MODULE 5

INTERMEDIATE ELEMENTS OF PYTHON 2

MODULE OVERVIEW

In this module, you will learn about functions, modules, sets, and arrays in Python. Functions are blocks of code that perform specific tasks and can be reused throughout a program. Understanding functions is essential for writing organized and efficient Python code. Modules are files containing Python code that can be imported and used in other Python scripts. Understanding modules is essential for organizing and reusing code effectively, as they allow you to break down complex programs into smaller, manageable parts. Sets are collections of unique elements, while arrays (represented using lists) are ordered collections of elements. Understanding sets and arrays is essential for efficiently managing data and performing various operations on collections.

MODULE LEARNING OBJECTIVES

By the end of this module, you will be able to:

- Understand the concept of modules and their role in code organization.
- Learn how to create and use modules.
- Learn how to define and call functions in Python.
- Apply functions to solve practical problems and improve code readability.
- Understand the concept of sets and arrays in Python.
- Learn how to create and manipulate sets and arrays.

LEARNING CONTENTS

5.1 Functions

Functions in programming are blocks of code that perform a specific task or a set of tasks. They allow you to break down complex problems into smaller, more manageable pieces of code, which makes your code more organized, easier to read, and reusable. Functions allow developers to write code once and use it multiple times throughout a program, promoting code reusability and modularity.

5.1.1 Characteristics of a Function

a) Functions are defined using the `def` keyword, followed by the function name, parentheses, and a colon. To use a function, you call it by its name followed by parentheses.
The function body is indented below the definition.

Example:

```
def greet():
    print("Hello, World!")
greet()
```

Output:

```
Hello, World!
```

b) Functions can accept input data known as parameters or arguments. These parameters are specified within the parentheses when defining the function. To use a function that requires parameter/s, you call it by its name followed by parentheses, passing any required arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

Example 1:

```
def greet(name):
    print(f"Hello, {name}!")

greet("Mark")
greet("Denver")
```

Output:

```
Hello, Mark!
Hello, Denver!
```

Example 2:

```
def greet(name):
    print(f"Hello, {name}!")

nameInput = input("Enter a name: ")
greet(nameInput)
```

Output:

```
Enter a name: Mark Denver
Hello, Mark Denver!
```

c) Functions can optionally return a result using the return keyword. This allows the function to pass data back to the caller. By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you must call the function with 2 arguments, not more, and not less.

Example 1:

```
def add_numbers(a, b):
    return a + b

add_numbers(1, 3)
add_numbers(7, 9)
```

Output:

```
4
16
```

Example 2:

```
def add_numbers(a, b):
    return a + b
result = add_numbers(5, 7)
print(result)
```

Output:

```
12
```

5.1.2 Passing a List as an Argument

In Python, you can pass a list as an argument to a function. When you pass a list to a function, the function receives a reference to the original list, allowing it to access and modify the list directly. This feature makes it convenient to work with collections of data and perform operations on lists within functions.

Here's how you can pass a list as an argument to a function.

Example 1:

```
def print_list(lst):
    for item in lst:
        print(item)

my_list = [1, 2, 3, 4, 5]
print_list(my_list)
```

Output:

```
1
2
3
4
5
```

5.2 Modules

In Python, a module is a file that contains Python code, such as functions, classes, and variables. Modules serve to organize code and promote code reusability by breaking down large programs into smaller, more manageable components. They allow you to logically group related code together and separate concerns in your projects.

Python modules can be of two types:

1. **Built-in Modules**
   In Python, there are numerous built-in modules that come pre-installed with the Python interpreter. These modules provide a wide range of functionalities, such as mathematical operations, working with dates and times, file management, system-related tasks, and more. You can use these modules directly in your Python programs without the need for additional installations.
   Here are some commonly used built-in modules in Python:

a) **math**: Provides mathematical functions and constants.

Example :

```
import math

radius = 5
area = math.pi * radius**2
print(area)
```

Output:

```
78.53981633974483
```

b) **random**: Offers functions for generating random numbers.

Example :

```
import random
random_number = random.randint(1, 10)
print(random_number)
```

Output:

```
# Random numbers from 1-10
```

c) **datetime**: Allows manipulation of dates and times.

