PROGRAMMING WITH PYTHON

By Randal Root

Module 09

In this module, you learn about **creating scripts using modules**. Modules are **used to organize function and classes**. Next, you learn how the **inherit code from one class** into another, then document the **relationships between classes using UML**. Finally, you learn how to use the **command console to work with GitHub**.

Modules

Script modules are a way to organized your code. Like functions and classes, code **modules make your code easier to read and re-use**! Module contain classes and the functions. Each **module can have many classes**, just as each class can have many methods. Listing 1 shows an example of a script module.

```
# ------------------------------------------------------- #
# Title: Listing01Module
# Description: Components of a typical module
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------- #

def standalone_function():
    print("Called standalone_function")

class MyData:
    def __init__(self):
        print("Created MyData object")

class FileProcessor:

    @staticmethod
    def process_data():
        print("Called process_data")

class IO:

    @staticmethod
    def print_data():
        print("Called print_data")
```

Listing 1

The Import Command

To create a code module in Python, you make a Python script as you normally would, but you do not directly run the code in that script. Instead, you **use code in the module indirectly from another script file**. Creating a script whose purpose is to test the module is known as using a "**Test Harness**." Listing 2 shows a simple example of **connection** a Test Harness script to **the "import" command**.

```
# ------------------------------------------------------- #
# Title: Listing02
# Description: A script that uses a module
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------- #
import Listing01Module # Note that you do not use the file's extension.

# Calling a standalone function
Listing01Module.standalone_function()
```

```
# Creating an object from a class
objMD = Listing01Module.MyData()

# Calling static methods in two classes
Listing01Module.FileProcessor.process_data()
Listing01Module.IO.print_data()
```

Listing 2

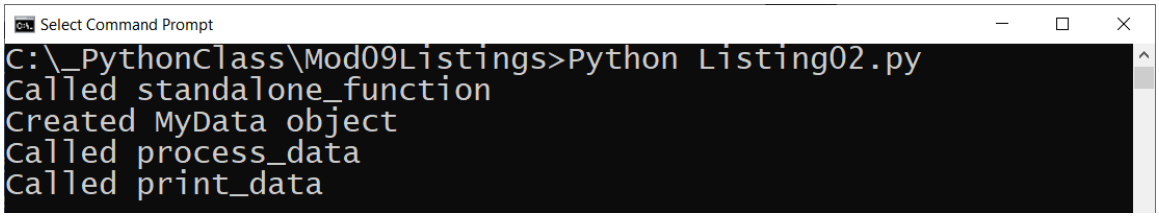

Figure 1. The results of Listing 2

When Python finds an import statement in a script file, it automatically searches and connects to the model's script file, if it can find it. First, it looks for the file within the same folder, and then it looks in any folder specified by the environment variable "PYTHONPATH," if one exists. If not, you can create one, and it uses the folders defined within the variable.

