Programming Concept with Python

UNIT 01 : Fundamentals of Computer

UNIT 02 : Programming Basics

Problem analysis

Problem analysis in programming is the process of identifying and analyzing problems, and developing solutions to address them. It involves breaking down a problem into parts to understand it, and then using that understanding to come up with a solution.

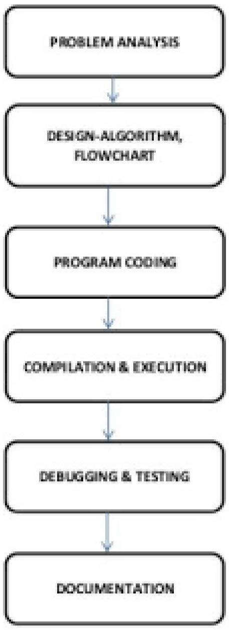
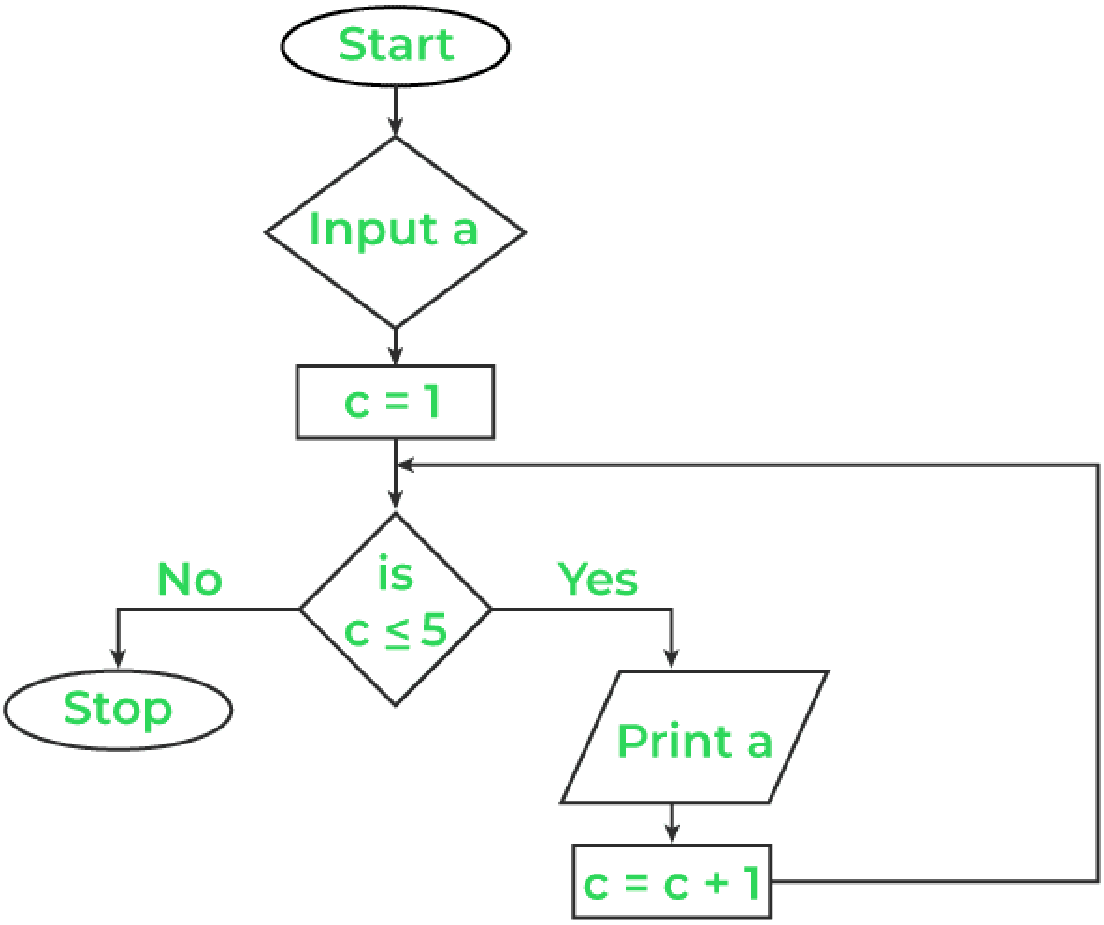


Figure: Problem solving process

Flowchart

A flowchart is a diagram that shows the steps, sequences, and decisions of a process or workflow. It's a versatile tool that can be used in many fields for planning, visualizing, documenting, and improving processes.



