**WEEK 2**

**CONTROL FLOW**

**DAY 1: CONTROL FLOW**

**What are control flow statements in Python?**

A program’s control flow is the order in which the program’s code executes.

The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Python has three types of control structures:

·         Sequential - default mode

·         Selection - used for decisions and branching

·         Repetition - used for looping, i.e., repeating a piece of code multiple times.

1    1. Sequential

Sequential statements are a set of statements whose execution process happens in a sequence. The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

A computer screen shot of a black and white screen

Description automatically generated

2. Selection/Decision control statements

 In Python, the selection statements are also known as Decision control statements or branching statements.

 The selection statement allows a program to test several conditions and execute instructions based on which condition is true.

Some Decision Control Statements are:

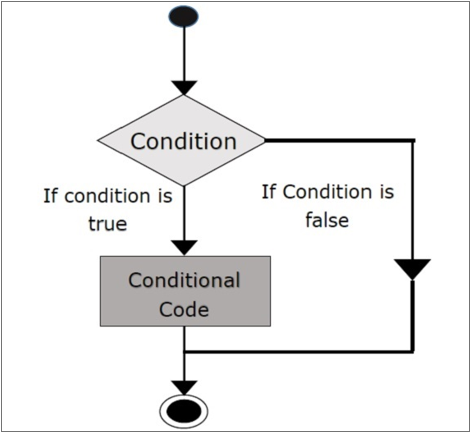
Simple if

if-else

nested if

if-elif-else

Simple if: If statements are control flow statements that help us to run a particular code, but only when a certain condition is met or satisfied. A simple if only has one condition to check.



A black and white screen with orange and white text

Description automatically generated

f-else: The if-else statement evaluates the condition and will execute the body of if if the test condition is True, but if the condition is False, then the body of else is executed.

A screenshot of a computer program

Description automatically generated

**nested if:** Nested if statements are an if statement inside another if statement.

A screenshot of a computer program

Description automatically generated

**if-elif-else:** The if-elif-else statement is used to conditionally execute a statement or a block of statements.

A screenshot of a computer

Description automatically generated

**2.**   **Repetition**

A repetition statement is used to repeat a group(block) of programming instructions.

In Python, we generally have two loops/repetitive statements:

for loop

while loop

for loop: A for loop is used to iterate over a sequence that is either a list, tuple, dictionary, or a set. We can execute a set of statements once for each item in a list, tuple, or dictionary.

A computer screen shot of a computer code

Description automatically generated

**while loop:** In Python, while loops are used to execute a block of statements repeatedly until a given condition is satisfied. Then, the expression is checked again and, if it is still true, the body is executed again. This continues until the expression becomes false.

A screenshot of a computer

Description automatically generated

**DAY 2: PYTHON FUNCTIONS**

**Introduction to Functions**

**1. What is a Function?**

- A function is a reusable block of code in a program that performs a specific task or a set of related tasks. It allows you to encapsulate a sequence of statements into a single entity, which can be invoked multiple times.

2. \*\*Key Concepts:\*\*

- \*\*Function Definition:\*\* A function is defined with a name, a set of parameters (optional), and a block of code. For example:

```python

def greet(name):

print("Hello, " + name)

```

**- \*\*Function Call:\*\*** To execute a function, you "call" it by using its name followed by parentheses. For example:

```python

greet("Alice")

```

**- \*\*Parameters:\*\*** Functions can accept parameters (also known as arguments), which are values passed to the function when it's called. Parameters allow you to pass data into a function. In the above example, `name` is a parameter.

- \*\*Return Value:\*\* Functions can return a value using the `return` statement. This value can be used in other parts of your program. For example:

```python

def add(a, b):

return a + b

```

**3. \*\*Advantages of Functions:\*\***

**- \*\*Modularity:\*\*** Functions promote code modularity by breaking down a program into smaller, manageable pieces. This makes code easier to read, understand, and maintain.

**- \*\*Reusability:\*\*** Once a function is defined, it can be called multiple times from different parts of the program. This reusability reduces redundancy and promotes efficient coding.

**- \*\*Abstraction:\*\*** Functions hide the details of their implementation from the caller. You can use a function without needing to know how it works internally, which simplifies the code.

**4. \*\*Function Signature:\*\***

- A function's signature includes its name and the number and type of its parameters. For example, `greet(name)` is the signature of the `greet` function, and it indicates that it takes one parameter, `name`.

5. **\*\*Function Best Practices:\*\***

- Choose descriptive and meaningful names for your functions.

- Functions should have a single responsibility, performing one specific task.

- Use comments and docstrings to document the purpose and usage of your functions

- Be mindful of function parameters and return values, ensuring they match the function's purpose.

Functions are fundamental to structured programming and are a key building block in the organization and readability of code. They enable code reusability and abstraction, making complex programs more manageable.