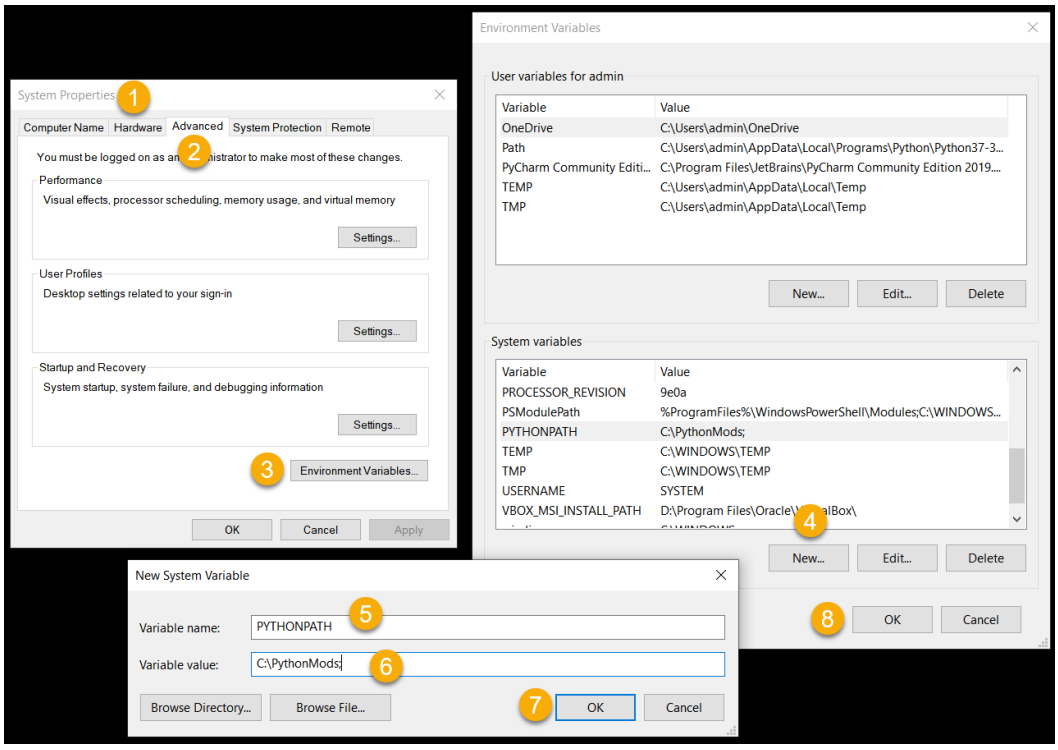

Figure 2. Creating *a "PYTHONPATH environment variable on Windows*

Note: *You do <u>not</u> need to create a "PYTHONPATH environment variable for this class, but it is nice to have an example of how a computer uses these variable. You can see all the system variable on your computer using the "set" command on Windows and the "printenv" command on Mac.*

The Main Module

Python applications (programs) are often created using **a set of two or more files**. However, you typically on run only one of the files directly. **Any script file you run directly**, at the start of your program, **is called the "Main" module**.

The __name__ System Variable

The system variable "__name__" returns the string "__main__" when you execute a script directly using the Python executable (Listing 3). Developers use the Python system variable "__name__" to verify that a script is running as the Main module.

```
# Description: A script that runs as the "Main" module
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------ #
print(__name__)
if __name__ == "__main__":
    print("This file is the starting point of my program!")
```

Listing 3

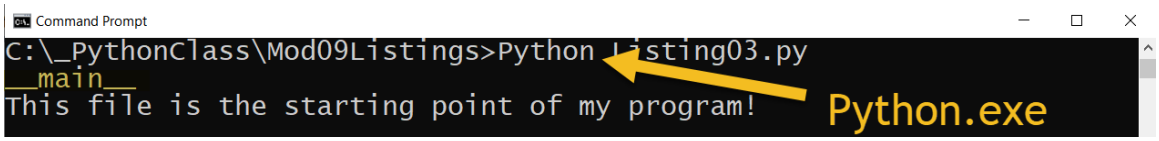

Figure 3. The results of Listing 3

You reference other modules you want to include in your program using an “import” statement at the beginning of the Main module, using a comma-separated list of each module. The code in Listing 4 checks that its script file is running as the main module before it tries to import other modules.

```
# ------------------------------------------------------ #
# Title: Listing 04
# Description: A script module that should be run as "Main"
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------ #
if __name__ == "__main__":
    import DataProcessor, Employees # Will error if the modules are not found
else:
    raise Exception("This file was not created to be imported")
```

Listing 4

When a script is not ran directly as the main module the "__name__" system variable returns the string with the name of the file. The code in Listing 5 raises an error message when someone tries to run its script file directly. Use this code in all modules that are not your applications main module.

```
# ------------------------------------------------------ #
# Title: Listing05
# Description: A script module that should not run as "Main"
# ChangeLog (Who,When,What):
```

```
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------------ #
if __name__ == "__main__":
    raise Exception("This file is not meant to ran by itself")
```

Listing 5

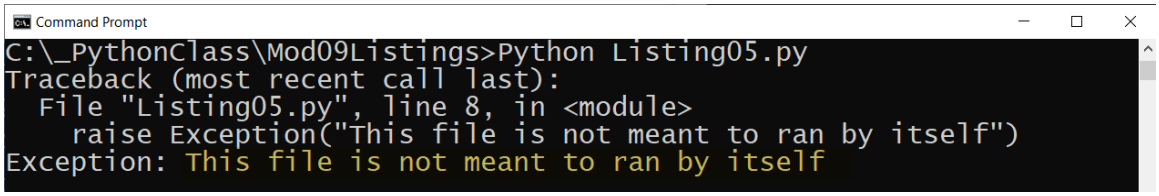

Figure 4. The result of listing 4

Python automatically creates a subfolder called *__pycache__* when you first import a module. In there, you will find a binary version of your Python code that is recreated if you delete it.