Algorithms

An algorithm is a set of instructions that are followed in order to complete a task or solve a problem. Algorithms are a key part of computer programming and are used to perform a variety of tasks, such as: calculations, finding information in databases, navigation, search engines, and music streaming services.

Pseudo codes

Pseudo code is a way of writing algorithms in a simple and understandable manner, using plain language and programming-like syntax. It is not tied to any specific programming language and is used for planning and explaining code logic.

```
FUNCTION IsPrime(n)
   IF n <= 1 THEN
      RETURN False
   ENDIF
   FOR i = 2 TO n-1 DO
      IF n MOD i == 0 THEN
         RETURN False
      ENDIF
   ENDFOR
   RETURN True
ENDFUNCTION
```

Structured Programming:

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of software. It emphasizes breaking a program into smaller, manageable, and logical units called **modules**.

Advantages of Structured Programming

1. **Readability:**
   - Code is easier to read and understand.
2. **Maintainability:**

    - Modular design allows updating or fixing parts of the program without affecting the whole system.
3. **Debugging:**
    - Errors are easier to identify and fix due to the structured approach.
4. **Reusability:**
    - Modules and functions can be reused across programs.
5. **Reduced Complexity:**
    - Breaking problems into smaller tasks simplifies the overall solution.

Example of Flowchart and Algorithm representation

Calculate the Sum of the First N Natural Numbers

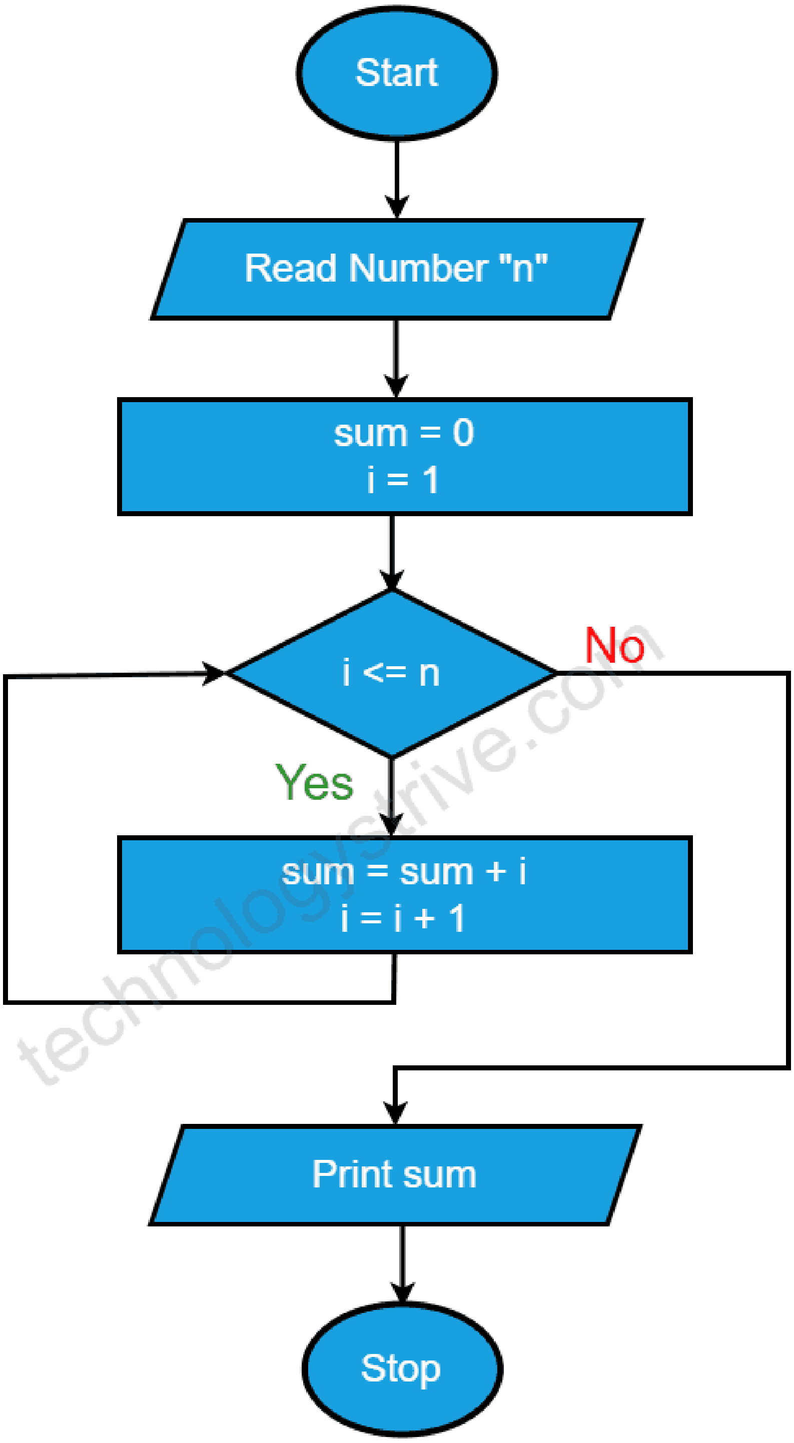
Algorithm

1. Start.
2. Input the value of NNN.
3. Initialize `sum = 0`.
4. Loop from `i = 1` to `N`:
    - Add iii to `sum`.
5. Print the value of `sum`.
6. Stop.

Flowchart

technologystrive.com

Flowchart - Sum of first n natural numbers



technologystrive.com
technologystrive.com
technologystrive.com
technologystrive.com

UNIT 03 : Variable and Expression

Variables

Definition: Variables are names that refer to values stored in memory.

Naming Rules: Must start with a letter or underscore (_), followed by letters, digits, or underscores.

Example:
x = 10
name = "Alice"

Expressions

Arithmetic Expressions: Combine numbers and operators to perform calculations.

Operators: +, -, *, /, // (floor division), % (modulus), ** (exponentiation)

Example:
result = (5 + 3) * 2

Logical Expressions: Combine boolean values and operators to perform logical operations.

