Python Programming

Comprehensive Notes on Python

By Virendra Goura

NOTE

Although every effort has been made to avoid errors and omissions, there is still a possibility that some mistakes may be missed due to invisibility.

This E - book is issued with the Understanding that the author is not responsible in any way for any errors/omissions.

BCA 302: Python Programming

Question Paper pattern for Main University Examination **Max Marks: 100**

Part-1 (very short answer) consists 10 questions of two marks each with two questions from each unit. Maximum limit for each question is up to 40 words.

Part-II (short answer) consists 5 questions of four marks each with one question from each unit. Maximum limit for each question is up to 80 words.

Part-III (Long answer) consists 5 questions of twelve marks each with one question from each unit with internal choice.

UNIT-I

Python Concepts: Origin, Comparison, Comments, Variables and Assignments, Identifiers, basic style Guidelines, Standard Types, Internal types, Operators, Built-in Functions, Numbers and strings, Sequences: strings, sequences, String operators & functions, Special features of strings, Memory management, programs & examples

Conditionals and loops: If statement, else statement, elif statement, while statement, for statement, break statement, Continue Statement, pass statement, else statement

UNIT- II

Object and classes: classes in Python, Principles of object orientation, Creating classes, Instance methods, Class variables, Inheritance, Polymorphism, type Identification, Python libraries (Strings, Data structure & Algorithms).

Lists and sets: Built-in Functions, List type built in Methods, Tuples, tuple operators, Special features of tuples, **Set:** Introduction, Accessing, built- in Methods (Add, update, Clear, copy, discard, remove), Operations (union, Intersection, Difference).

UNIT-III

Dictionaries: introduction to Dictionaries, Built-in Functions, Built-in Methods, Dictionaries keys, Sorting and Looping, Nested Dictionaries.

Files: File Objects, File Built-in Function, File Built-in methods, File Built- in Attributes. Standard files, Command-line Arguments, File system, File Execution, Persistent Storage Modules.

UNIT-IV

Regular Expression: Regular expression: Introduction/Motivation, Special Symbols and Characters for REs, REs and python.

Exceptions: Concepts of Exceptions, Exceptions in python, detecting and handling exceptions, exceptions as strings, raising exceptions, assertions, standard exceptions.

UNIT-V

Database Interaction: SQL Database Connection using python, Creating and Searching tables, reading and storing config information on database, programming using database connections,

Python Multithreading: understanding threads, Forking threads, synchronizing the threads, Programming using multithreading.

UNIT-I: Python Concepts

Python Concepts: Origin, Comparison, Comments, Variables and Assignments, Identifiers, basic style Guidelines, Standard Types, Internal types, Operators, Built-in Functions, Numbers and strings, Sequences: strings, sequences, String operators & functions, Special features of strings, Memory management, programs & examples

Conditionals and loops: If statement, else statement, elif statement, while statement, for statement, break statement, Continue Statement, pass statement, else statement.

---

1. Python Basics

Python is a high-level, interpreted programming language designed by **Guido van Rossum** and first released in **1991**. It emphasizes code readability, simplicity, and flexibility, making it suitable for beginners and experienced developers alike.

Origin of Python:

Python was created as a successor to the ABC programming language. Its goal was to provide a language that was simple to understand and use but also powerful enough to support complex systems. It was influenced by several languages, including C, C++, and Unix shell scripting.

Comparison with Other Languages:

Python is often compared with other programming languages like **Java**, **C++**, and **JavaScript**. Some key differences include:

- **Syntax**: Python's syntax is simpler and more readable.
- **Typing**: Python is dynamically typed, while Java and C++ are statically typed.
- **Memory management**: Python handles memory management automatically through garbage collection, unlike C/C++ where the programmer has to manage memory manually.

---

2. Key Concepts in Python

Comments in Python:

- **Single-line comments** start with a # symbol.

```
# This is a comment
print("Hello, World!")
```

- **Multi-line comments** are enclosed in triple quotes (''' or """).

```
'''
This is a multi-line comment.
It spans multiple lines.
'''
```

Variables and Assignments:

In Python, you don't need to declare the type of variable beforehand. It is dynamically typed.

```
x = 10  # Integer

name = "John"  # String
```

Identifiers:

- Identifiers are names used to identify variables, functions, classes, etc.
- Python identifiers must start with a letter or an underscore (_), followed by letters, digits, or underscores.
- They are case-sensitive, meaning myvariable and MyVariable are different.