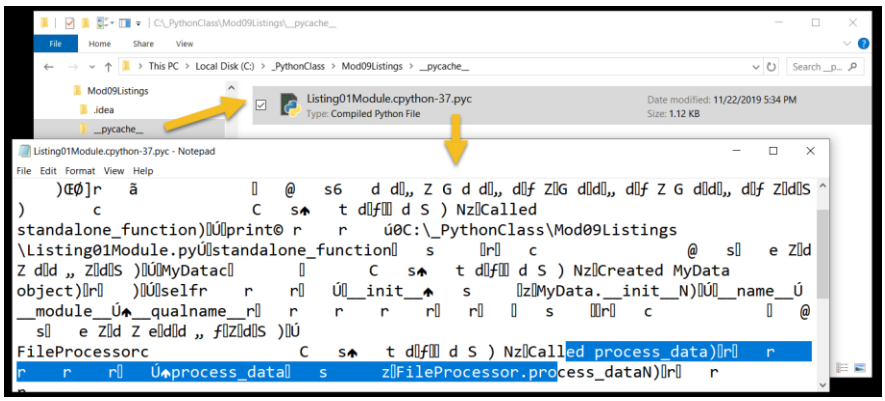

Figure 5. The compiled python version of a script module

Note: *You do <u>not</u> need to turn the __pycache__ folder in as part of your assignment, but it is also OK if you do.*

Linking Modules Together

Modules are often designed to hold a set of classes or function to support **data, processing, or presentation** tasks (separation of concerns). Listing 6 shows a module designed for data.

```
# ------------------------------------------------------------ #
# Title: Listing 06
# Description: A module of data classes
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------------ #
if __name__ == "__main__":
    raise Exception("This file is not meant to ran by itself")

class Person():
    """ Stores data about a person:
```

```
    properties:
        first_name: (string) with the persons's first name

        last_name: (string) with the persons's last name
    methods:
        to_string() returns comma separated product data (alias for __str__())
    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
    """
    # -- Constructor --
    def __init__(self, first_name, last_name):
        # -- Attributes --
        self.__first_name = first_name
        self.__last_name = last_name

    # -- Properties --
    @property
    def first_name(self):
        return str(self.__first_name).title()

    @first_name.setter
    def first_name(self, value):
        if not str(value).isnumeric():
            self.__first_name = value
        else:
            raise Exception("Names cannot be numbers")

    @property
    def last_name(self):
        return str(self.__last_name).title()

    @last_name.setter
    def last_name(self, value):
        if not str(value).isnumeric():
            self.__last_name = value
        else:
            raise Exception("Names cannot be numbers")

    # -- Methods --
    def to_string(self):
        """ Explicitly returns a string with this object's data """
        return self.__str__()

    def __str__(self):
        """ Implicitly returns a string with this object's data """
        return self.first_name + ',' + self.last_name
```

Listing 6

You can create a **module with multiple classes**. Listing 7 shows a module for processing.

```
# ------------------------------------------------------------ #
# Title: Listing 07
# Description: A module of multiple processing classes
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------------ #
if __name__ == "__main__":
    raise Exception("This file is not meant to ran by itself")

class FileProcessor:
    """Processes data to and from a file and a list of objects:

    methods:
        save_data_to_file(file_name,list_of_objects):

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

    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
    """

    @staticmethod
    def save_data_to_file(file_name: str, list_of_objects: list):
        """ Write data to a file from a list of object rows

        :param file_name: (string) with name of file
        :param list_of_objects: (list) of objects data saved to file
        :return: (bool) with status of success status
        """
        success_status = False
        try:
            file = open(file_name, "w")
            for row in list_of_objects:
                file.write(row.__str__() + "\n")
            file.close()
            success_status = True
        except Exception as e:
            print("There was a general error!")
            print(e, e.__doc__, type(e), sep='\n')
        return success_status

    @staticmethod
    def read_data_from_file(file_name: str):
        """ Reads data from a file into a list of object rows

        :param file_name: (string) with name of file
        :return: (list) of object rows
        """
        list_of_rows = []
        try:
            file = open(file_name, "r")
            for line in file:
```

```
                row = line.split(",")
                list_of_rows.append(row)
            file.close()
        except Exception as e:
            print("There was a general error!")
            print(e, e.__doc__, type(e), sep='\n')
        return list_of_rows

class DatabaseProcessor:
    pass
    # TODO: Add code to process to and from a database
```

Listing 7

You should **test modules as you create them**. Listing 8 shows a Test Harness that import a data and processing module. The code **uses an alias for each module** to make the modules more convenient to reference.

```
# ------------------------------------------------------ #
# Title: Listing 08
# Description: A main module for testing
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------ #
if __name__ == "__main__":
    import Listing06 as D  # data classes
    import Listing07 as P  # processing classes
else:
    raise Exception("This file was not created to be imported")

# Test data module
objP1 = D.Person("Bob", "Smith")
objP2 = D.Person("Sue", "Jones")
lstTable = [objP1, objP2]
for row in lstTable:
    print(row.to_string(), type(row))

# Test processing module
P.FileProcessor.save_data_to_file("PersonData.txt", lstTable)
lstFileData = P.FileProcessor.read_data_from_file("PersonData.txt")
for row in lstFileData:
    p = D.Person(row[0], row[1])
    print(p.to_string().strip(), type(p))

# Test IO classes
# TODO: create and test IO module
```