Operators: and, or, not

Example:
is_valid = (age > 18) and (age < 65)

Evaluation of Expressions

- Associativity: Determines the order in which operators of the same precedence are processed.
  - Left-to-right: Most operators (e.g., `+`, `-`, `*`, `/`)
  - Right-to-left: Exponentiation (`**`)
- Precedence: Determines the order in which different operators are processed.

Example:
result = 5 + 3 * 2 # Multiplication before addition

Assignment Operation

- Syntax: `variable = expression`
- Left Hand Side (LHS): The variable name where the result will be stored.
- Right Hand Side (RHS): The expression to be evaluated and assigned to the variable.

Example:
x = 5 + 3 # RHS is evaluated first, then assigned to LHS

Console Input/Output

- Taking Input from User: Using the `input()` function.

Example:
name = input("Enter your name: ")

```
age = int(input("Enter your age: "))  # Convert input to integer
```

- Printing User Information: Using the `print()` function.

Example:

```
print("Name:", name)
```

- print("Age:", age)

<u>UNIT 04 : Control Statement and Iteration</u>

If Statement

Syntax:

```
if condition:
    # code block
```

Example:

```
if age > 18:
    print("You are an adult.")
```

Else-If Statement (Elif)

Syntax:

```
if condition1:
    # code block
elif condition2:
    # code block
else:
    # code block
```

Example:

```
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
else:
    grade = 'C'
```

Multiple Statements within If

Example:

```
if age > 18:
    print("You are an adult.")
    print("You can vote.")
```

Multiple If Statements

Example:

```
if age > 18:
    print("You are an adult.")
if age >= 65:
    print("You are a senior citizen.")
```

Loops

While Loop

Syntax:

```
while condition:
    # code block
```

Example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

For Loop

Syntax:

```
for variable in iterable:
    # code block
```

Example:

```
for i in range(5):
    print(i)
```

Nesting Loops

Example:

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

Controlling Loops

Break Statement

Usage: Exits the loop prematurely.

Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Continue Statement

Usage: Skips the current iteration and continues with the next.

Example:

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Else Statement in Loops

Usage: Executes after the loop completes normally (not via `break`).