Basic Style Guidelines:

- **PEP 8** is the style guide for Python code.
- Use **4 spaces** for indentation.
- Function and variable names should be written in **snake_case** (e.g., my_function).
- Class names should be written in **CamelCase** (e.g., MyClass).

Standard Types:

- **int**: Integer (e.g., 10)
- **float**: Floating-point number (e.g., 3.14)
- **str**: String (e.g., "hello")
- **bool**: Boolean (e.g., True, False)
- **list**: List of values (e.g., [1, 2, 3])
- **tuple**: Immutable sequence (e.g., (1, 2, 3))
- **dict**: Dictionary (e.g., {"key": "value"})

Internal Types:

Python also has internal types for more complex data structures such as **sets**, **frozensets**, and **byte sequences**.

Operators:

Python supports various operators like:

- **Arithmetic Operators**: +, -, *, /, //, %, **
- **Comparison Operators**: ==, !=, <, >, <=, >=
- **Logical Operators**: and, or, not
- **Assignment Operators**: =, +=, -=, *=, /=
- **Bitwise Operators**: &, |, ^, ~, <<, >>

Built-in Functions:

Python includes many built-in functions such as:

- print() - to output data
- len() - to find the length of a sequence
- input() - to take input from the user

- int(), str(), float() - type conversion functions

Numbers and Strings:

- **Numbers**: Python supports integers (int) and floating-point numbers (float).

```
num = 10  # Integer
price = 19.99  # Float
```

- **Strings**: Python strings are sequences of characters enclosed in single (') or double (") quotes.

```
name = "John"
greeting = 'Hello, ' + name
```

---

3. Sequences in Python

Sequences are ordered collections of elements in Python. The common sequence types are **strings**, **lists**, **tuples**, and **ranges**.

- **Strings**: Strings are sequences of characters.

```
s = "Hello"
print(s[0])  # H
```

- **Lists**: Lists are mutable sequences, allowing you to modify their elements.

```
lst = [1, 2, 3]
lst[0] = 10  # Lists are mutable
print(lst)  # [10, 2, 3]
```

- **Tuples**: Tuples are immutable sequences.

```
tup = (1, 2, 3)
# tup[0] = 10  # This would raise an error because tuples are immutable
```

String Operators and Functions:

Strings in Python come with various operators and functions to manipulate them:

- **String Concatenation**: You can join strings using the + operator.

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name  # John Doe
```

- **String Methods**: Some useful methods include:
    - upper(): Converts to uppercase
    - lower(): Converts to lowercase
    - split(): Splits string into a list
    - replace(): Replaces substrings

```
text = "hello world"
```

```
print(text.upper())  # "HELLO WORLD"

print(text.split())  # ['hello', 'world']
```

Special Features of Strings:

- **Slicing**: You can extract parts of a string using slicing.

```
s = "Python"

print(s[1:4])  # "yth"
```

- **Escape Sequences**: Special characters within strings, such as newline (\n), tab (\t), and backslash (\\).

---

4. Memory Management in Python

Python automatically manages memory through a built-in garbage collector. It keeps track of objects in memory and reclaims unused objects when they are no longer needed.

Garbage Collection:

Python uses reference counting and cyclic garbage collection to reclaim memory from objects that are no longer in use.

---

5. Programs and Examples

Example 1: A Simple Calculator

```
# Simple Python Calculator


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


def subtract(a, b):
    return a - b


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


def divide(a, b):
    return a / b


print("Addition: ", add(10, 5))

print("Subtraction: ", subtract(10, 5))

print("Multiplication: ", multiply(10, 5))
```

```
print("Division: ", divide(10, 5))
```

Example 2: Check Even or Odd

```
# Check if a number is even or odd


num = int(input("Enter a number: "))


if num % 2 == 0:
    print(f"{num} is even.")
else:
    print(f"{num} is odd.")
```

---

6. Conditionals and Loops

If-Else Statements:

Python uses if, else, and elif for conditional statements:

```
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is less than or equal to 5")
```

Loops:

- **While Loop**: Continues to run as long as a condition is True.

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

- **For Loop**: Iterates over a sequence (like a list or range).

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

- **Break Statement**: Exits the loop.

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

- **Continue Statement**: Skips the rest of the current loop iteration.

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

- **Pass Statement**: A placeholder for future code.