Listing 8

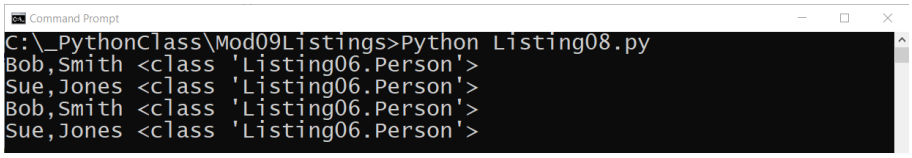

Figure 6. The result of Listing 8

Lab 9-1

In this lab, you create a script module with a "Person" class and another with a "FileProcessor" class. You then import the modules into a test harness and verify that they work.

Note: I have provided the listing code in a file called "Mod08Listings.zip" if you do not want to type the code yourself.

1. Create a new script module called "DataClasses.py".

2. Add the code in Listing 6 to the DataClasses.py script to create the "Person" class.

3. Create a new script module called "ProcessingClasses.py".

4. Add the code in Listing 7 to the ProcessingClasses.py script to create the "FileProcessor" and "DatabaseProcessor" classes.

5. Create a new script module called "TestHarness.py".

6. Add the code in Listing 8 to the TestHarness.py script to test the "FileProcessor" and "Person" classes.

7. Verify that the results look like those shown in Figure 6.

Inheritance

One of the main advantages of using classes to organize your code is the ability to "inherit" code from another class. Inheriting code is a convenient way for you to **write code once and uses it multiple times**.

The relationship between the class that inherits code from another is often referred to as a "**Parent class – Child class" relationship**. This can be documented using an arrow point from the child to its parent, like this:

```
Child class -> Parent class
```

Oddly, though, you might expect the arrow to indicate otherwise, code is inherited from the Parent class to the Child class.

People also use other terms for the inheritance relationship. Two popular ones you should know is a "Derived class – Base class" relationship or and "Sub class – Super class" relationship! Like before, you use an arrow to point to the class whose code is inherited.

```
Derived class -> Base class
Sub class -> Super class
```

You can have a long string of inherited classes and the child of one class can be parent of another.

```
Child class -> Child class -> Child class -> Parent class
```

Python has a class called **"object" that is the ultimate parent class of all classes.** Listing 9 shows an example of a Person class that is the parent class for an Employee class. Note how the Person class inherits code from Python's "object" class. **Inheritance of the object class is implied** if you leave the parenthesis empty as we did in Listing 6.

```
# ------------------------------------------------------- #
# Title: Listing 09
# Description: A module of data classes
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------- #
if __name__ == "__main__":
    raise Exception("This file is not meant to ran by itself")

class Person(object):  # Inherits from object
    """Stores data about a person:

    properties:
        first_name: (string) with the persons's first name

        last_name: (string) with the persons's last name


    methods:
        to_string() returns comma separated product data (alias for __str__())
    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
    """

    # -- Constructor --
    def __init__(self, first_name, last_name):
        # -- Attributes --
        self.__first_name = first_name
        self.__last_name = last_name

    # -- Properties --
    @property
    def first_name(self):
        return str(self.__first_name).title()

    @first_name.setter
    def first_name(self, value):
        if not str(value).isnumeric():
            self.__first_name = value
        else:
            raise Exception("Names cannot be numbers")

    @property
    def last_name(self):
        return str(self.__last_name).title()
```

```
    @last_name.setter
    def last_name(self, value):
        if not str(value).isnumeric():
            self.__last_name = value
        else:
            raise Exception("Names cannot be numbers")

    # -- Methods --
    def to_string(self):
        """ Explicitly returns a string with this object's data """
        return self.__str__()

    def __str__(self):
        """ Implicitly returns a string with this object's data """
        return self.first_name + ',' + self.last_name


class Employee(Person):  # Inherits from Person
    """Stores data about an Employee:

    properties:
        employee_id: (int) with the employees's ID

        first_name: (string) with the employees's first name

        last_name: (string) with the employees's last name


    methods:
        to_string() returns comma separated product data (alias for __str__())
    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
    """

    def __init__(self, employee_id=""):
        # Attributes
        self.__employee_id = employee_id

    # --Properties--
    @property
    def employee_id(self):
        return self.__employee_id

    @employee_id.setter
    def employee_id(self, value):
        if str(value).isnumeric():
            self.__last_name = value
        else:
            raise Exception("IDs must be numbers")
```

```
    # --Methods--
    def to_string(self):  # Overrides the original method (polymorphic)
        """ Explicitly returns a string with this object's data """
        # Oddly, linking to self.__str__() doesn't work as expected
        data = super().__str__()  # get data from parent(super) class
        return str(self.employee_id) + ',' + data

    def __str__(self):  # Overrides the original method (polymorphic)
        """ Implicitly returns field data """
        data = super().__str__()  # get data from parent(super) class
        return str(self.employee_id) + ',' + data
```

