor --
    def __init__(self, first_name='', last_name=''):
        #Attributes
        self.FirstName = first_name
        self.LastName = last_name
    # -- Properties --
    # -- Methods --
# --End of class--
# --- Use the class ----
objP1 = Person("Bob", "Smith")
#objP1.strFirstName = "Bob" #commented out to try using
attributes instead of fileds
#objP1.strLastName = "Smith" #commented out to try using
attributes instead of fileds
print(objP1.FirstName, objP1.LastName)
```

Properties

The getter and setter commands allow you to control the properties of the objects you are creating in a class. Typically you would call them to validate the data type that is being input to ensure that it will act as expected. Generally the getter and setter are used together when you are both reading and writing the input. The syntax differs in the getter and setter commands. The setter uses the syntax **@methodname.setter**, while the getter uses the command **@property** and the **return** command.

The name of the attribute must match the property. By putting double underscores in front of the attribute name it will stay within the class construction. Be sure to create both a getter and setter for every attribute.

```
# -- Constructor --
def __init__(self, first_name):
    # -- Attributes --
    self.__first_name = first_name
# -- Properties --
# FirstName
@property  # DON'T USE NAME for this directive!
def first_name(self):  # (getter or accessor)
    return str(self.__first_name).title()  # Title case
```

```
@first_name.setter  # The NAME MUST MATCH the property's!
def first_name(self, value):  # (setter or mutator)
    if str(value).isnumeric() == False:
        self.__first_name = value
    else:
        raise Exception("Names cannot be numbers")
```

Methods

Methods refer to functions within a class, some of which are used also outside of a class as a stand alone function and others that are more specialized. You can restrict them to referencing the object from a class by using the **self** keyword.

The string method is called invisibly, but you can call it explicitly by using the double underscore with the keyword **str**.

```
# ------------------------------------------------- #
# Title: Listing06
# Description: A overriding the __str__() method
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ------------------------------------------------- #
```

```
class Demo1:
    var1 = "Some Data"

class Demo2:
    var1 = "Some Data"
    def __str__(self):
        return self.var1
d1 = Demo1()  # This object uses the default __str__() method
print(d1)
d2 = Demo2()  # This object uses the overridden __str__() method
s = str(d2)  # __str__() method run when the str() function is called
print(s)
print(d2)  # __str__() method run when the print() function is called
print(d2.__str__())  # __str__() method run when function is called directly
```

Static Method

You can use these within a class by using the keyword self but you don't have to. In general you would use a method that is static outside of a class, so you don't need the self parameter. Instance methods use the self parameter.

Private

You can mix instance methods and static methods. Instance methods are private, or contained within a class. They are indicated by the double underscore around the name of the method. Here is an example from the listing

```
@staticmethod  # You do not use the self keyword
def get_object_count():  # This is a PUBLIC static method
    return Person.__counter
```

```
@staticmethod  # You do not use the self keyword
def __set_object_count():  # This is a PRIVATE and static
method
    Person.__counter += 1
```

Type Hints

These are usually put in functions and methods to indicate the type of data expected.

Doc Strings

You can add in a documentation string on a class to help other developers understand what your code is doing. It is called by enclosing the text in triple quotes and then using the double underscores before and after the keyword **doc**. These can be accessed from the class or the object instance.

Assignment

This assignment, like assignment 6, remains a bit of a mystery to me. I have downloaded the answer file and rewatched the review video from assignment 6 but the code still will not run for me (probably due to the indent errors). Fundamentally, I think there is something wrong in how the data input is being put into the file. Here is the code that I have, it doesn't error but it also doesn't do what it is supposed to do.

```
#
------------------------------------------------------------
------------- #
```

```
# Title: Assignment 08
# Description: Working with classes

# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# RRoot,1.1.2030,Added pseudo-code to start assignment 8
# Anya Pryor, 12/7/21 ,Modified code to complete assignment 
8
# 
-----------------------------------------------------------
------------- #

# Data 
-----------------------------------------------------------
--------- #
strFileName = 'products.txt'
lstOfProductObjects = []

class Product:
    """Stores data about a product:

    properties:
        product_name: (string) with the products's  name
        product_price: (float) with the products's standard 
price
```