**\*\*Using Functions:\*\***

**1. \*\*Function Invocation:\*\***

- To use a function, you invoke (call) it by using its name followed by parentheses. For example:

```python

result = add(5, 3)

```

**2. \*\*Arguments:\*\***

- Arguments are values that you pass to a function when you call it. They provide input to the function. In the example above, `5` and `3` are arguments.

- Python functions can accept multiple arguments, and the order in which you pass them matters.

**3. \*\*Return Values:\*\***

- Functions can return values using the `return` statement. The returned value can be assigned to a variable or used directly in expressions. For example:

```python

def add(a, b):

return a + b

result = add(5, 3) # result will be 8

```

- Not all functions need to return values. Some functions are used for their side effects (e.g., printing to the console) rather than producing a return value.

**4. \*\*Scope:\*\***

- Variables declared inside a function are local to that function and have limited scope. They are not accessible outside the function.

- Variables declared outside of any function have global scope and can be accessed from any part of the program.

**5. \*\*Pass by Value:\*\***

- In Python, when you pass a variable to a function as an argument, you are passing a reference to the object that the variable points to, not the variable itself. This means that changes made to the object within the function can affect the original object.

**6. \*\*Default Arguments:\*\***

- You can set default values for function parameters. If a caller does not provide a value for a parameter, the default value is used. For example:

```python

def greet(name="Guest"):

print("Hello, " + name)

greet() # Output: Hello, Guest

```

**7. \*\*Variable-Length Arguments:\*\***

- Functions can accept a variable number of arguments by using `\*args` and `\*\*kwargs`. `\*args` allows you to pass a variable number of non-keyword arguments, and `\*\*kwargs` allows you to pass a variable number of keyword arguments.

**8. \*\*Function Overloading:\*\***

- Python does not support function overloading in the traditional sense, as it doesn't allow defining multiple functions with the same name but different parameter lists. However, you can achieve similar functionality using default arguments and variable-length arguments.

**9. \*\*Best Practices:\*\***

- Choose descriptive and meaningful function names.

- Document the purpose and usage of functions using comments and docstrings.

- Keep functions focused on a single task or responsibility (the Single Responsibility Principle).

- Use functions to encapsulate and abstract functionality, promoting code modularity and reusability.

Using functions enhances code organization, readability, and maintainability. It allows you to break down complex tasks into smaller, manageable pieces and promotes the reuse of code. Properly designed and documented functions are crucial for effective software development.

**The Random Module**

**1. \*\*Introduction:\*\***

- The `random` module in Python provides functions for generating pseudo-random numbers and working with random data.

**2. \*\*Pseudo-Random Numbers:\*\***

- The numbers generated by the `random` module are pseudo-random, meaning they appear random but are generated using a deterministic algorithm. For true randomness, you'd need external sources like hardware-based random number generators.

**3. \*\*Importing the Module:\*\***

- To use the `random` module, you need to import it using `import random`.

**4. \*\*Common Functions:\*\***

- Some commonly used functions in the `random` module include:

- `random()`: Generates a random float in the range [0.0, 1.0).

- `randint(a, b)`: Generates a random integer in the range [a, b].

- `choice(seq)`: Selects a random element from a sequence.

- `shuffle(seq)`: Shuffles the elements in a sequence randomly.

- `randrange(start, stop, step)`: Generates a random integer within a range with an optional step.

**5. \*\*Setting the Seed:\*\***

- You can use `random.seed(x)` to initialize the random number generator with a specific seed value. Using the same seed produces the same sequence of random numbers, which can be useful for reproducibility.

**6. \*\*Random Sequences:\*\***

- Functions like `sample(seq, k)` and `choices(seq, k)` allow you to generate random samples or choices from a sequence, with or without replacement.

**7. \*\*Random Floats and Integers:\*\***

- `uniform(a, b)` generates a random float within the specified range [a, b], while `randint(a, b)` generates random integers within the range [a, b].

**8. \*\*Random Choices:\*\***

- `choice(seq)` selects a random element from a sequence, while `choices(seq, k)` selects `k` random elements from the sequence with possible duplicates.

**9. \*\*Shuffling Sequences:\*\***

- `shuffle(seq)` shuffles the elements of a mutable sequence like a list in-place, randomly reordering the elements.

**10. \*\*Randomness in Applications:\*\***

- The `random` module is used in various applications, including games, simulations, statistical sampling, and cryptography (for non-security-critical applications).

**11. \*\*Caution:\*\***

- Pseudo-random numbers are not suitable for cryptographic or security applications. For those purposes, the `secrets` module is recommended.

The `random` module is a valuable tool for introducing randomness into programs and is commonly used for scenarios where true randomness is not required. It offers a wide range of functions for generating random data, making it versatile for various programming tasks.

**Recursive Functions**

Recursion in programming involves a method where the solution to a problem depends on the result of the same solution. It's a process in which a function calls itself during its execution. A classic example of recursion is the Fibonacci sequence, where each number is the sum of the previous two numbers.