Listing 9

The Employee class **inherits the fields, attributes, properties, and methods** of the Person class. So, the new Employee class includes both the first_name, last_name properties, as well as the new ID properties. We test this with the code in Listing 10.

```
# ---------------------------------------------------------- #
# Title: Listing 10
# Description: A main module for testing
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ---------------------------------------------------------- #
if __name__ == "__main__":
    from Listing09 import Employee as Emp  # Employee class only!
    import Listing07 as P  # processing classes
else:
    raise Exception("This file was not created to be imported")

# Test data module
objP1 = Emp(1, "Bob", "Smith")
objP2 = Emp(2, "Sue", "Jones")
lstTable = [objP1, objP2]
for row in lstTable:
    print(row.to_string(), type(row))

# Test processing module
P.FileProcessor.save_data_to_file("EmployeeData.txt", lstTable)
lstFileData = P.FileProcessor.read_data_from_file("EmployeeData.txt")
lstTable.clear()  # Clear list before loading from file
for line in lstFileData: # Convert list of string to Employee objects
    lstTable.append(Emp(line[0], line[1], line[2].strip()))
for row in lstTable: # show Employee data from refilled list
print(row.to_string(), type(row))
```

Listing 10

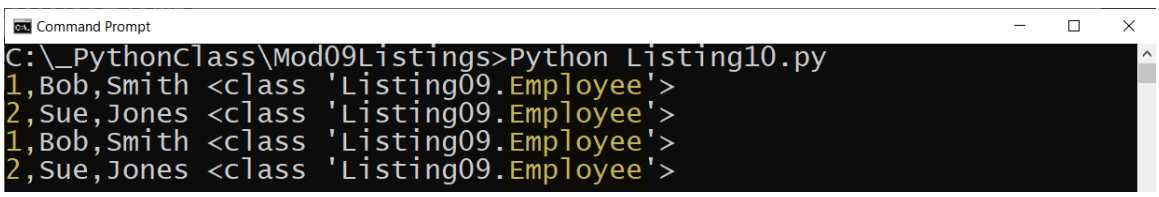

Figure 7. The results of Listing 10

Lab 9-2

In this lab, you modify a script module to add an "Employee" class. You then add code to a test harness and verify that it works.

Note: I have provided the listing code in a file called "Mod08Listings.zip" if you do not want to type the code yourself.

1. Add the code from listing 9 to the "DataClasses.py" script to create the "Employee" class.

2. Add the code in Listing 10 to the TestHarness.py script to test the "FileProcessor" and "Employee" classes.

7. Verify that the results look like those shown in Figure 7.

UML

Unified Modeling Language (UML) is a standard way of modeling relationships between software components. UML consists of many types of modeling diagrams, but let's look at three of the most common ones.

Class Diagrams

Class diagrams show relationships between classes. In Figure 8, you can tell that the Employee class is a child of the Person class by looking at the direction the arrow points (It always points from Child to Parent). Class Diagrams also shows the properties and methods included in each class. These diagrams also may show the data types of each properties and method parameters.

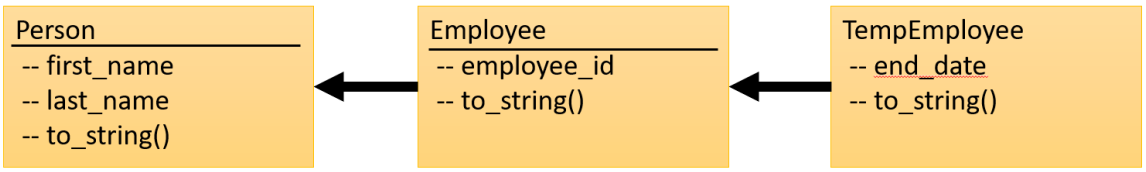

Figure 8. A simple class diagram

Use Case Diagrams

"Use Case" diagrams are another popular way to show how components within the software, and the actions they perform, will be used by humans and other software. The person or software using the component is called "Actor," even when the actor is software, it is represented with a stick figure, as shown in Figure 9.

