Fundamentals of Python

1.1 Introduction

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python emphasizes code readability and allows programmers to express concepts in fewer lines of code compared to other languages.

Key Features:

- **Easy to read and write**: Python's syntax is simple and intuitive.
- **Interpreted**: Python code is executed line by line.
- **Dynamically typed**: You don't need to declare the type of variable while writing the code.
- **Extensive libraries**: Python has a wide variety of libraries like NumPy, Pandas, Matplotlib, etc., for different applications.

1.2 Keywords, Identifiers, Literals, Operators

- **Keywords**: Reserved words in Python that have a special meaning, like `if`, `else`, `while`, `for`, `break`, etc.
- **Identifiers**: Names given to variables, functions, classes, etc. They must start with a letter or underscore (_).
- **Literals**: Constant values assigned to variables. Examples include `10` (integer literal), `"hello"` (string literal).
- **Operators**: Symbols used to perform operations. Categories include:
  - **Arithmetic Operators**: `+`, `-`, `*`, `/`
  - **Comparison Operators**: `==`, `!=`, `<`, `>`
  - **Logical Operators**: `and`, `or`, `not`

1.3 Data Types

- **Numbers**: Integers, floating points, and complex numbers.
  - Example: `a = 5`, `b = 3.14`, `c = 2 + 3j`
- **Strings**: A sequence of characters enclosed in quotes.
  - Example: `name = "Python"`
- **Lists**: Ordered, mutable collections of items.
  - Example: `my_list = [1, 2, "Hello"]`
- **Tuples**: Ordered, immutable collections of items.
  - Example: `my_tuple = (1, 2, "Hello")`
- **Dictionaries**: Unordered collections of key-value pairs.
  - Example: `my_dict = {"name": "John", "age": 25}`
- **Sets**: Unordered collections of unique items.
  - Example: `my_set = {1, 2, 3, 4}`

1.4 Understanding Python Blocks

Python uses indentation (whitespace) to define the blocks of code, unlike other programming languages that use curly braces `{}`.

Example of a block:

```
css
Copy code
if a > b:
    print("a is greater")
else:
    print("b is greater")
```

The indentation after `if` and `else` defines the code blocks for each condition.

1.5 Control Flow - `if`, `else`, `elif`

Python provides `if`, `else`, and `elif` statements to control the flow of the program based on conditions.

- **if**: Executes if a condition is true.
- **elif**: Checks another condition if the previous one was false.
- **else**: Executes if all conditions are false.

Example:

```
bash
Copy code
if marks > 90:
    print("Excellent")
elif marks > 75:
    print("Good")
else:
    print("Needs Improvement")
```

1.6 Loops - `while`, `for`, `continue`, `break`

- **while loop**: Repeats as long as the condition is true.
    - Example:

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

- **for loop**: Iterates over a sequence (list, tuple, string).
    - Example:

```
scss
Copy code
for i in range(5):
    print(i)
```

- **break**: Exits the loop when encountered.
- **continue**: Skips the current iteration and moves to the next.
    - Example:

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

1.7 Loop Manipulation Using `pass`, `continue`, `break`, and `else`

- **pass**: Used when a statement is required syntactically but no action is needed.
    - Example:

```
python
Copy code
for i in range(5):
    pass
```

- **continue**: Skips the current iteration and proceeds with the next one.
- **break**: Terminates the loop entirely.
- **else in loops**: Executes after the loop completes normally (without a `break`).
    - Example:

```
arduino
Copy code
for i in range(5):
    if i == 3:
        break
else:
    print("Loop completed")
```

1.8 For Loop Using Ranges, Strings, Lists, and Dictionaries

- **Range**:

```
scss
Copy code
for i in range(1, 6):
    print(i)
```

- **String**:

```
lua
Copy code
for char in "Python":
    print(char)
```

- **List**:

```
css
Copy code
my_list = [10, 20, 30]
for item in my_list:
    print(item)
```

- **Dictionary**:

```
vbnet
Copy code
my_dict = {"name": "Alice", "age": 25}
for key, value in my_dict.items():
    print(key, value)
```

1.9 Programming Using Python Conditional and Loops Block

Python allows combining conditionals and loops to build complex logic.

