## Lab 06: Explore Python Classes

### Case Study

TechCom Networks, a major telecommunications company in Ireland, delivers broadband and fiber services to millions of customers. With a workforce of over 1,000 professionals, the company is committed to improving its infrastructure and software development practices through automation. Recognizing the rising demand for digital efficiency, TechCom aimed to modernize its internal systems by enhancing its use of Python and object-oriented programming.

### Business Challenge

TechCom Networks encountered several challenges in its day-to-day technical operations. Development teams often worked in inconsistent Python environments, leading to version conflicts and delays in automation deployment. Meanwhile, network engineers relied on manual procedures to back up router configurations and apply changes to network devices, an approach that was not only time-consuming but also prone to error. The lack of reusable code and organized structure made it difficult to maintain systems or scale automation efforts effectively. To resolve these issues, TechCom onboarded a certified Cisco DevNet Associate to lead the transition toward more structured Python practices, with a focus on automation, maintainability, and collaboration.

### Solution

The company introduced a structured programming model based on Python’s core principles, beginning with the use of functions and methods to separate reusable logic from object-specific behavior. Functions were developed to perform standalone tasks such as logging, data formatting, or sending messages, while methods were tied to objects representing network devices, allowing actions to be performed in a modular and repeatable manner. Python classes were then introduced to model network entities like routers and switches, encapsulating their properties and behaviors. Each device could now be represented as an object with its attributes and methods. Additionally, Python virtual environments were adopted to ensure consistent dependencies across development machines, solving the version conflict issues. This overall approach improved operational efficiency, reduced human error, and laid the groundwork for scalable network automation.

Follow these steps to complete the lab:

1. Review Functions, Methods, and Classes
2. Define a Function
3. Define a Class with Methods
4. Review the circleClass.py Script