```
    methods:
    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
        Anya Pryor,12/6/21,Modified code to complete 
assignment 8
    """
    #pass
    # TODO: Add Code to the Product class
    def __init__(self, product_name, product_price):
        self.__PName = product_name
        self.__PPrice= product_price
    @property
    def product_name(self): #getter
        return str(__PName).title() #added title case
    @product_name.setter
    def product_name(self, value):
        if str(value).isNumeric == False:
            self.__PName=value
        else:
            raise Exception("Names cannot be numbers")
    @property
    def product_price(self):
        return float(__PPrice)
    @product_price.setter
    def product_price(self,value):
        if float(value).isNumeric == True:
            self.__PPrice=value
        else:
            raise Exception("Prices must be numbers")
    def listmethod(self):
        return [product_name, product_price]
# Data
------------------------------------------------------------
---------- #

# Processing
------------------------------------------------------------
-- #
class FileProcessor:
    """Processes data to and from a file and a list of
product objects:

    methods:
        save_data_to_file(file_name,
list_of_product_objects):

        read_data_from_file(file_name): -> (a list of
product objects)
```

```
    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
       Anya Pryor, 12/6/21,Modified code to complete
assignment 8
    """
    #pass
    # TODO: Add Code to process data from a file
    def read_data_from_file(file_name, list_of_rows):
        file = open(file_name, "r")
        for row in list_of_rows:
            file.read(row)
        file.close()
        return list_of_rows
    # TODO: Add Code to process data to a file
    def write_data_to_file(file_name, list_of_rows):
        file = open(file_name, "w")
        for row in list_of_rows:
            file.write(row["Product Name"] + "," +
row["Price"] + "/n")
        file.close()
        return list_of_rows
# Processing
------------------------------------------------------------
-- #
    @staticmethod
    def add_data_to_list(product, price, list_of_rows):
        """ Adds data into a list of dictionary rows
        :param task (string) task to add:
        :param priority (string) priority to add:
        :param list_of_rows: (list) you want filled with
file data:
        :return: (list) of dictionary rows
        """
        row = {"Product":
str(Product.product_name).strip(), "Price":
str(Product.product_price)}
        list_of_rows.append(row)
        return list_of_rows
# Presentation (Input/Output)
-------------------------------------------- #
class IO:
    # TODO: Add docstring
    #pass
# Presentation (Input/Output)
-------------------------------------------- #
    """ Performs Input and Output tasks """

    @staticmethod
    def output_menu_tasks(): # TODO: Add code to show menu
to user
```

```
        """  Display a menu of choices to the user

        :return: nothing
        """
        print('''
        Menu of Options
        1) See the current file
        2) Add a new Product and Price
        3) Save Data to File
        4) Exit Program
        ''')
        print()  # Add an extra line for looks

    @staticmethod
    def input_menu_choice():      # TODO: Add code to get 
user's choice
        """ Gets the menu choice from a user

        :return: string
        """
        choice = str(input("Which option would you like to 
perform? [1 to 4] - ")).strip()
        print()  # Add an extra line for looks
        return choice
    @staticmethod
    def output_current_prodcuts_in_list(list_of_rows):
# TODO: Add code to get user's choice
        """ Shows the current Tasks in the list of
dictionaries rows

        :param list_of_rows: (list) of rows you want to
display
        :return: nothing
        """
        print("******* The current products are: *******")
        for row in list_of_rows:
            print(row["Product"] + " (" + row["Price"] +
")")

print("*******************************************")
        print()  # Add an extra line for looks
    @staticmethod
    def input_new_product_and_price():     # TODO: Add code
to get product data from user
        """  Gather input from the user

        :parameter: task, priority
        :return: string, float
        """
        product = input("Enter a product: ")

        price = input("Enter a price ")

        return [product, price]
```

```
# Main Body of Script
----------------------------------------------------- #

# Load data from file into a list of product objects when
script starts

FileProcessor.read_data_from_file(file_name=strFileName,
list_of_rows=lstOfProductObjects)

# Show user a menu of options

while (True):

# Get user's menu option choice

    IO.output_menu_tasks()

    choice=IO.input_menu_choice()

    # Show user current data in the list of product objects

    if choice =='1':
```

```
IO.output_current_prodcuts_in_list(list_of_rows=lstOfProduc
tObjects)

    # Let user add data to the list of product objects

    elif choice =='2':

        product, price=IO.input_new_product_and_price()
        list_of_rows =
FileProcessor.add_data_to_list(product, price,
lstOfProductObjects)

    # let user save current data to file and exit program

    elif choice =='3':

        list_of_rows =
FileProcessor.write_data_to_file(file_name=strFileName,
list_of_rows=lstOfProductObjects)

        print("All Saved!")

    elif choice =='4':

        print("goodbye")

        break
```

Summary

Classes can contain objects that are have sophisticated logic, including the ability to control access to the objects and to control their format and function. The development of classes allows the programmer to create reusable blocks of code, where new arguments can be easily passed in so the same code can fulfill different needs.