```
for i in range(5):
    pass  # This does nothing
```

Else in Loops:

The else block can be used with loops and is executed when the loop completes without encountering a break statement.

```
for i in range(5):
    print(i)
else:
    print("Loop finished!")
```

Conclusion

In this unit, we learned about the fundamental concepts of **Python programming**, including its syntax, variables, operators, and built-in functions. We explored **sequences** like strings, lists, and tuples, and learned various operations and functions available for string manipulation. We also examined **memory management** and how Python handles memory automatically. Finally, we covered **conditionals and loops**, allowing us to control the flow of execution based on conditions and iterate through sequences. This foundation provides the necessary tools to write and understand Python programs effectively.

UNIT-II: Object and Classes in Python

Object and classes: classes in Python, Principles of object orientation, Creating classes, Instance methods, Class variables, Inheritance, Polymorphism, type Identification, Python libraries (Strings, Data structure & Algorithms).

Lists and sets: Built-in Functions, List type built in Methods, Tuples, tuple operators, Special features of tuples, **Set:** Introduction, Accessing, built- in Methods (Add, update, Clear, copy, discard, remove), Operations (union, Intersection, Difference).

---

1. Object and Classes in Python

Classes in Python:

A class in Python is a blueprint for creating objects (instances). It defines the properties and behaviors (methods) that the objects created from it will have. Python is an **object-oriented** language, which means everything in Python is treated as an object.

- **Creating a Class**:

```
class Dog:
    # Constructor to initialize attributes
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    # Method to display dog details
    def bark(self):
        print(f"{self.name} says Woof!")
```

- **Creating an Object**: After defining a class, you can create an instance (object) of the class.

```
dog1 = Dog("Buddy", "Golden Retriever")
dog1.bark()  # Output: Buddy says Woof!
```

Principles of Object-Oriented Programming (OOP):

1. **Encapsulation**: Bundling the data (attributes) and methods that operate on the data into a single unit, i.e., a class. It also restricts access to certain details of an object to safeguard data.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary  # Private attribute
```

```
    def get_salary(self):
        return self.__salary
```

2. **Abstraction**: Hiding the complexity of the implementation from the user and only exposing essential features.
3. **Inheritance**: A way for one class (child class) to inherit attributes and methods from another class (parent class).
4. **Polymorphism**: The ability to take many forms, i.e., having methods in different classes with the same name but different implementations.

```
class Animal:
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        print("Woof!")

class Cat(Animal):
    def sound(self):
        print("Meow!")

dog = Dog()
cat = Cat()

dog.sound()  # Woof!
cat.sound()  # Meow!
```

Creating Classes in Python:

You can create a class in Python using the class keyword followed by the class name.

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

Instance Methods:

Instance methods are functions defined inside a class that operate on instances of the class.

```
class Car:
    def __init__(self, model, year):
        self.model = model
        self.year = year

    def display_info(self):
        print(f"Car Model: {self.model}, Year: {self.year}")
```

Class Variables:

Class variables are shared by all instances of a class. They are defined inside the class but outside of any instance methods.

```
class Counter:
    count = 0  # Class variable

    def __init__(self):
        Counter.count += 1
```

Inheritance:

Inheritance allows one class to inherit the properties and methods of another class.

```
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):
    def sound(self):
        print("Woof!")
```

Polymorphism:

Polymorphism allows methods to have the same name but different behaviors based on the object type.

```
class Bird:
    def fly(self):
        print("Birds fly in the sky")

class Airplane:
    def fly(self):
```

```
        print("Airplanes fly at high altitudes")


# Demonstrating polymorphism
def make_it_fly(flyable):
    flyable.fly()


bird = Bird()
plane = Airplane()


make_it_fly(bird)    # Output: Birds fly in the sky
make_it_fly(plane)   # Output: Airplanes fly at high altitudes
```

Type Identification:

To check the type of an object, you can use the type() function.

```
x = 10
print(type(x))  # Output: <class 'int'>


dog = Dog("Buddy", "Golden Retriever")
print(type(dog))  # Output: <class '__main__.Dog'>
```

Python Libraries:

- **Strings**: Python provides a vast collection of string manipulation methods such as upper(), lower(), replace(), etc.
- **Data Structures & Algorithms**: Python has built-in libraries like collections and heapq for efficient data structure manipulation and algorithmic operations.