|  |  |  |
| --- | --- | --- |
| **// Review Functions, Methods, and Classes**  1. Launch the DEVASC VM.  2. Double-click on a VS Code icon to open it.    3. Create a new file and save it with the name **my\_function.py** and save it.    4. Define a function using the correct Python syntax by typing **def functionName():,** making sure to include the parentheses. Inside the function, add a simple command like **print("Hello from the function!")** to demonstrate functionality. After the function definition, call the function by writing functionName() on a new line.   |  | | --- | | # Define the function  def functionName():  ...blocks of code...  # Call the function  functionName() |     5. Open the terminal and use the command **cd labs/devnet-src/python**. Next, run the Python script by entering **python3 my\_function.py**. If the function is correctly defined and called inside the file, the terminal will display the output such as **"Hello from the function!",** indicating that your function executed successfully.    6. To perform Step, start by opening Visual Studio Code and creating a new Python file named **my\_class.py**. Begin by defining a class using the class keyword followed by the class name and a colon. Inside the class, define three methods using the def keyword, making sure to include self as a parameter for each method. Add simple print statements inside each method to indicate which method is being executed. After defining the class, instantiate it by creating an object from the class. Then, call each method using the object name followed by a dot and the method name with parentheses.   |  | | --- | | # Define the class  class className  # Define a method  def method1Name  ...blocks of code  # Define another method  def method2Name  ...blocks of code  # Define yet another method  def method3Name  ...blocks of code  # Instantiate the class  myClass = className()  # Call the instantiation and associated methods  myClass.method1Name()  myClass.method2Name()  myClass.method3Name() |     7. Save the file as **my\_class.py** inside the path: **labs/devnet-src/python.** Make sure the file extension is .**py.**    8. To run your saved Python script, open the terminal in your DEVASC VM and type the command **python3 my\_class.py**, then press **Enter**. This will execute the script and prompt you to input |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **// Define a Function**  1. To define a function with arguments, begin by opening a new text file and saving it as **myCity.py** inside the **~/labs/devnet-src/python** directory.    2. In this file, define a function named myCity that takes one argument called city. Inside the function, include a print statement that displays a message such as "I live in [city]." Next, call the myCity function multiple times, each time passing a different city name as an argument—for example, "Austin", "Tokyo", and "Salzburg". This demonstrates how a single function can produce different outputs based on the input it receives.   |  | | --- | | def myCity(city):  print("I live in " + city + ".")  myCity("Austin")  myCity("Tokyo")  myCity("Salzburg") |     3. To run your saved Python script, open the terminal in your DEVASC VM and type the command **python3 myCity.py**, then press Enter. This will execute the script and prompt you to input    **//Define a Class with Methods**  1. To define a class with methods, begin by opening a new text file and saving it as **myLocation.py**    2. In this file, define a class called Location using the class keyword followed by a colon. Inside the class, define the \_\_init\_\_() method with self, name, and country as parameters. Within this method, assign the values to the instance variables using self. name = name and self.country = country. Next, instantiate the class by creating an object—for example, loc = Location("Your\_Name", "Your\_Country"), replacing the placeholders with your actual name and country. To confirm the object's properties were set correctly, add print(loc.name) and print(loc.country) to display the values. Lastly, use print(type(loc)) to verify that loc is indeed an instance of the Location class.   |  | | --- | | class Location:  def \_\_init\_\_(self, name, country):  self.name = name  self.country = country  loc = Location("Your\_Name", "Your\_Country")  print(loc.name)  print(loc.country) |     3. Save the file and run it using the command **python3 myLocation.py**, and the output should show your name, country, and the class type.    4. To add a method to the Location class, open the **myLocation.py** file and remove any instantiation or print statements outside the class definition. Leave only the Location class and its \_\_init\_\_() method. Then, place your cursor at the end of the line that defines self. country = country, press Enter twice, and backspace once to set the correct indentation level. Next, define a method named myLocation using the def keyword and the self parameter so it can access the instance variables. Inside the method, write a single-line print statement that displays: “Hi, my name is [name] and I live in [country].” using the self. name and self.country variables. Once added, press Enter twice and backspace twice to return to the left margin.   |  | | --- | | class Location:  def \_\_init\_\_(self, name, country):  self.name = name  self.country = country  def myLocation(self):  print("Hi, my name is " + self.name + " and I live in " + self.country + ".") |     4. Save and Run the file. Since the method is defined but not yet called, there will be no output at this stage.    5. To complete the process of instantiating the Location class multiple times and calling the myLocation() method, begin by opening your myLocation.py script. Below the class definition, add code to create the first instance of the class using' loc1 = Location("Tomas", "Portugal") ', followed by a call to' loc1.myLocation() ' to display the message. Save and run the script, and you should see the output confirming the method works. Next, add two more instances of the class: loc2 = Location("Ying", "China") and loc3 = Location("Amare", "Kenya"), and call their myLocation() methods. Finally, create a fourth instance named your\_loc with your name and country, such as your\_loc = Location("Your\_Name", "Your\_Country"), and call your\_loc.myLocation() as well.   |  | | --- | | lass Location:  def \_\_init\_\_(self, name, country):  self.name = name  self.country = country  def myLocation(self):  print("Hi, my name is " + self.name + " and I live in " + self.country + ".")  loc1 = Location("Thomas", "Portugal")  loc2 = Location("Ying", "China")  loc3 = Location("Amare", "Kenya")  loc1.myLocation()  loc2.myLocation()  loc3.myLocation()  your\_loc = Location("Your\_Name ", "Your\_Country")  your\_loc.myLocation() |     5. Save and run the script again, and the terminal should display four personalized messages, each corresponding to the object created and its data    6. Once the Location class is created, it can be used to make as many objects as needed. Each time a new object is created from the class, different values can be provided for the name and country. For example, the class is first used to create an object for a person named Tomas from Portugal. When the method inside the class is called, it prints a sentence introducing Tomas and his country.  After confirming that the method works correctly, more objects are created to represent different individuals. Two more people are added—Ying from China and Amare from Kenya. Each of them is represented by a separate object of the Location class. Their method is also called, which prints their names and countries in the same way.  Finally, a fourth object is created with your own name and country. This demonstrates how the class can be used multiple times with different data, and how the same method can produce personalized results for each instance. This part of the lab helps understand how object-oriented programming makes code reusable and organized.   |  | | --- | | # Then define a method that belongs to the class. The method’s  # purpose is to print a sentence that uses the variables.  class Location:  def \_\_init\_\_(self, name, country):  self.name = name  self.country = country  def myLocation(self):  print("Hi, my name is " + self.name + " and I live in " + self.country + ".")  # First instantiation of the Location class  loc1 = Location("Tomas", "Portugal")  # Call a method from the instantiated class  loc1.myLocation()  # Three more instantiations and method calls for the Location class  loc2 = Location("Ying", "China")  loc3 = Location("Amare", "Kenya")  loc2.myLocation()  loc3.myLocation()  your\_loc = Location("Your\_Name", "Your\_Country")  your\_loc.myLocation() |     7. Save and run the script again, and the terminal should display four personalized messages, each corresponding to the object created and its data |

