**Name – Sakshi Soni**

**Registered email – sonisakshi3031@gmail.com**

**Course name – Data Analyst**

**Assignment name - First Assignment**

**Submission date – 12/11/2024**

**Registered mobile no. - 7974984923**

## 1. Explain the key features of Python that make it a popular choice for programming ?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991. Python's popularity stems from its versatility, simplicity and extensive libraries. Key features include:

1. **Simple Syntax**: Python's clear and readable syntax makes it easy to learn and write code quickly.
2. **Interpreted Language**: Python is interpreted, which allows for easier debugging and rapid development without the need for compilation.
3. **Dynamic Typing**: Variables don't need explicit type declarations, offering flexibility and simplicity in coding.
4. **Extensive Libraries**: Python's rich standard library and large ecosystem of third-party packages help solve a wide variety of problems without reinventing the wheel.
5. **Cross-Platform**: Python runs on all major operating systems (Windows, macOS, Linux), making it versatile for different environments.
6. **Community Support**: A large, active community means plenty of resources, tutorials, and frameworks for every kind of project.
7. **Versatility**: Python is used in web development, data science, machine learning, automation, and more, supporting various programming paradigms.
8. **Integration**: Python easily integrates with other languages and technologies, making it useful for diverse projects.

These features make Python an accessible, efficient, and powerful language for both beginners and experienced developers.

1. **Describe the role of predefined keywords in Python and provide examples of how they are used in a program ?**

In Python, **predefined keywords** (also known as **reserved words**) are special identifiers that have a specific meaning in the language. These keywords cannot be used as identifiers (e.g., variable names, function names, etc.) because they are reserved by the Python interpreter for specific syntactic purposes. They define the structure of the Python language and control the flow of programs.

### List of Common Python Keywords:

Here are some examples of the most commonly used Python keywords (note that this is not a complete list, but it includes the most relevant ones):

- `False`, `True` – Boolean values
- `None` – Represents the absence of a value

- `if`, `else`, `elif` – Conditional statements
- `for`, `while` – Looping structures
- `break`, `continue` – Control flow within loops
- `def`, `return` – Function definition and return values
- `class` – Class definition
- `try`, `except`, `finally`, `raise` – Exception handling
- `import`, `from`, `as` – Importing modules
- `with` – Context management (e.g., file handling)
- `lambda` – Anonymous function
- `global`, `nonlocal` – Variable scope declaration
- `and`, `or`, `not` – Logical operators
- `is`, `is not` – Object identity comparison
- `in`, `not in` – Membership operators
- `del` – Deletes an object
- **Conditional Statements (`if`, `else`, `elif`)**:

- `x = 10`
- `if x > 5:`
- `    print("x is greater than 5")`
- `elif x == 5:`
- `    print("x is equal to 5")`
- `else:`
- `    print("x is less than 5")`
- **Defining Functions (`def`, `return`)**:
- `def add(a, b):`
- `    return a + b`
- 
- `result = add(3, 4)`
- `print(result)  # Output: 7`
- **Loops (`for`, `while`, `break`, `continue`)**:
- `for i in range(5):`
- `    if i == 3:`
- `        continue  # Skip iteration when i is 3`
- `    print(i)`
- `# Output: 0, 1, 2, 4`
- 
- **Exception Handling (`try`, `except`, `finally`)**:
- `    num = int(input("Enter a number: "))`
- `except ValueError:`
- `    print("That's not a valid number!")`
- `finally:`
- `    print("Execution completed.")`
- **Class Definition (`class`)**:
- `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'm {self.age} years old.")`
- 
- `p = Person("Alice", 30)`
- `p.greet()  # Output: Hello, my name is Alice and I'm 30 years`
- **Variable Scope (`global`, `nonlocal`)**:
- `x = 10  # global variable`
- 
- `def func():`

```
    global x
    x = 20  # modifies the global x
    print(x)

func()  # Output: 20
print(x)  # Output: 20 (global x was modified)
```

2. Compare and contrast mutable and immutable objects in Python with examples.

In Python, **mutable** and **immutable** objects differ in whether their values can be changed after creation.

## Mutable Objects

- **Definition**: Mutable objects can be modified after they are created.
- **Examples**: `list`, `dict`, `set`
- **Behavior**: Changes to the object will affect all references to that object.

*Example (Mutable: List):*

```
python
Copy code
my_list = [1, 2, 3]
my_list.append(4)  # Modify the list
print(my_list)  # Output: [1, 2, 3, 4]
```

In this case, the list can be modified in place, and any reference to `my_list` will see the updated version.

## Immutable Objects

- **Definition**: Immutable objects cannot be modified after they are created. Any operation that seems to modify them creates a new object.
- **Examples**: `int`, `str`, `tuple`
- **Behavior**: Modifying an immutable object creates a new object rather than altering the original one.