---

2. Lists and Sets in Python

Lists:

A **list** is an ordered collection of elements that can hold elements of any data type (int, string, etc.). Lists are mutable, meaning you can modify their contents after creation.

```
my_list = [1, 2, 3, 4, 5]
my_list.append(6)  # Adds an element to the end of the list
```

Built-in Functions for Lists:

- len(): Returns the length of the list.
- sorted(): Returns a sorted version of the list.
- min() and max(): Return the minimum and maximum values in the list, respectively.

List Methods:

- append(): Adds an element at the end.
- insert(): Adds an element at a specific index.
- remove(): Removes the first occurrence of an element.
- pop(): Removes and returns the last element or the element at a specific index.

```
my_list.append(7)

my_list.remove(4)

my_list.pop()
```

Tuples:

A **tuple** is an ordered, immutable collection of elements. Once defined, the elements in a tuple cannot be changed.

```
my_tuple = (1, 2, 3, 4, 5)
```

Tuple Operators:

- **Concatenation**: You can concatenate two tuples using +.

```
tuple1 = (1, 2)

tuple2 = (3, 4)

result = tuple1 + tuple2  # (1, 2, 3, 4)
```

- **Repetition**: You can repeat a tuple using *.

```
repeated_tuple = (1, 2) * 3  # (1, 2, 1, 2, 1, 2)
```

Special Features of Tuples:

- **Immutability**: Tuples cannot be modified after creation.
- **Indexing** and **Slicing**: You can access tuple elements by index and slice them.

Sets:

A **set** is an unordered collection of unique elements. Sets are used for membership tests, removing duplicates from sequences, and performing mathematical set operations.

Introduction to Sets:

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

Accessing Sets:

You cannot access set elements by index, as sets are unordered.

```
for elem in my_set:
    print(elem)
```

Set Methods:

- add(): Adds an element to the set.
- update(): Adds multiple elements to the set.
- clear(): Removes all elements from the set.

- copy(): Returns a shallow copy of the set.
- discard(): Removes an element if present, but does not raise an error if the element is not found.
- remove(): Removes an element, and raises a KeyError if the element is not present.

```
my_set.add(6)          # Adds 6 to the set
my_set.update([7, 8])  # Adds 7 and 8 to the set
my_set.remove(3)       # Removes 3 from the set
my_set.discard(10)     # Does nothing if 10 is not present
```

Set Operations:

- **Union**: Combines elements of two sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2  # {1, 2, 3, 4, 5}
```

- **Intersection**: Returns common elements of two sets.

```
intersection_set = set1 & set2  # {3}
```

- **Difference**: Returns elements in the first set but not in the second.

```
difference_set = set1 - set2  # {1, 2}
```

Conclusion

In this unit, we explored **object-oriented programming** in Python, including the creation and manipulation of **classes** and **objects**, principles like **inheritance** and **polymorphism**, and how Python handles **type identification**. We also learned about various **Python libraries** such as those for handling **strings** and **data structures**. Additionally, we delved into **lists**, **tuples**, and **sets**, understanding their built-in functions, methods, and operations that make them powerful tools for managing collections of data. These concepts form the foundation for writing more advanced and efficient Python programs.

UNIT-III: Dictionaries and Files in Python

Dictionaries: introduction to Dictionaries, Built-in Functions, Built-in Methods, Dictionaries keys, Sorting and Looping, Nested Dictionaries.

Files: File Objects, File Built-in Function, File Built-in methods, File Built- in Attributes. Standard files, Command-line Arguments, File system, File Execution, Persistent Storage Modules.

---

1. Dictionaries in Python

A **dictionary** in Python is an unordered collection of items. Each item is stored as a key-value pair, where the key must be unique, and the value can be any data type. Dictionaries are mutable, meaning you can change their contents.

Introduction to Dictionaries:

Dictionaries are defined using curly braces {}, with keys and values separated by a colon :. Each key-value pair is separated by a comma.