Example :

```
import datetime
current_time = datetime.datetime.now()
print(current_time)
```

Output:

```
2023-07-27 21:18:54.859310
```

2. User-Defined Modules

User-defined modules in Python are custom modules created by developers to organize their code into separate files and promote code reusability. Instead of writing all the code in a single file, you can break it down into logical components and store them in separate Python files. These files act as modules, and you can import and use the functions, classes, and variables defined in these modules in other Python scripts.

To create a user-defined module, follow these steps:

Step 1 Create a Python file with a .py extension. This will be your user-defined module.

Step 2 Write your Python code in the file, defining functions, classes, or variables as needed.

Step 3 Save the file with a meaningful name. This name will be used when importing the module in other scripts.

Example of a user-defined module:

Create a file named my_module.py with the following content:

```
def say_hello():
    print("Hello, this is my custom module.")

def add_numbers(a, b):
    return a + b
```

In another Python script, you can import and use the functions from my_module:

```
import my_module

my_module.say_hello()
result = my_module.add_numbers(5, 3)
print(result)
```

Output:

```
Hello, this is my custom module.
8
```

You can also use the from keyword to import specific functions directly:

```
from my_module import say_hello, add_numbers

say_hello()
result = add_numbers(5, 3)
print(result)
```

Output:

```
Hello, this is my custom module.
8
```

5.3 Sets

In Python, sets are a built-in data type used to store collections of unique elements. A set is an unordered collection of elements, meaning that the elements have no specific order, and they are defined using curly braces {} or the set() constructor. Sets do not allow duplicate elements. If you try to add an element that already exists in the set, it will not be duplicated. The elements in a set have no specific order. This means that sets do not support indexing or slicing. It is also mutable, so you can add or remove elements after they are created.

5.3.1 Creating a Set

In Python, you can create a set using curly braces {} or the set() constructor. To create a set, simply place the elements you want to include in the set inside the curly braces, separated by commas.

Example:

```
mySet = {1, 2, 3, 4, 5}
myStringSet = {"apple", "banana", "orange"}
empty_set = set()
```

Note: You cannot create an empty set using {} because it will create an empty dictionary

5.3.2 Set Length

You can determine the length of a set by using a len() function.

Example:

```
mySet = {1, 2, 3, 4, 5}
result = len(mySet)
print(result)
```

Output:

```
5
```

5.3.3 Accessing Set

In Python, you can access elements in a set using iteration or check for membership using the `in` keyword. However, sets are unordered, meaning that there is no specific index or position for each element in the set. Therefore, you cannot access elements in a set using indexing or slicing like you would with a list or tuple.

Here's how you can access elements in a set:

1. **Iteration** - You can use a for loop to iterate over the elements in the set. Since sets are unordered, the order of iteration is not guaranteed.

   Example:

   ```
   mySet = {1, 2, 3, 4, 5}
   for val in mySet:
       print(val)
   ```

   Output:

   ```
   1
   2
   3
   4
   5
   ```

2. **Checking Membership** - You can use the in keyword to check if a specific element is present in the set.

Example:

```
mySet = {1, 2, 3, 4, 5}

print(1 in mySet)
print(7 in mySet)
```

Output:

```
True
False
```

Remember that sets are optimized for membership tests and provide faster lookups compared to lists. Since sets are unordered and do not have indices, if you need to access elements by their position or index, you should use a list or tuple instead. Sets are more suitable for tasks that involve checking membership, removing duplicates, or performing set operations like union, intersection, and difference.

5.3.4 Modifying Sets

a) Adding Set Items

In Python, you can add items to a set using the add() method or by performing set operations like union.

a.1. add() method - method allows you to add **one** element at a time to the set. If the element you are trying to add is already in the set, the add() method will have no effect, as sets do not allow duplicate elements.

Example:

```
mySet = {1, 2, 3, 4, 5}
mySet.add(6)
mySet.add(7)
print(mySet)
```

Output:

```
{1, 2, 3, 4, 5, 6, 7}
```

As you can see, the set can have one element added to it at a time using the add() method. The add() method has no effect if the element you are attempting to add is already present in the set since sets do not permit duplicate elements.

a.2. union() method - You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple.