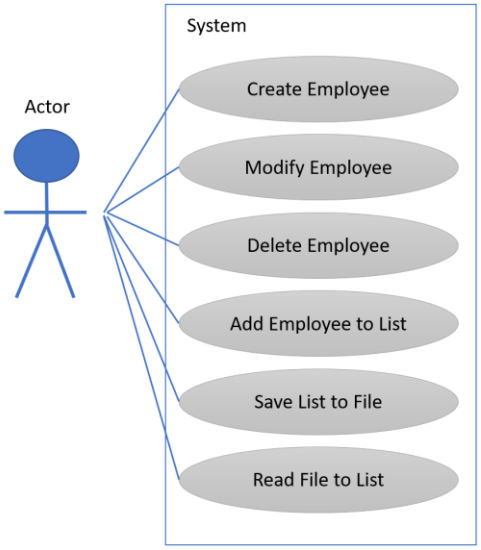

Figure 9. A simple use case diagram

Composition Diagrams

The third common UML document we look at uses ovals to represent objects made from classes. The lines between the ovals to indicate a relationship between the objects. The two most common relationship types are Composition and Aggregation (Figure 10). A filled diamond arrow indicates Composition, while a empty diamond represents Aggregation.

Figure 10. As simple composition diagram

Tip: Here is a good definition and article about the difference.

Aggregation *implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). ...* ***Composition*** *implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). (****https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/****, 2019)*

The numbers used in the diagram indicate “Cardinality” or the expected number of objects in the relationship. For example, “0:1” near the EmployeeList object means that there will be either zero or one EmployeeList object connected to Employee objects. The “1:N” near the Employee object indicates that the must be at least 1, but possibly many (of an undefined Number). Composition represents a strong bond, and in this case, the numbers indicate that without at least one Employee the EmployeeList is not needed.

Creating an Application (an example)

To create an application, you identify what you are trying to accomplish and plan it out. You may have to modify your UML documents as you go through the design process.

Planning

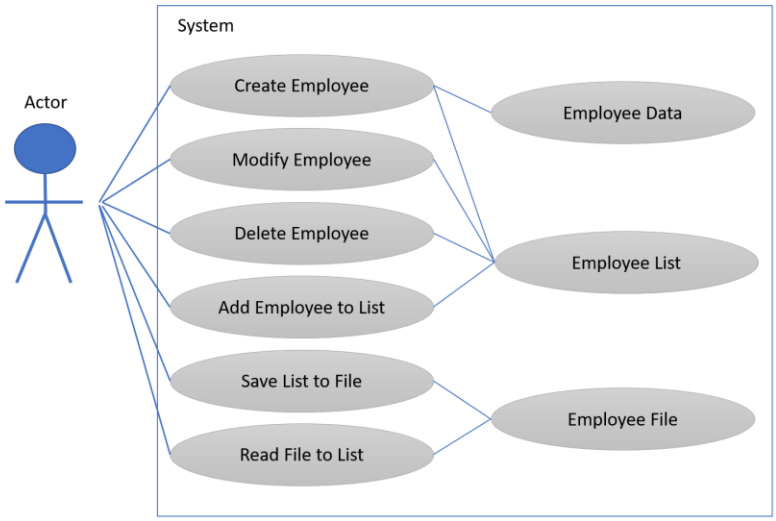

Figure 11. A modified use case diagram

You identify what data, processing and IO tasks you need to accomplish your goal.

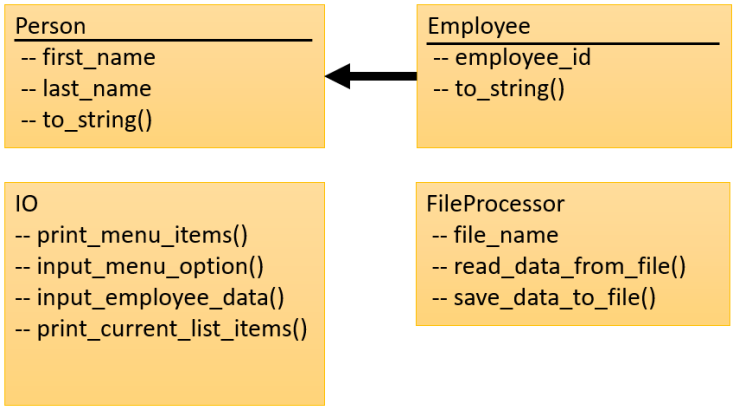

Figure 12. A modified class diagram

You create an object diagram to outline the objects you need.