```
my_dict = {
    "name": "John",
    "age": 25,
    "city": "New York"
}
```

Built-in Functions:

Python provides several built-in functions for dictionaries:

- len(): Returns the number of key-value pairs in the dictionary.
- del: Removes a key-value pair.
- sorted(): Returns a sorted list of the dictionary keys.

```
print(len(my_dict))  # Output: 3
del my_dict["age"]   # Removes the key 'age'
print(my_dict)       # Output: {'name': 'John', 'city': 'New York'}
```

Built-in Methods:

Python dictionaries have several methods that can be used to manipulate them:

- keys(): Returns a view object that displays all the keys.
- values(): Returns a view object that displays all the values.
- items(): Returns a view object that displays all key-value pairs.
- get(): Returns the value associated with the specified key.
- pop(): Removes a key and returns its value.

```
# Using the get() method
print(my_dict.get("name"))  # Output: John
```

```
# Using the pop() method
removed_value = my_dict.pop("city")
print(removed_value)        # Output: New York
```

Dictionaries Keys:

Keys in dictionaries must be immutable types (e.g., strings, numbers, or tuples), but the values can be any data type.

```
my_dict = {
    (1, 2): "tuple as key",
    "age": 30
}
```

Sorting and Looping:

Dictionaries are unordered by default, but you can sort their keys or values using the sorted() function.

```
# Sorting by keys
sorted_keys = sorted(my_dict.keys())
print(sorted_keys)  # Output: ['age', (1, 2)]


# Looping through a dictionary
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

Nested Dictionaries:

A dictionary can also contain other dictionaries as values, allowing for nested structures.

```
nested_dict = {
    "person1": {"name": "Alice", "age": 30},
    "person2": {"name": "Bob", "age": 25}
}


print(nested_dict["person1"]["name"])  # Output: Alice
```

---

2. Files in Python

Files in Python allow you to store and retrieve data from your computer's file system. Python provides a simple interface for file handling.

File Objects:

To work with files in Python, you first need to open a file using the open() function. This function returns a file object, which you can use to read from or write to the file.

```
# Opening a file in read mode

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

File Built-in Functions:

- open(): Opens a file.
- close(): Closes the file object after use.
- read(): Reads the entire content of the file.
- readline(): Reads one line at a time.
- write(): Writes content to the file.
- writelines(): Writes multiple lines to the file.

```
file = open("example.txt", "w")  # Opening a file in write mode

file.write("Hello, World!")

file.close()
```

File Built-in Methods:

- flush(): Flushes the internal buffer, ensuring the written content is written to disk.
- seek(): Moves the file pointer to a specific location.
- tell(): Returns the current position of the file pointer.

```
# Example of seek() and tell()

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

print(file.tell())  # Output: 0 (initial position)

file.seek(5)

print(file.tell())  # Output: 5 (position after seek)

file.close()
```

File Built-in Attributes:

Files in Python have some important attributes, such as:

- name: Returns the name of the file.
- mode: Returns the access mode of the file (e.g., 'r' for read, 'w' for write).
- closed: Returns True if the file is closed, False otherwise.

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

print(file.name)  # Output: example.txt

print(file.mode)  # Output: r

file.close()
```

Standard Files:

In Python, three standard files are available by default:

- stdin: Standard input (keyboard).
- stdout: Standard output (screen).
- stderr: Standard error (screen).

Example: Writing to standard output.

```
import sys

sys.stdout.write("This is standard output\n")
```

Command-line Arguments:

Python allows you to pass arguments to a program through the command line. These arguments can be accessed using the sys.argv list, which contains the command-line arguments passed to the script.

```
import sys

print(sys.argv)  # Output: List of command-line arguments
```

File System:

Python provides the os module to interact with the operating system and manipulate files and directories.

```
import os


# Checking if a file exists

print(os.path.exists("example.txt"))  # Output: True or False


# Creating a directory

os.mkdir("new_directory")
```

File Execution:

You can execute files in Python using the subprocess module, which allows you to run system commands.

```
import subprocess

subprocess.run(["python", "script.py"])  # Executes script.py
```

Persistent Storage Modules:

Python offers libraries for persistent storage, such as pickle and json, which allow you to store and retrieve Python objects or data in files.