Example:

```
for i in range(5):
    print(i)
else:
    print("Loop completed.")
```

-

Range Statement

- Usage: Generates a sequence of numbers.
- Syntax: `range(start, stop, step)`

Example:

```
for i in range(1, 10, 2):
    print(i)
```

-

Pass Statement

- Usage: Acts as a placeholder; does nothing.

Example:

```
for i in range(5):
    if i == 3:
        pass

        print(i)
```

UNIT 05 : Collections

Strings

- Definition: A sequence of characters.

Creation:
text = "Hello, World!"

- Common Methods:
  - `len(text)`: Returns the length of the string.
  - `text.upper()`: Converts to uppercase.
  - `text.lower()`: Converts to lowercase.
  - `text.split()`: Splits the string into a list of words.
  - `text.replace("World", "Python")`: Replaces a substring.

Lists

- Definition: An ordered collection of items.

Creation:
numbers = [1, 2, 3, 4, 5]

-
- Common Methods:
  - `len(numbers)`: Returns the length of the list.
  - `numbers.append(6)`: Adds an item to the end.

    - `numbers.remove(3)`: Removes the first occurrence of an item.
    - `numbers.sort()`: Sorts the list.
    - `numbers.reverse()`: Reverses the list.

Tuples

- Definition: An ordered, immutable collection of items.

Creation:
coordinates = (10, 20)

-
- Common Methods:
    - `len(coordinates)`: Returns the length of the tuple.
    - `coordinates.count(10)`: Counts occurrences of an item.
    - `coordinates.index(20)`: Returns the index of an item.

Dictionary

- Definition: A collection of key-value pairs.

Creation:
student = {"name": "Alice", "age": 21, "grade": "A"}

-
- Common Methods:

    - `len(student)`: Returns the number of key-value pairs.
    - `student.keys()`: Returns a list of keys.
    - `student.values()`: Returns a list of values.
    - `student.items()`: Returns a list of key-value pairs.
    - `student.get("name")`: Returns the value for a key.

Set

- Definition: An unordered collection of unique items.

Creation:
fruits = {"apple", "banana", "cherry"}

-
- Common Methods:
    - `len(fruits)`: Returns the number of items.
    - `fruits.add("orange")`: Adds an item.
    - `fruits.remove("banana")`: Removes an item.
    - `fruits.union({"grape", "melon"})`: Returns the union of sets.

        - `fruits.intersection({"apple", "melon"})`: Returns the intersection of sets.

Sorting Algorithms

Selection Sort

- Definition: A simple comparison-based sorting algorithm.
- Steps:
    1. Find the minimum element in the unsorted part of the list.
    2. Swap it with the first unsorted element.
    3. Move the boundary of the sorted part one element to the right.

Example:

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

-

Bubble Sort

- Definition: A simple comparison-based sorting algorithm.
- Steps:
    1. Repeatedly step through the list.
    2. Compare adjacent elements and swap them if they are in the wrong order.
    3. Continue until the list is sorted.

Example:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

                return arr
```

UNIT 06 : Functions

Built-in Functions

- Definition: Functions that are pre-defined in Python and can be used directly.
- Examples:
  - `print()`: Outputs data to the console.
  - `len()`: Returns the length of an object.
  - `type()`: Returns the type of an object.
  - `sum()`: Returns the sum of elements in an iterable.
  - `max()`, `min()`: Returns the maximum and minimum values.

User-Defined Functions

- Definition: Functions created by the user to perform specific tasks.

Syntax:

```
def function_name(parameters):
    # code block
    return value
```

-

Example:

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

Function Passing Values

- Definition: Functions can accept values (arguments) when they are called.

Example:

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

result = add(5, 3)  # Passing 5 and 3 as arguments
```

-

Function Returning Values

- Definition: Functions can return a value using the `return` statement.

Example:

```
def multiply(a, b):
    return a * b

product = multiply(4, 5)  # Function returns 20
```

-

Default Parameter Values

- Definition: Parameters can have default values, which are used if no argument is provided.

Syntax:

```
def function_name(parameter=default_value):
    # code block
```

-

Example:

```
def greet(name="Guest"):
    return f"Hello, {name}!"

print(greet())  # Uses default value "Guest"
print(greet("Alice"))  # Uses provided argument "Alice"
```

Recursive Functions

- Definition: Functions that call themselves to solve a problem.

Example: Calculating the factorial of a number.