*Example (Immutable: Tuple):*

```
python
Copy code
my_tuple = (1, 2, 3)
# my_tuple[1] = 4  # This would raise an error (TypeError: 'tuple' object does not
support item assignment)
new_tuple = my_tuple + (4,)  # Create a new tuple
print(new_tuple)  # Output: (1, 2, 3, 4)
```

Here, attempting to modify the tuple directly would cause an error. Instead, a new tuple is created by concatenation.

## Key Differences:

| Feature | Mutable Objects | Immutable Objects |
|---|---|---|

| Feature | Mutable Objects | Immutable Objects |
|---|---|---|
| **Modification** | Can be changed in place. | Cannot be changed; operations create new objects. |
| **Examples** | `list`, `dict`, `set` | `int`, `str`, `tuple`, `frozenset` |
| **Effect on References** | All references to the object see changes. | References remain unchanged; any modification creates a new object. |

In Python, operators are symbols used to perform operations on variables and values. The main types of operators in Python are:

## 4.Discuss the different types of operators in Python and provide examples of how they are used

Python supports a wide variety of operators that are used to perform various operations on variables **and** values.
Operators **in** Python can be classified into several categories based on their functionality. Below **is** an overview of the different types of operators **in** Python along **with** examples of how they are used:
1. Arithmetic Operators
These operators are used to perform mathematical operations.
**+,-,*,/,%**
2. Comparison (Relational) Operators
These operators are used to compare two values **and return** a boolean result (**True or False**).
**==,>,<,>=,<=**
3. Logical Operators
Logical operators are used to combine conditional statements.
**and,or,not**
4. Assignment Operators
Assignment operators are used to assign values to variables.
**=,+=**

```
#Arithmetic Operators
a = 10
b = 3

print(a + b)
```

```
13
```

In [23]:

```
#Comparison  Operators
a = 10
b = 5

print(a == b)
```

```
False
```

In [24]:

```
#Logical Operators
a = True
b = False

print(a and b)
```

```
false
```

5.Explain the concept of type casting **in** Python **with** examples.

In [ ]:

Type Casting **in** Python
Type casting **in** Python refers to the process of converting one data type into another.
This can be done explicitly **or** implicitly
Python provides built**-in** functions to facilitate explicit type casting.

1. Implicit Type Conversion (Automatic Type Conversion)
Python performs implicit type conversion automatically when you operate on mixed data types.
For example, when an operation involves an integer **and** a float, Python will convert the integer to a float to ensure the operation proceeds.

```
# Implicit type conversion
x = 5    # integer
y = 3.2  # float
result = x + y
print(result)
# Implicit type conversion
x = 5    # integer
y = 3.2  # float
result = x + y
print(result)
```

```
8.2
```

```
#Converting an integer to a float
x = 10
y = float(x)
print(y)
```

```
10.0
```

6.How do conditional statements work **in** Python? Illustrate **with** examples.

Conditional Statements **in** Python
Conditional statements **in** Python allow you to execute certain blocks of code based on whether a given condition **is True or False.**
The basic conditional statements **in** Python are **if, elif, and else.**

1. **if** Statement
The **if** statement **is** used to check a condition **and** execute a block of code **if** the condition evaluates to **True.**

```
# Simple if statement
age = 18
if age >= 18:
    print("You are eligible to vote.")
```

```
You are eligible to vote.
```

2. **if-else** Statement
The **else** statement can be used to define a block of code that will execute when the **if** condition **is False.**

```
# if-else statement
age = 16
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

```
You are not eligible to vote.
```

3. **if-elif-else** Statement

The **elif** (short **for** "else if") statement **is** used to check multiple conditions. You can have multiple **elif** blocks between an **if and else** to check **for** more than two conditions.

```
# if-elif-else statement
day = "Monday"

if day == "Monday":
    print("Start of the week!")
elif day == "Wednesday":
    print("Midweek!")
elif day == "Friday":
    print("Weekend is near!")
else:
    print("Another day!")
```

```
Start of the week!
```

7.Describe the different types of loops **in** Python **and** their use cases **with** examples.

In Python, there are three main types of loops: **for** loops, **while** loops, **and** nested loops. Each loop type **is** useful **in** different situations depending on the task at hand. Let's go over each one with explanations and examples:

1. **for** Loop
The **for** loop **is** used when you want to iterate over a sequence of elements (like a list, tuple, string, **or** range). It **is** particularly useful when you know the number of iterations **or** you are working **with** an iterable object.

:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

```
apple
banana
cherry
```

2. **while** Loop
The **while** loop **is** used when you want to repeat a block of code **as** long **as** a certain condition **is True**. It **is** useful when you don't know in advance how many times the loop should run, but you have a condition that determines when to stop.

```
count = 0
while count < 5:
    print(count)
    count += 1  # Increment the counter
```

```
0
1
2
3
4
```

3. Nested Loops
A nested loop **is** a loop inside another loop. This **is** useful when dealing **with** multidimensional data structures (like lists of lists, matrices, etc.), **or** when you need to perform more complex iteration.

```
matrix = [
```

```
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row in matrix:
    for element in row:
        print(element, end=" ")
    print()  # Move to the next line after each row
```

```
1 2 3
4 5 6
7 8 9
```