- **Using json for persistence:**

```
import json


data = {"name": "Alice", "age": 30}

with open("data.json", "w") as json_file:

    json.dump(data, json_file)
```

```
# Loading data from the file
with open("data.json", "r") as json_file:
    loaded_data = json.load(json_file)
    print(loaded_data)  # Output: {'name': 'Alice', 'age': 30}
```

Conclusion

In this unit, we explored **dictionaries**, an essential data structure in Python that allows storing key-value pairs. We discussed how to use dictionaries, their built-in functions and methods, and how to perform operations such as sorting, looping, and working with nested dictionaries. We also covered **file handling** in Python, which includes working with files, using built-in file functions and methods, and handling persistent storage through modules like json. These concepts are crucial for managing and processing data in Python programs.

UNIT-IV: Regular Expression and Exceptions in Python

Regular Expression: Regular expression: Introduction/Motivation, Special Symbols and Characters for REs, REs and python.

Exceptions: Concepts of Exceptions, Exceptions in python, detecting and handling exceptions, exceptions as strings, raising exceptions, assertions, standard exceptions.

---

1. Regular Expression in Python

A **Regular Expression (RE)** is a sequence of characters that defines a search pattern. It is widely used for string searching, pattern matching, and text processing. Regular expressions allow you to perform complex search and match operations on strings in a concise manner.

Introduction/Motivation:

Regular expressions provide a flexible and efficient way to search for patterns in strings, which can be difficult to achieve with simple string methods. They are commonly used for:

- Validating input (e.g., email, phone number).
- Searching for specific patterns (e.g., dates, URLs).
- Text parsing and transformation.

Special Symbols and Characters for REs:

Here are some of the key symbols and characters used in regular expressions:

- . (Dot): Matches any character except a newline.
- **^**: Matches the start of the string.
- **$**: Matches the end of the string.
- **[]**: Matches any character inside the square brackets.
- **|**: Acts as a logical OR between two patterns.
- *****: Matches 0 or more repetitions of the preceding pattern.
- **+**: Matches 1 or more repetitions of the preceding pattern.
- **?**: Matches 0 or 1 repetition of the preceding pattern.
- **\d**: Matches any digit (0-9).
- **\w**: Matches any alphanumeric character (letters and numbers).
- **\s**: Matches any whitespace character (space, tab, newline).
- **{n,m}**: Matches between n and m repetitions of the preceding pattern.

Examples of Regular Expressions:

1. **Matching an Email Address:**

```
import re

email_pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"

email = "test@example.com"
```

```
if re.match(email_pattern, email):
    print("Valid email address")
else:
    print("Invalid email address")
```

2. **Matching a Phone Number (simple version):**

```
phone_pattern = r"^\d{3}-\d{3}-\d{4}$"
phone_number = "123-456-7890"
if re.match(phone_pattern, phone_number):
    print("Valid phone number")
else:
    print("Invalid phone number")
```

3. **Extracting all digits from a string:**

```
text = "My phone number is 123-456-7890"
digits = re.findall(r"\d+", text)
print(digits)  # Output: ['123', '456', '7890']
```

Regular Expressions in Python:

In Python, the re module provides support for working with regular expressions. Some key functions in the re module are:

- **re.match()**: Checks for a match only at the beginning of the string.
- **re.search()**: Searches the entire string for a match.
- **re.findall()**: Returns all non-overlapping matches in a string as a list.
- **re.sub()**: Replaces occurrences of a pattern with a specified string.

```
import re

# Search for a pattern in a string
text = "Python is great"
pattern = r"great"
result = re.search(pattern, text)
if result:
    print("Pattern found!")
else:
    print("Pattern not found.")
```

2. Exceptions in Python

Exceptions are events that can modify the flow of control of a program. They are typically errors that occur during program execution (e.g., dividing by zero, accessing an index that is out of range, etc.). In Python, exceptions are handled using try, except, and other keywords to ensure that a program can continue running smoothly even in the presence of errors.

Concepts of Exceptions:

An exception is a problem that arises during the execution of a program, and it disrupts the normal flow of instructions. When an error occurs, Python stops executing the program and jumps to the nearest except block, where the error can be handled.

Exceptions in Python:

Python provides several built-in exceptions, and you can also create custom exceptions.