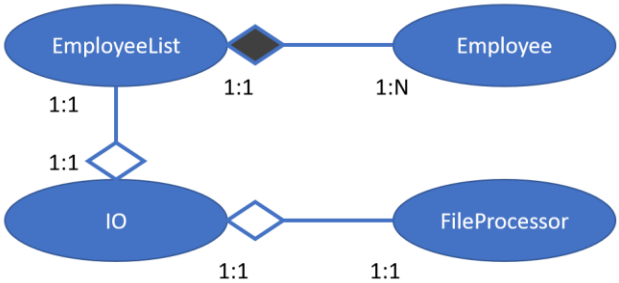

Figure 12. A modified composition diagram

Implementing

Once you have your plan, you decide if you need to create new scripts and modules or if you can reuse existing ones. For our application, we can reuse the DataProcessing, Person, and Employees modules, but need to build a IO module (Listing 11).

```
# ------------------------------------------------------ #
# Title: IO Classes
# Description: A module of IO classes
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------ #
if __name__ == "__main__":
    raise Exception("This file is not meant to ran by itself")
else:
    import DataClasses as DC

class EmployeeIO:
    """  A class for performing Employee Input and Output

    methods:
        print_menu_items():

        print_current_list_items(list_of_rows):

        input_employee_data():

    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class:
    """
    @staticmethod
    def print_menu_items():
        """ Print a menu of choices to the user  """
        print('''
        Menu of Options
        1) Show current employee data
        2) Add new employee data.
        3) Save employee data to File
        4) Exit program
        ''')
        print()  # Add an extra line for looks

    @staticmethod
    def input_menu_options():
        """ 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 print_current_list_items(list_of_rows: list):
        """ Print the current items in the list of Employee rows

        :param list_of_rows: (list) of rows you want to display
        """
        print("******* The current items employees are: *******")
        for row in list_of_rows:
            print(str(row.employee_id)
                  + ","
                  + row.first_name
                  + ","
                  + row.last_name)
        print("*****************************************")
        print()  # Add an extra line for looks

    @staticmethod
    def input_employee_data():
        """ Gets data for an employee object

        :return: (employee) object with input data
        """
        try:
            employee_id = (input("What is the employee Id? - ").strip())
            first_name = str(input("What is the employee First Name? - ").strip())
            last_name = str(input("What is the employee Last Name? - ").strip())
            print()  # Add an extra line for looks
            emp = DC.Employee(employee_id,first_name,last_name)
        except Exception as e:
            print(e)
        return emp
```

Listing 11

Once that is done, you test your modules. Listing 12 shows how you might add more code to the existing test harness to make sure that all three modules are working.

```
 # ------------------------------------------------------- #
# Title: Listing 12
# Description: A main module for testing
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# ------------------------------------------------------- #
if __name__ == "__main__":
    from DataClasses import Employee as Emp
    from ProcessingClasses import FileProcessor as Fp
```

```
    from IOClasses import EmployeeIO as Eio
else:
    raise Exception("This file was not created to be imported")

# Test data module
objP1 = Emp(1, "Bob", "Smith")
objP2 = Emp(2, "Sue", "Jones")
lstTable = [objP1, objP2]
for row in lstTable:
    print(row.to_string(), type(row))

# Test processing module
Fp.save_data_to_file("EmployeeData.txt", lstTable)
lstFileData = Fp.read_data_from_file("EmployeeData.txt")
lstTable.clear()
for line in lstFileData:
    lstTable.append(Emp(line[0], line[1], line[2].strip()))
for row in lstTable:
    print(row.to_string(), type(row))

# Test IO module
Eio.print_menu_items()
Eio.print_current_list_items(lstTable)
print(Eio.input_employee_data())
print(Eio.input_menu_options())
```