Example:

```
mySet1 = {1, 2, 3, 4, 5}
mySet2 = {6, 7, 8}
finalSet = mySet1.union(mySet2)
print(finalSet)
```

Output:

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Or you can also use `|` (union operator) to combine sets.

a.3. union() method - You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple.

Example:

```
mySet1 = {1, 2, 3, 4, 5}
mySet2 = {6, 7, 8}
finalSet = mySet1 | mySet2
print(finalSet)
```

Output:

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

a.4. update() method – This method is used to modify a set by adding elements from another iterable (like another set, list, tuple, or any other iterable). It performs a union operation, adding all the unique elements from the given iterable to the original set.

Example 1:

```
mySet1 = {1, 2, 3, 4, 5}
mySet2 = {6, 7, 8}
mySet1.update(mySet2)
print(mySet1)
```

Output:

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Example 2:

```
mySet1 = {1, 2, 3, 4, 5}
myList= [6, 7, 8]
mySet1.update(mySet2)
print(mySet1)
```

Output:

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Remember that sets automatically handle duplicates, so adding an element that already exists in the set will have no effect.

b) Changing Set Items

In Python, you cannot change individual items in a set directly. Sets are mutable, but they do not support item assignment because they are unordered collections of unique elements. If you want to change the elements in a set, you must remove the existing elements and then add new ones.

c) Removing Set Items

In Python, you can remove items from a set using various methods. Sets provide two methods specifically designed for removing elements: remove() and discard(). Additionally, you can use the pop() method, but it's important to note that sets are unordered, so pop() will remove an arbitrary item from the set.

c.1. remove() - This method removes a specific element from the set. If the element is not present in the set, it will raise a KeyError.

Example:

```
mySet = {1, 2, 3, 4, 5}
mySet.remove(3)
print(mySet)
```

Output:

```
{1, 2, 4, 5}
```

c.2. discard() - This method removes a specific element from the set if it exists. If the element is not present, it does nothing and does not raise an error.

Example:

```
mySet = {1, 2, 3, 4, 5}
mySet.discard(3)
print(mySet)
```

Output:

```
{1, 2, 4, 5}
```

c.3. pop() - This method removes and returns an arbitrary element from the set. As sets are unordered, the choice of which element is removed is arbitrary.

Example:

```
mySet = {1, 2, 3, 4, 5}
removedElement = mySet.pop()
print(removedElement)
print(mySet)
```

Output:

```
1
{2, 3, 4, 5}
```

Remember that sets are unordered collections, so the order of removal with the pop() method may vary each time you call it. When removing elements from a set, be cautious to avoid KeyError when using remove(). If you're unsure whether an element exists in the set, you can use discard() to remove it safely without raising an error.

5.4 Arrays

In Python, arrays are used to store collections of items of the same data type. They are like lists but have a fixed size and must contain elements of the same type. In addition, they are more memory-efficient compared to lists when dealing with many elements of the same data type. However, lists are more versatile because they can contain elements of different data types and can be resized dynamically. Depending on your specific use case, you can choose between arrays and lists to store collections of data. Python arrays are provided by the array module, which is part of the Python standard library.

5.4.1 Creating an Array

To use arrays in Python, you first need to import the array module. Then, you can create an array by specifying the data type and the initial elements. Here's the general syntax for creating an array:

Example:

```
import array
array_name = array.array('data_type',
[initial_elements])
```

Where:

- **data_type**: The type code representing the data type of the elements in the array. For example, 'i' for integers, 'f' for floating-point numbers, 'd' for double-precision floating-point numbers, etc. (You can find a list of available type codes in the Python documentation for the array module).
- **initial_elements:** A list or iterable containing the initial elements of the array.

Here are some examples of using arrays in Python:

Example 1 - Creating an array of integers

```
import array

numbers = array.array('i', [1, 2, 3, 4, 5])
```

Example 2 - Creating an array of floating-point numbers

```
import array

scores = array.array('f', [98.5, 76.2, 84.7])
```

5.4.2 Array Length

You can determine the length of a set by using a len() function.