- **SyntaxError**: Raised when the Python interpreter encounters incorrect syntax.
- **ZeroDivisionError**: Raised when dividing a number by zero.
- **IndexError**: Raised when trying to access an index that is out of range.
- **ValueError**: Raised when an operation or function receives an argument with the right type but an inappropriate value.
- **FileNotFoundError**: Raised when trying to open a file that does not exist.

Detecting and Handling Exceptions:

Python handles exceptions using a try and except block. If an error occurs in the try block, the control is transferred to the except block where the exception is handled.

```
try:
    x = 5 / 0  # Division by zero
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

Exceptions as Strings:

Sometimes, exceptions can be handled as string representations. You can convert an exception to a string and access its details.

```
try:
    x = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError as e:
    print(f"Exception: {str(e)}")  # Output: Exception: division by zero
```

Raising Exceptions:

You can raise exceptions deliberately using the raise keyword. This is useful when you want to enforce certain conditions or when you encounter an error that needs to be raised explicitly.

```
def check_age(age):
    if age < 18:
```

```
        raise ValueError("Age must be 18 or older.")
    else:
        print("Access granted.")

try:
    check_age(15)  # This will raise an exception
except ValueError as e:
    print(e)  # Output: Age must be 18 or older.
```

Assertions:

An **assertion** is a statement used to check whether a condition is True during debugging. If the condition is False, an AssertionError is raised.

```
x = 10
assert x > 0, "x must be greater than 0"
# This will pass without any error.
assert x < 0, "x must be less than 0"
# This will raise an AssertionError with the message "x must be less than 0".
```

Standard Exceptions:

Python comes with a set of standard exceptions, some of which are:

- TypeError: Raised when an operation or function is applied to an object of an inappropriate type.
- KeyError: Raised when a dictionary key is not found.
- NameError: Raised when a local or global name is not found.

You can handle these exceptions using try and except blocks.

```
my_dict = {"name": "Alice"}
try:
    value = my_dict["age"]
except KeyError as e:
    print(f"KeyError: {e}")  # Output: KeyError: 'age'
```

Conclusion

In this unit, we explored **Regular Expressions** in Python, which provide powerful tools for pattern matching, text search, and text manipulation. Regular expressions are incredibly useful in various fields like text processing, validation, and parsing.

We also covered **exceptions** in Python, discussing the concept, types of exceptions, and how to detect, handle, and raise exceptions using try, except, and raise. Exception handling allows programs to continue executing smoothly even in the presence of errors. These concepts are crucial for building robust and error-resistant Python programs.

UNIT-V: Database Interaction and Python Multithreading

Database Interaction: SQL Database Connection using python, Creating and Searching tables, reading and storing config information on database, programming using database connections,
Python Multithreading: understanding threads, Forking threads, synchronizing the threads, Programming using multithreading.

---

1. Database Interaction in Python

Python provides several libraries for interacting with databases, such as sqlite3, MySQLdb, and psycopg2. Here, we will focus on SQL database connection using sqlite3 (a built-in module in Python) for interacting with a database.

SQL Database Connection Using Python:

To connect to a database, Python uses a **Database API (DB-API)** which provides a standard interface for interacting with various databases. For SQLite (a file-based database), you can use the built-in sqlite3 module.

1. **Connecting to a Database:**

```
import sqlite3


# Connect to a database (or create it if it doesn't exist)
connection = sqlite3.connect('example.db')
cursor = connection.cursor()  # Create a cursor object to interact with the database
```

2. **Creating a Table:**

You can create a table in an SQL database using the CREATE TABLE SQL command. Here's how to create a table in SQLite:

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER
)
''')
connection.commit()  # Commit the changes to the database
```

3. **Inserting Data into a Table:**

To insert data into the table, you can use the INSERT INTO SQL command. You can use parameterized queries to prevent SQL injection.

```
cursor.execute('''
INSERT INTO users (name, age)
VALUES (?, ?)
```

''', ('Alice', 30))

connection.commit()

4. **Searching Data in a Table:**

To search and retrieve data, you can use the SELECT SQL command. This allows you to query the database and fetch results.

cursor.execute('SELECT * FROM users')

rows = cursor.fetchall() # Fetch all rows from the result set

for row in rows:

  print(row) # Prints each row in the table

5. **Reading and Storing Configuration Information on the Database:**

Sometimes you need to store configuration settings in a database. You can create a configuration table to store these settings.

cursor.execute('''

CREATE TABLE IF NOT EXISTS config (

  key TEXT PRIMARY KEY,

  value TEXT

)

''')

Storing configuration data

cursor.execute('''

INSERT INTO config (key, value)

VALUES (?, ?)