Example:

```
python
Copy code
students = ["John", "Jane", "Alice", "Bob"]
for student in students:
    if student == "Alice":
        print("Special student found!")
    else:
        print(f"{student} is a regular student.")
```

1.10 Comprehensions on List, Tuple, Dictionaries

Comprehensions provide a concise way to create lists, tuples, and dictionaries. They offer a more readable and compact alternative to traditional loops for generating these data structures.

List Comprehension

List comprehension allows you to create a new list by iterating over an existing iterable (like a list or range) and applying an optional condition.

Syntax:

```
csharp
Copy code
[expression for item in iterable if condition]
```

Example:

```
less
Copy code
squares = [x**2 for x in range(1, 6)]
# Output: [1, 4, 9, 16, 25]
```

Example with condition:

```
csharp
Copy code
even_squares = [x**2 for x in range(1, 6) if x % 2 == 0]
# Output: [4, 16]
```

Tuple Comprehension (Using `tuple()` with Generator)

Tuple comprehension doesn't exist directly like list comprehension. You use a generator expression and pass it to the `tuple()` constructor.

Example:

```
python
Copy code
squares_tuple = tuple(x**2 for x in range(1, 6))
# Output: (1, 4, 9, 16, 25)
```

Dictionary Comprehension

Dictionary comprehension allows you to create dictionaries by iterating over an iterable and defining key-value pairs.

Syntax:

```
css
Copy code
{key: value for item in iterable if condition}
```

Example:

```
css
Copy code
squares_dict = {x: x**2 for x in range(1, 6)}
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Example with condition:

```
css
Copy code
even_squares_dict = {x: x**2 for x in range(1, 6) if x % 2 == 0}
# Output: {2: 4, 4: 16}
```

Functions, Modules & Packages, Exception Handling

2.1. Function Basics: Scope, Nested Functions, Non-local Statements

- **Scope**: Refers to the visibility of variables. Python has different scopes:
  - **Local Scope**: Variables declared inside a function, accessible only within that function.
  - **Global Scope**: Variables declared outside of any function, accessible from any part of the code.
  - **Enclosed Scope**: Variables defined in the outer function of a nested function.
  - **Built-in Scope**: Predefined variables like `print`, `len`, etc.

Example of scope:

```
python
Copy code
def outer():
    x = "outer variable"
    def inner():
        nonlocal x
        x = "inner variable"
    inner()
    print(x)

outer()  # Output: inner variable
```

2.2. Built-in Functions

- Python provides several **built-in functions** like `print()`, `len()`, `max()`, `type()`, etc.
  - Example:

```
python
Copy code
print(len("MCA"))  # Output: 3
print(max(3, 4, 5))  # Output: 5
```

2.3. Types of Functions: Anonymous Functions (Lambda)

- **Anonymous Function (Lambda)**: Small one-line functions defined using `lambda` keyword.
  - Example:

```
python
Copy code
square = lambda x: x * x
print(square(5))  # Output: 25
```

- **Types of Functions**:
  - **User-defined functions**: Created by the user using `def`.
  - **Built-in functions**: Provided by Python like `print()`, `len()`.
  - **Recursive functions**: Functions that call themselves.
  - **Higher-order functions**: Functions that accept other functions as parameters.

2.4. Decorators and Generators

- **Decorators**: Functions that modify the behavior of other functions.
  - Example:

```
python
Copy code
def decorator(func):
    def wrapper():
        print("Before calling the function")
        func()
        print("After calling the function")
    return wrapper

@decorator
def greet():
    print("Hello, MCA!")

greet()
```

- **Generators**: Functions that return an iterator using `yield`.
  - Example:

```
python
Copy code
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for i in countdown(3):
    print(i)
```

2.5. Modules: Module Basic Usage, Creating, Importing Modules

- A **module** is a file containing Python code, like functions or classes, that can be imported into other Python scripts.
  - Example of creating a module:

```
python
Copy code
# mymodule.py
def greet(name):
    return f"Hello, {name}"
```

- **Importing Modules**:

```
python
Copy code
import mymodule
print(mymodule.greet("MCA"))  # Output: Hello, MCA
```

2.6. Importing Functions and Variables from Different Modules

- You can import specific functions or variables from modules.

```
python
Copy code
from mymodule import greet
print(greet("MCA"))  # Output: Hello, MCA
```

2.7. Python Built-in Modules: math, random, datetime

- **math module**: Provides mathematical functions.

```
python
Copy code
import math
print(math.sqrt(16))  # Output: 4.0
```

- **random module**: Used to generate random numbers.