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)


        print(factorial(5))  # Output: 120
```

UNIT 07 : File Management

Operations on Files

Opening Files

Syntax:
file = open("filename", mode, encoding)

-
- Modes:
    - `'r'`: Read (default mode)
    - `'w'`: Write (creates a new file or truncates an existing file)
    - `'a'`: Append (adds to the end of the file)
    - `'b'`: Binary mode (e.g., `'rb'`, `'wb'`)
    - `'+'`: Read and write (e.g., `'r+'`, `'w+'`)

Example:
file = open("example.txt", "r")

-

Attributes

- Common Attributes:
    - `file.name`: Name of the file.
    - `file.mode`: Mode in which the file was opened.
    - `file.closed`: Boolean indicating if the file is closed.

Encoding

- Definition: Specifies the encoding used to decode or encode the file.

Example:

```
file = open("example.txt", "r", encoding="utf-8")
```

Closing Files

Syntax:

```
file.close()
```

Example:

```
file = open("example.txt", "r")
# Perform file operations
file.close()
```

Read and Write Methods

read() Method

- Definition: Reads the entire file or a specified number of characters.

Syntax:

```
content = file.read(size)
```

Example:

```
file = open("example.txt", "r")
content = file.read()
file.close()
```

write() Method

- Definition: Writes a string to the file.

Syntax:

```
file.write(string)
```

Example:

```
file = open("example.txt", "w")
file.write("Hello, World!")
file.close()
```

tell() and seek() Methods

tell() Method

- Definition: Returns the current position of the file pointer.

Syntax:

```
position = file.tell()
```

Example:

```
file = open("example.txt", "r")
position = file.tell()
file.close()
```

seek() Method

- Definition: Moves the file pointer to a specified position.

Syntax:

```
file.seek(offset, whence)
```

- Parameters:
  - `offset`: Number of bytes to move.
  - `whence`: Reference point (0: beginning, 1: current position, 2: end).

Example:

```
file = open("example.txt", "r")
file.seek(10)
file.close()
```

Renaming and Deleting Files and Directories

Renaming Files

Syntax:

```
import os
```

```
os.rename("old_name.txt", "new_name.txt")
```

Example:

```
import os
os.rename("example.txt", "new_example.txt")
```

Deleting Files

Syntax:

```
import os
os.remove("filename")
```

Example:

```
import os
os.remove("example.txt")
```

Deleting Directories

Syntax:

```
import os
os.rmdir("directory_name")
```

Example:

```
import os
```

- os.rmdir("example_directory")

UNIT 08 : Errors and Exception Handling

Dealing with Syntax Errors

- Definition: Errors in the code that violate the syntax rules of the programming language.
- Common Causes: Missing colons, parentheses, indentation errors, etc.

Example:

```
if True
    print("This will cause a syntax error")
```

Fix: Correct the syntax.

```
if True:
    print("This is correct")
```

Exceptions

- Definition: Errors that occur during the execution of a program.
- Common Types:
    - `ZeroDivisionError`: Division by zero.
    - `IndexError`: Index out of range.
    - `KeyError`: Key not found in a dictionary.
    - `TypeError`: Operation on incompatible types.
    - `ValueError`: Invalid value.

Example:

```
result = 10 / 0  # This will raise a ZeroDivisionError
```

Handling Exceptions with try/except

Syntax:

```
try:
    # code that may raise an exception
except ExceptionType:
    # code to handle the exception
```

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Cleaning Up with finally

- Definition: The `finally` block is executed no matter what, whether an exception is raised or not.

Syntax:

```
try:
    # code that may raise an exception
except ExceptionType:
    # code to handle the exception
```

```
finally:
    # code that will always execute
```

Example:

```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found")
finally:
    file.close()
```

- print("File closed")

UNIT 09 : Classes and Objects

Create a Class

- Definition: A blueprint for creating objects (a particular data structure).

Syntax:

```
class ClassName:
    # class attributes and methods
```

-

Example:

```
class Person:
    pass
```

-

Create Object

- Definition: An instance of a class.

Syntax:

```
object_name = ClassName()
```

-

Example:

```
person1 = Person()
```

-

`__init__()` Function

- Definition: A special method called when an object is instantiated.

Syntax:

```
class ClassName:
    def __init__(self, parameters):
        # initialize attributes
```

-

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person1 = Person("Alice", 30)
```

-

Methods

- Definition: Functions defined within a class.