''', ('site_name', 'My Website'))

connection.commit()

Retrieving configuration data

cursor.execute('SELECT * FROM config WHERE key="site_name"')

config = cursor.fetchone()

print(config) # Output: ('site_name', 'My Website')

6. **Closing the Connection:**

After you're done with the database operations, make sure to close the connection:

connection.close()

2. Python Multithreading

Python's **multithreading** module allows for concurrent execution of code. Threads run in parallel and share the same memory space, which makes it easier to share data between threads, but requires proper synchronization to avoid conflicts.

Understanding Threads:

A thread is a unit of execution within a process. By default, Python programs run in a single thread (main thread), but you can create additional threads to perform tasks concurrently.

Creating and Starting Threads:

To work with threads in Python, you can use the threading module. Here's how to create and start threads:

```
import threading


# Function to be executed by each thread
def print_numbers():
    for i in range(5):
        print(i)


# Creating threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_numbers)


# Starting threads
thread1.start()
thread2.start()


# Joining threads to ensure they complete before continuing
thread1.join()
thread2.join()


print("Both threads have completed.")
```

In this example, thread1 and thread2 will run the print_numbers function concurrently.

Forking Threads:

You can create multiple threads for concurrent execution, which helps perform multiple tasks at once. Thread forking refers to creating several threads that will execute tasks concurrently.

```
def task(name):
```

```
    print(f"Task {name} is running")

# Create multiple threads
threads = []
for i in range(5):
    thread = threading.Thread(target=task, args=(i,))
    threads.append(thread)
    thread.start()

# Wait for all threads to finish
for thread in threads:
    thread.join()
```

Synchronizing Threads:

When threads share data or resources, it's essential to synchronize their access to avoid data corruption or race conditions. Python provides a Lock class to handle this synchronization.

```
import threading

# Shared resource
counter = 0
lock = threading.Lock()

# Function to increment counter
def increment():
    global counter
    with lock:  # Acquire lock before accessing shared resource
        local_counter = counter
        local_counter += 1
        counter = local_counter

# Creating threads to modify the counter
threads = []
for _ in range(100):
    thread = threading.Thread(target=increment)
    threads.append(thread)
```

```
    thread.start()

# Wait for all threads to complete
for thread in threads:
    thread.join()

print(f"Final counter value: {counter}")
```

In this example, the lock ensures that only one thread can modify the counter at a time, avoiding conflicts.

Programming Using Multithreading:

You can use multithreading for various use cases, such as:

- I/O-bound tasks (e.g., reading from files, network operations).
- Performing tasks concurrently to improve performance.
- Running parallel tasks that are independent of each other.

For instance, if you need to download multiple files at once or process several items concurrently, multithreading can help.

Example: Downloading files concurrently:

```
import threading
import requests

def download_file(url):
    response = requests.get(url)
    with open(url.split('/')[-1], 'wb') as file:
        file.write(response.content)

# List of URLs to download
urls = [
    "https://example.com/file1.jpg",
    "https://example.com/file2.jpg",
    "https://example.com/file3.jpg"
]

threads = []
for url in urls:
    thread = threading.Thread(target=download_file, args=(url,))
```

```
    threads.append(thread)
    thread.start()

# Wait for all threads to finish
for thread in threads:
    thread.join()
```

This example uses threads to download multiple files concurrently, which can improve performance by performing I/O tasks in parallel.

Conclusion

In this unit, we covered **Database Interaction** in Python, particularly focusing on SQLite. We learned how to connect to databases, create and query tables, insert data, and manage configuration settings in the database. Additionally, we explored **Python Multithreading**, which allows concurrent execution of tasks, and discussed thread creation, forking, synchronization, and practical use cases of multithreading in Python programs. These topics are essential for building efficient and scalable applications that interact with databases and utilize concurrency.