```
python
Copy code
import random
print(random.randint(1, 10))  # Output: Random number between 1 and 10
```

- **datetime module**: Handles date and time.

```
python
Copy code
from datetime import datetime
print(datetime.now())  # Output: Current date and time
```

2.8. Package: Import Basics

- A **package** is a collection of modules grouped together in directories.
    - Importing a package:

```
python
Copy code
from package_name.module_name import function_name
```

2.9. Python Namespace Packages

- **Namespace packages**: Packages without an `__init__.py` file that allow modules spread across directories to be grouped under the same package name.
    - Usage:

```
python
Copy code
from mynamespace.module1 import function1
```

2.10. User-Defined Modules and Packages

- You can create your own **modules** and **packages** for reuse.
    - Example:

```
python
Copy code
# directory structure
# mypackage/
# └── __init__.py
# └── mymodule.py

from mypackage.mymodule import myfunction
```

2.11. Exception Handling

- **Exception Handling**: Used to handle runtime errors and prevent program crashes.
  - Example:

```
python
Copy code
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Division by zero is not allowed!")
```

2.11.1 Avoiding Code Break Using Exception Handling

- Using **try-except** blocks avoids breaking the code even when an error occurs.

```
python
Copy code
try:
    file = open("non_existent_file.txt")
except FileNotFoundError:
    print("File not found")
```

2.11.2 Safeguarding File Operation Using Exception Handling

- Exception handling in file operations prevents crashes if the file is not found or cannot be opened.

```
python
Copy code
try:
    with open("file.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
    print("The file does not exist")
```

2.11.3 Handling Multiple and User-Defined Exceptions

- **Multiple exceptions** can be handled by specifying multiple `except` blocks.

```
python
Copy code
try:
    x = 1 / 0
except ZeroDivisionError:
```

```
    print("Division by zero")
except ValueError:
    print("Invalid value")
```

- **User-defined exceptions** are created by inheriting from `Exception`.

```
python
Copy code
class MyError(Exception):
    pass

try:
    raise MyError("Something went wrong")
except MyError as e:
    print(e)
```

2.11.4 Handling and Helping Developer with Error Code

- By providing informative error messages, developers can easily trace issues.

```
python
Copy code
try:
    raise ValueError("Incorrect value!")
except ValueError as e:
    print(f"Error: {e}")
```

2.11.5 Programming Using Exception Handling

- Exception handling is integral in building robust programs that can manage unexpected situations without crashing.

```
python
Copy code
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return "Division by zero is not allowed"

print(divide(10, 2))  # Output: 5.0
print(divide(10, 0))  # Output: Division by zero is not allowed
```

Python Object-Oriented Programming

3.1 Concept of Class, Object, and Instances, Method Call, Real-Time Use of Class in Live Projects

- **Class**: A blueprint or template for creating objects (instances). It encapsulates data for the object (attributes) and functionalities (methods).
- **Object**: An instance of a class. Each object can have unique values for its attributes.
- **Instance**: A particular realization of a class. Each instance (object) of a class can have different attribute values.
- **Method Call**: Methods in a class are called using the dot `.` operator with the object.

Example:

```
python
Copy code
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def show_details(self):
        return f"Car: {self.brand} {self.model}"

car1 = Car("Tesla", "Model S")
print(car1.show_details())
```

- **Real-time Use**: Classes are widely used in real-world applications such as building software systems like billing systems, gaming applications, and database management, where different instances of a class represent specific components.

3.2 Constructor, Class Attributes, and Destructors

- **Constructor (`__init__`)**: A special method in Python called when an object is created. It is used to initialize the object's attributes.
- **Class Attributes**: Attributes shared across all instances of a class.
- **Destructor (`__del__`)**: A method called when an object is deleted or no longer in use. It cleans up resources.

Example:

```
python
Copy code
class Employee:
    company_name = "TechCorp"  # Class Attribute

    def __init__(self, name, position):
        self.name = name  # Instance Attribute
        self.position = position

    def __del__(self):
        print(f"Object for {self.name} destroyed")
```

```
emp1 = Employee("Alice", "Developer")
```

3.3 Inheritance, Super Class, Method Overriding

- **Inheritance**: Mechanism by which one class can inherit attributes and methods from another class.
- **Super Class**: The class from which a subclass inherits. Also called parent class.
- **Method Overriding**: Redefining a method in a subclass that exists in the parent class.

Example:

```
python
Copy code
class Animal:
    def sound(self):
        return "Some sound"

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

dog = Dog()
print(dog.sound())  # Method Overriding
```

3.4 Overloading Operators

- **Operator Overloading**: Defining how operators like `+`, `-`, `*`, etc., behave when applied to objects of a class.

Example:

```
python
Copy code
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2  # Overloading the `+` operator
```

3.5 Static and Class Methods

- **Static Method**: A method that belongs to the class, not the instance, and cannot access instance-specific data.
- **Class Method**: A method that has access to the class itself using the `cls` parameter.

Example:

```
python
```

```
Copy code
class MyClass:
    @staticmethod
    def static_method():
        return "This is a static method"

    @classmethod
    def class_method(cls):
        return f"This is a class method of {cls}"

print(MyClass.static_method())
print(MyClass.class_method())
```

3.6 Delegation and Containership

- **Delegation**: When an object passes a task to another object (often by calling another method or class).
- **Containership (Composition)**: When one class contains objects of another class as its members (a "has-a" relationship).

Example:

```
python
Copy code
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()  # Containership

    def start(self):
        return self.engine.start()  # Delegation

my_car = Car()
print(my_car.start())
```

3.7 Python Regular Expressions

- **Regular Expressions (regex)**: A tool for pattern matching in strings.

3.7.1 Pattern Matching and Searching Using Regex in Python

- You can use the `re` module to find patterns in strings.

Example:

```
python
Copy code
import re
pattern = r"\d+"  # Pattern to find digits
text = "There are 123 apples"
match = re.findall(pattern, text)
print(match)
```

3.7.2 Real-Time Parsing of Data Using Regex

- Regex is often used in parsing logs, user input, or files for specific patterns (like dates, times, etc.).

3.7.3 Applications of Regex

- **Password Validation**: Ensuring that the password contains letters, numbers, and special characters.
- **Email Validation**: Verifying that an email follows a proper format like `example@domain.com`.
- **URL Validation**: Ensuring that URLs are well-formed.

Example (for email validation):

```
python
Copy code
pattern = r"[^@]+@[^@]+\.[^@]+"
email = "test@example.com"
if re.match(pattern, email):
    print("Valid email")
```

3.8 Multithreading

- **Multithreading**: Running multiple threads (lightweight processes) in parallel to execute tasks concurrently.

3.8.1 Understanding Threads

- A thread is the smallest unit of a process. Python provides the `threading` module to handle threads.

3.8.2 Synchronizing Threads

- Thread synchronization ensures that threads access shared resources in a controlled manner using locks.

Example:

```
python
Copy code
import threading

lock = threading.Lock()

def critical_section():
    with lock:
        print("Thread-safe operation")

thread1 = threading.Thread(target=critical_section)
thread1.start()
```

3.8.3 Programming Using Multithreading

- Python programs can handle multiple tasks like I/O operations, data processing, and more using multiple threads.

4.1 Introduction to NoSQL Database

NoSQL databases are designed to handle large volumes of unstructured or semi-structured data. Unlike traditional SQL databases that use fixed schemas and tables, NoSQL databases are more flexible, allowing for various data types and structures. They excel in distributed environments and can scale horizontally.

Example: Imagine a social media application where user profiles can vary greatly. Some users may have additional fields like "bio" or "location," while others may not. A NoSQL database can accommodate this variability without requiring a predefined schema.

4.2 Types of NoSQL

NoSQL databases can be categorized into four main types:

4.2.1 Document-Based: MongoDB

MongoDB stores data in flexible, JSON-like documents called BSON (Binary JSON). Each document can have a different structure, making it suitable for a wide range of applications.

Example: In a MongoDB database for a bookstore, you might have a collection named "books" containing documents like:

- Document 1:

```
json
Copy code
{ "title": "1984", "author": "George Orwell", "published": 1949 }
```

- Document 2:

```
json
Copy code
{ "title": "The Great Gatsby", "author": "F. Scott Fitzgerald", "published": 1925,
"genres": ["Fiction", "Classic"] }
```

4.2.2 Key-Value Database: Couchbase

Couchbase stores data as key-value pairs. Each key is unique and maps to a specific value, which can be a simple object or a more complex structure.

Example: In a user session management system, you might store session data as:

- Key: `"session:12345"`
- Value:

```
json
Copy code
{ "user_id": 42, "expires": "2024-01-01T12:00:00Z" }
```

4.2.3 Wide-Column Databases: Cassandra

Cassandra organizes data in rows and columns but allows each row to have a variable number of columns. It is designed for high availability and performance.

Example: For a user activity tracking system, you might have:

- Row Key: `"user_1"` → Columns:

```
json
Copy code
{ "login_time": "2024-01-01T08:00:00Z", "page_views": 25 }
```

- Row Key: `"user_2"` → Columns:

```
json
Copy code
{ "login_time": "2024-01-01T09:30:00Z", "page_views": 10, "last_activity": "2024-
01-01T10:00:00Z" }
```

4.2.4 Graph/Node Databases: Neo4j

Neo4j stores data in the form of nodes, relationships, and properties, making it ideal for applications that require deep relationship analysis.

Example: In a social networking application, you might represent relationships like:

- Node: Alice (type: Person)
- Node: Bob (type: Person)
- Relationship:

```
css
Copy code
Alice --FRIENDS_WITH--> Bob
```

4.3 SQL vs. NoSQL

- **SQL Databases**: Use a structured query language for defining and manipulating data. They require a fixed schema and support ACID transactions, ensuring reliability.
- **NoSQL Databases**: Allow for flexible schemas and are optimized for scalability and speed. They do not require complex joins or fixed structures.

Comparison Example:

- **SQL**:

```
sql
Copy code
CREATE TABLE users (id INT PRIMARY KEY, name VARCHAR(100), age INT);
```

- **NoSQL (MongoDB)**: You can directly insert documents without defining a table:

```
lua
Copy code
db.users.insert({ "name": "Alice", "age": 30 });
```

4.4 Introduction to MongoDB with Python

To interact with MongoDB in Python, you can use the `PyMongo` library. This library allows you to connect to MongoDB, perform CRUD operations, and manage your database.

4.5 Installing MongoDB on Windows

1. **Download MongoDB**: Go to the official MongoDB website and download the latest version for Windows.
2. **Run the Installer**: Execute the installer and follow the setup instructions.
3. **Set Environment Variables**: Add the MongoDB `bin` folder to your system PATH to access MongoDB commands from the command line.
4. **Start the MongoDB Server**: Open Command Prompt and run `mongod` to start the server.

4.6 Exploring Collections and Documents

In MongoDB, a database consists of collections, and each collection contains documents.

Example:

- **Database**: "school"
- **Collection**: "students"
- **Document**:

```
json
Copy code
{ "name": "Frank", "grade": "A", "age": 21 }
```

You can retrieve all documents in the "students" collection with a command like:

```
lua
Copy code
db.students.find()
```

4.7 Performing CRUD Operations

CRUD stands for Create, Read, Update, and Delete, which are the basic operations for managing data in a database.

- **Create**: To add a new document to a collection. For example:

```
lua
Copy code
db.students.insert({ "name": "Grace", "grade": "B", "age": 20 });
```

- **Read**: To retrieve documents based on specific criteria. For example:

```
lua
Copy code
db.students.find({ "grade": "A" });
```

- **Update**: To modify an existing document. For example:

```
bash
Copy code
db.students.update({ "name": "Frank" }, { "$set": { "age": 22 } });
```

- **Delete**: To remove a document from a collection. For example:

```
csharp
Copy code
db.students.remove({ "name": "Grace" });
```

4.8 Commit, Rollback, and Cursor Operation

In MongoDB, transactions are handled through sessions, allowing you to group operations.

- **Commit**: Saves all changes made during a transaction.
- **Rollback**: Reverts changes if an error occurs.
- **Cursor**: A pointer to the results of a query, allowing you to iterate through documents.

Example: To perform a transaction:

```
python
Copy code
with client.start_session() as session:
    session.start_transaction()
    try:
        # Operations
        session.commit_transaction()
    except:
        session.abort_transaction()
```

4.9 Handling Errors

Error handling is crucial when working with databases. Common exceptions may include connection errors or data validation failures.

Example: You can use try-except blocks to manage errors when performing database operations. For instance:

```
css
Copy code
try:
    db.students.insert({ "name": "Hank", "grade": "C" });
except pymongo.errors.DuplicateKeyError:
    print("Duplicate entry detected.")
```

This way, you can ensure your application handles errors gracefully without crashing.