|  |  |  |
| --- | --- | --- |
| **// Define a Function**  1. To define a function with arguments, begin by opening a new text file and saving it as **myCity.py** inside the **~/labs/devnet-src/python** directory.    2. In this file, define a function named myCity that takes one argument called city. Inside the function, include a print statement that displays a message such as "I live in [city]." Next, call the myCity function multiple times, each time passing a different city name as an argument—for example, "Austin", "Tokyo", and "Salzburg". This demonstrates how a single function can produce different outputs based on the input it receives.   |  | | --- | | def myCity(city):  print("I live in " + city + ".")  myCity("Austin")  myCity("Tokyo")  myCity("Salzburg") |     3. To run your saved Python script, open the terminal in your DEVASC VM and type the command **python3 myCity.py**, then press Enter. This will execute the script and prompt you to input.    **// Review the circleClass.py Script**  1. Begin by opening a new text file and saving it as myLocation.py    2. How to build a class that calculates the circumference of a circle and displays the result. This class contains three methods. The first is the \_\_init\_\_() method, which is used to set the radius value when a new object of the class is created. The second method is circumference(), which uses the radius to compute the circumference of the circle and stores the result. The third method, printCircumference(), prints out a message showing the radius and the calculated circumference. To ensure the message displays correctly, the values are converted to strings using the str() function, because Python cannot combine numbers and text directly in a print statement. In this example, the Circle class is used three times to create different circle objects, each with a different radius, and each one prints its circumference.   |  | | --- | | # Given a radius value, print the circumference of a circle.  # Formula for a circumference is c = pi \* 2 \* radius  class Circle:  def \_\_init\_\_(self, radius):  self.radius = radius  def circumference(self):  pi = 3.14  circumferenceValue = pi \* self.radius \* 2  return circumferenceValue  def printCircumference(self):  myCircumference = self.circumference()  print ("Circumference of a circle with a radius of " + str(self.radius) + " is " + str(myCircumference))  # First instantiation of the Circle class.  circle1 = Circle(2)  # Call the printCircumference for the instantiated circle1 class.  circle1.printCircumference()  # Two more instantiations and method calls for the Circle class.  circle2 = Circle(5)  circle2.printCircumference()  circle3 = Circle(7)  circle3.printCircumference() |     3. To review the circleClass.py script, understand that it demonstrates how to use a Python class to calculate and display the circumference of a circle using object-oriented programming. The Circle class defines three methods: the \_\_init\_\_() method, which initializes each object with a radius value; the circumference() method, which calculates the circumference using the formula 2πr2 \pi r2πr and returns the result; and the printCircumference() method, which prints a descriptive message. To avoid errors when printing numbers in a string, the script uses str() to convert numerical values to strings. The class is instantiated three times with different radius values, showing that the class can be reused with various data to perform the same computation. |