Listing 12

After you test all the modules, it is time to create a main module to complete your application. Since this is what you do in this module's assignment, I have only included the comments in Listing 13.

```
# ------------------------------------------------------------------------ #
# Title: Assignment 09
# Description: Working with Modules
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# RRoot,1.1.2030,Added pseudo-code to start assignment 9
# <Your Name>,<Today's Date>,Modified code to complete assignment 9
# ------------------------------------------------------------------------ #
# TODO: Import Modules

# Main Body of Script  ------------------------------------------------------ #
# TODO: Add Data Code to the Main body
# Load data from file into a list of employee objects when script starts
# Show user a menu of options
# Get user's menu option choice
    # Show user current data in the list of employee objects
    # Let user add data to the list of employee objects
    # let user save current data to file and exit program

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

Listing 13

Lab 9-3

In this lab, you create a script module to add an "EmployeeIO" class. You then add code to a test harness and verify that it works.

Note: I have provided the listing code in a file called "Mod08Listings.zip" if you do not want to type the code yourself.

1. Add the code from listing 11 to a new "IOClasses.py" script module to create the "EmployeeIO" class.

2. Add the code in Listing 12 to the TestHarness.py script to test the "FileProcessor," "Employee," and "EmployeeIO" class.

7. Verify that the results look like those shown in Figure 13.

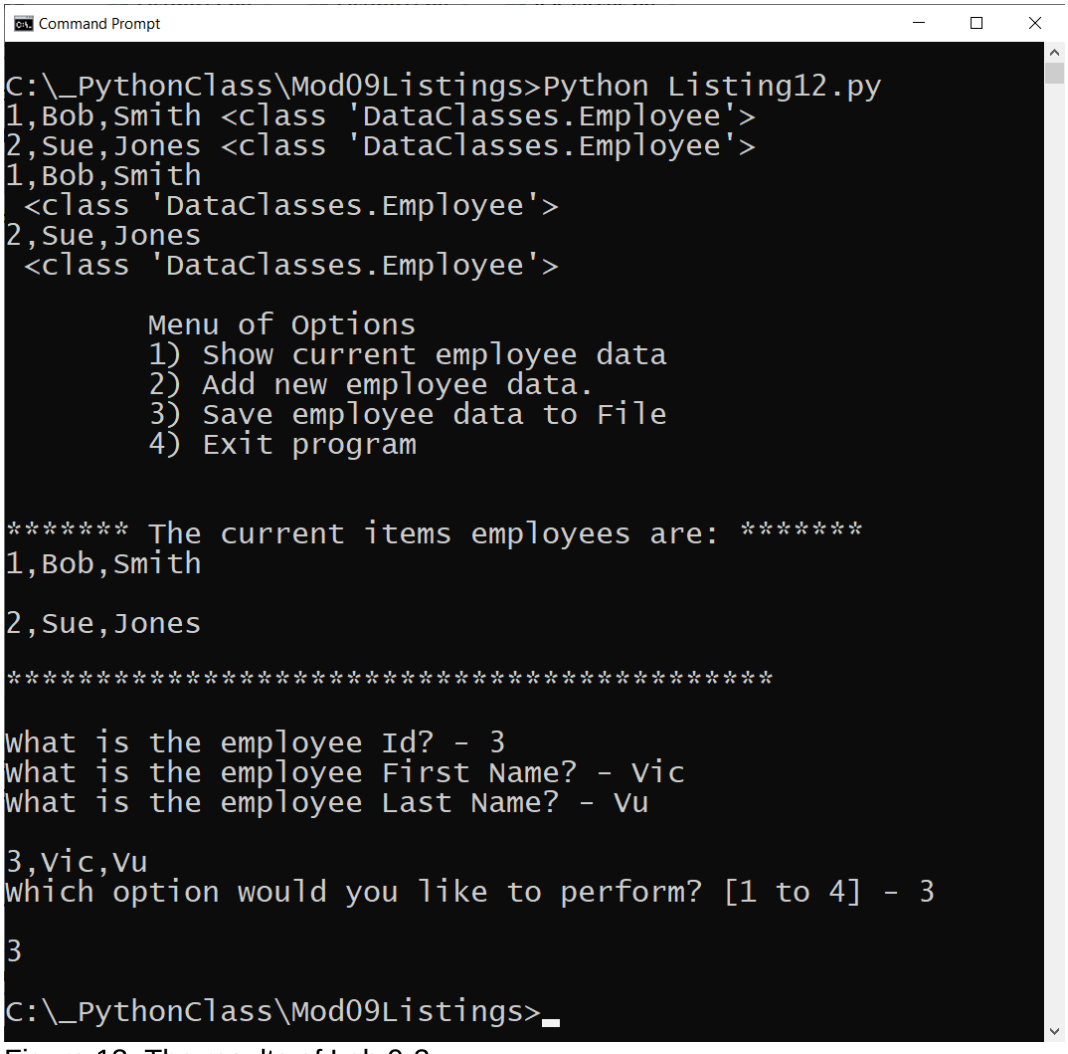

Figure 13. The results of Lab 9-3

Summary

In this module, we looked at how to use script modules to organize your classes.

At this point, you should try and answer as many of the following questions as you can from memory, then review the subjects that you cannot. Imagine being asked these questions by a co-worker or in an interview to connect this learning with aspects of your life.

- What is the difference between a class and module?
- What is the "main" module?
- What is the "__name__ " System Variable?
- How do you connect one module to another?
- What is class inheritance?
- What are three types of UML diagrams?

When you can answer all of these from memory, it is time to complete the module's assignment and move on to the next module.