\*\*Example - Fibonacci Recursive Function:\*\*

```python

def Fibonacci(num):

if num <= 1: # base case

return 1

else:

return num \* Fibonacci(num - 1) # recursive call

In this example, we have a recursive function `Fibonacci` that calculates Fibonacci numbers. It has a base case to handle when `num` is less than or equal to 1 and a recursive call that calculates the Fibonacci number by multiplying the current number with the result of the function for `num - 1`.

When you run the code for the first 11 Fibonacci numbers, it generates the following output:

0 = 1

1 = 1

2 = 2

3 = 6

4 = 24

5 = 120

6 = 720

7 = 5040

8 = 40320

9 = 362880

10 = 3628800

```The output shows the Fibonacci sequence calculated using recursion, with each number being the product of the previous number and the current number, as it works its way down the sequence.

**DAY 3: MODULES**

**INTRODUCTION TO MODULES**

Modules are the pre-defined files that contain the python codes which depict the basic functionalities of class, methods, variables, etc. It consists of different functions, classes in a group of files inside a directory. Modules can also be termed as Libraries. These are basically the pre-defined methods that can be used to make the code more efficient and reduce redundancy.

Modules bind the code and reduce the repetitions of functions frequently used in the code. Thus, it makes the code much clear and easy to understand.

Examples:

· OS

· Time

· Math

· MatPlotlib

**Mechanism of Python Modules**

A pink rectangular sign with black text

Description automatically generated

The moment a module is imported through a program, Python Interpreter fetches the module from either of the following locations:

**Program Directory**

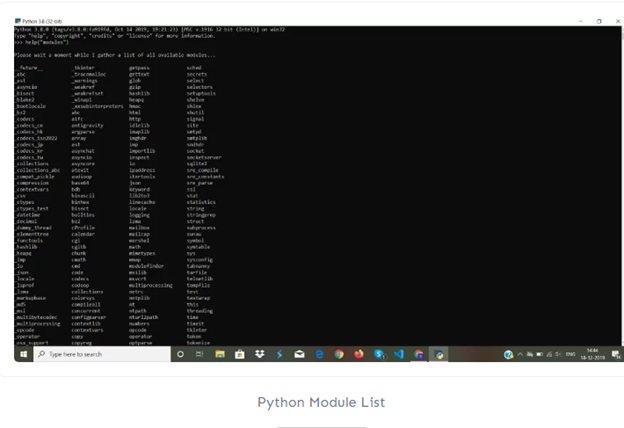
The directory in the PYTHONPATH variable

Default directory

**Listing of Modules**

The list of available modules in Python can be found out by executing the following command in the command prompt (interpreter shell).

>>> help(“module”)



## Importing modules from Python Standard path

**Syntax**:

import module\_name

**Example:**

import math

## Importing Modules from other Sources

To fetch and use modules from other and new sources, we need to install Python PIP.

Python pip is a software that installs python modules from an index or using a manager like Anaconda.

**Run the following command to install modules from new sources using python pip:**

python -m pip3 install module\_name

**Run the following command to install modules from new sources using Ananconda:**

conda install module\_name

**Example: Steps to install NumPy**

python -m pip3 install numpy  
conda install numpy  
sudo apt install python3-numpy

## Example: Built-in Modules

|  |
| --- |
| import math    print (math.sqrt(121))    print (math.pi)    print (dir(math)) |

**Output**:

11.0  
3.141592653589793  
[‘doc’, ‘loader’, ‘name’, ‘package’, ‘spec’, ‘acos’, ‘acosh’, ‘asin’, ‘asinh’, ‘atan’, ‘atan2’, ‘atanh’, ‘ceil’, ‘copysign  
‘, ‘cos’, ‘cosh’, ‘degrees’, ‘e’, ‘erf’, ‘erfc’, ‘exp’, ‘expm1’, ‘fabs’, ‘factorial’, ‘floor’, ‘fmod’, ‘frexp’, ‘fsum’, ‘gamma’, ‘hypot’, ‘isf  
inite’, ‘isinf’, ‘isnan’, ‘ldexp’, ‘lgamma’, ‘log’, ‘log10’, ‘log1p’, ‘log2’, ‘modf’, ‘pi’, ‘pow’, ‘radians’, ‘sin’, ‘sinh’, ‘sqrt’, ‘tan’, ‘t  
anh’, ‘trunc’]

In the above example, dir() method gives the function name, variables, etc in the math module.

## Variable in a Module

Apart from methods and classes, A module can also contain variables.

**Example**:

|  |
| --- |
| Fruit = {    "name": "Apple",    "color": "Green"  } |

Save the above snippet of code in the file Module1.py

|  |
| --- |
| import Module1    x = Module1.Fruit["name"]  print(x) |

**Output:**

Apple

In the above piece of code, Module1 is imported and functionality is performed on it.

## Difference between a module and a package in Python

**Python Module**: These are set of pre-defined files that contain the python codes which depict the basic functionalities of class, methods, variables, etc.

**Python Package**: It is a directory that holds and contains modules and sub-packages.