Syntax:

```
class ClassName:
    def method_name(self, parameters):
        # method body
```

-

Example:

```
class Person:
    def greet(self):
        return f"Hello, my name is {self.name}"
```

```
person1 = Person("Alice", 30)
print(person1.greet())
```

-

Self Parameter

- Definition: Refers to the instance of the class.
- Usage: Used to access attributes and methods of the class.

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

-

Modification and Deletion of Object Parameters

Modification:

```
person1.name = "Bob"
```

-

Deletion:

```
del person1.age
```

-

Deletion of Object

Syntax:

```
del object_name
```

-

Example:

```
del person1
```

-

Pass Statement

- Definition: A placeholder for future code.

Syntax:

```
class Person:
    pass
```

-

Inheritance and Polymorphism

Inheritance

- Definition: A class can inherit attributes and methods from another class.

Syntax:

```
class ChildClass(ParentClass):
    # additional attributes and methods
```

-

Example:

```
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id
```

-

Polymorphism

- Definition: The ability to use a common interface for multiple forms (data types).

Example:

```
class Dog:
    def sound(self):
        return "Bark"

class Cat:
    def sound(self):
        return "Meow"

def make_sound(animal):
    print(animal.sound())

dog = Dog()
cat = Cat()
make_sound(dog)
make_sound(cat)
```

-

Scope

- Definition: The region of the code where a variable is accessible.
- Types:
    - Local Scope: Variables defined within a function.

    - Global Scope: Variables defined outside any function.

Example:

```
x = "global"

def my_function():
    x = "local"
    print(x)

my_function()
print(x)
```

-

Modules

- Definition: A file containing Python code (functions, classes, variables).

Importing Modules:

```
import module_name
```

-

Example:

```
import math
print(math.sqrt(16))
```

-

Built-In Math Functions

- Examples:

    - `abs(x)`: Returns the absolute value.
    - `pow(x, y)`: Returns (x^y).
    - `round(x)`: Rounds a number.

Math Module

- Common Functions:
    - `math.sqrt(x)`: Returns the square root.
    - `math.factorial(x)`: Returns the factorial.
    - `math.sin(x)`, `math.cos(x)`, `math.tan(x)`: Trigonometric functions.

Module datetime and Date Objects

Importing:
import datetime

-

Creating Date Objects:
today = datetime.date.today()
print(today)

-

RegEx Module and RegEx Functions

Importing:
import re

-
- Common Functions:

- `re.search(pattern, string)`: Searches for a pattern.
- `re.match(pattern, string)`: Matches a pattern at the beginning.
- `re.findall(pattern, string)`: Finds all occurrences.

Exception Handling

- Definition: Managing errors using `try`, `except`, and `finally`.

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
```

- print("Execution completed")

UNIT 10 : Modules& Packages

Importing a Module

- Definition: Bringing in a module to use its functions, classes, and variables.

Syntax:

```
import module_name
```

-

Example:

```
import math
print(math.sqrt(16))
```

-

Import Specific Functions or Variables:

```
from module_name import function_name
```

-

Example:

```
from math import sqrt
print(sqrt(16))
```

-

Creating a Module

- Definition: A file containing Python code (functions, classes, variables) that can be imported.
- Steps:

1. Create a Python file (e.g., `mymodule.py`).
2. Define functions, classes, or variables in this file.
3. Import the module in another script.

Example:

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"
# main.py
import mymodule
print(mymodule.greet("Alice"))
```

-

Function Aliases

- Definition: Assigning an alias to a function or module for easier use.

Syntax:

```
import module_name as alias
```

Example:

```
import numpy as np
array = np.array([1, 2, 3])
```

Alias for Functions:

```
from module_name import function_name as alias
```

Example:

```
from math import sqrt as square_root
print(square_root(16))
```

Packages

- Definition: A way of organizing related modules into a directory hierarchy.
- Structure:
  - A package is a directory containing a special `__init__.py` file and one or more module files.

Example:

```
mypackage/
    __init__.py
    module1.py
    module2.py
```

-

Importing from a Package:

```
from mypackage import module1
```

Example:

```
# mypackage/module1.py
def func1():
    return "Function 1 from module 1"
# main.py
from mypackage import module1
```

- print(module1.func1())