Example

```
import array

numbers = array.array('i', [1, 2, 3, 4, 5])
print(len(numbers))
```

Output:

```
5
```

5.4.3 Accessing Arrays

In Python, you can access elements in an array using indexing. Arrays are zero-indexed, which means the first element has an index of 0, the second element has an index of 1, and so on. To access an element in an array, use the array name followed by the index of the element enclosed in square brackets [].

Example

```
import array

numbers = array.array('i', [1, 2, 3, 4, 5])
print(numbers[0])
print(numbers[2])
print(numbers[3])
```

Output:

```
1
3
4
```

Keep in mind that the index you provide must be within the valid range of the array. If you try to access an index that is out of range, it will raise an IndexError. For example, if the array has a length of 5, valid indices are 0 to 4, and accessing numbers[5] would raise an IndexError.

5.4.4 Modifying Arrays

a) Adding Array Items

In Python, you can add items to an array using the append() method or by concatenating another iterable (like a list, tuple, or array) to the original array. However, it's important to note that Python arrays have a fixed size, so adding elements one by one with append() may cause performance issues for large arrays. For dynamic-sized collections, lists are more commonly used.

Here's how you can add items to a Python array:

a.1. append() – You can only append one item at a time otherwise, it will cause you an error.

Example

```
import array
numbers = array.array('i', [1, 2, 3, 4, 5])
numbers.append(6)
numbers.append(7)
print(numbers)
```

Output:

```
array('i', [1, 2, 3, 4, 5, 6, 7])
```

a.2 Array Concatenation – You can concatenate another iterable (like a list) to the array.

Example

```
import array
numbers = array.array('i', [1, 2, 3, 4, 5])
moreNumb = [6, 7]
numbers += array.array('i', numbers)
print(numbers)
```

Output:

```
array('i', [1, 2, 3, 4, 5, 6, 7])
```

Keep in mind that you can only add elements to an array if their data type matches the array's data type. In this example, we used the 'i' type code, indicating that the array can only store integers. If you try to add elements of a different data type, it will raise a TypeError.

b) Changing Array Items

In Python, you can change the value of elements in an array using indexing. Arrays are mutable, so you can modify individual elements after the array is created.

Here's how you can change array items in Python:

Example

```
import array

numbers = array.array('i', [1, 2, 3, 4, 5])
numbers[1] = 10
numbers[3] = 20
print(numbers)
```

Output:

```
array('i', [1, 10, 3, 20, 5])
```

In the example above, we first import the array module and create an array of integers named numbers. Then, we use indexing to access individual elements and modify their values. We change the second element (2) to 10 and the fourth element (4) to 20.

Remember that arrays in Python are zero-indexed, meaning the first element has an index of 0, the second element has an index of 1, and so on. You can use positive and negative indexing to access and modify elements in the array. Keep in mind that when modifying elements in the array, the new value must be of the same data type as the array. For instance, if the array is created with integers ('i'), you can only assign integer values to its elements.

c) Removing Array Items

In Python, the array module does not provide a built-in method to directly remove items from an array like you would with a list. Python arrays are designed to have a fixed size, and they don't support dynamic resizing or item removal. If you need to remove items from a collection and could resize it, you should use a list instead of an array. Lists are more versatile and offer several built-in methods to add, remove, and modify elements.

5.4.5 Looping Arrays

In Python, you can loop through an array using a for loop. The array module provides a data structure called "array," which supports iteration, just like other iterable objects such as lists, tuples, and strings.

Here's how you can loop through an array:

Example

```
import array

numbers = array.array('i', [1, 2, 3, 4, 5])
for num in numbers:
    print(num)
```

Output:

```
1
2
3
4
5
```

You can use the same looping constructs (e.g., for, while) with arrays as you would with other iterable objects in Python. This makes it easy to perform operations on each element in the array, such as

calculations, filtering, or applying functions.

Keep in mind that arrays have a fixed size, and their elements are stored in a contiguous block of memory, making array iteration efficient in terms of memory access and speed. However, if you need more flexibility and functionality, you may prefer to use lists, which offer more built-in methods and are more commonly used in Python for general collections.