In [ ]:
```
#python_basics_assignment
```

Explain the key features of python that make it a popular choice for programming

Python's popularity as a programming language can be attributed to several key features:

1. Readability and Simplicity:

Clear Syntax: Python's syntax is designed to be easy to read and understand, with a focus on natural language constructs. Indentation: Python uses indentation to define code blocks, making the structure of the code more visually apparent. Minimalist Syntax: Python avoids unnecessary punctuation and keywords, reducing the complexity of the language.

1. Versatility:

General-Purpose Language: Python can be used for a wide range of applications, from web development and data analysis to scientific computing and machine learning.   Large Standard Library: Python's extensive standard library provides a rich set of modules and functions for various tasks, reducing the need for external dependencies. Third-Party Libraries: Python has a vast ecosystem of third-party libraries and frameworks, such as NumPy, Pandas, TensorFlow, and Django, that can be easily integrated into projects.

1. Cross-Platform Compatibility:

Runs on Multiple Platforms: Python code can be written once and run on different operating systems, including Windows, macOS, and Linux. Portable Code: Python's portability makes it easier to develop and deploy applications across various platforms.

1. Strong Community and Support:

Active Community: Python has a large and active community of developers who contribute to the language's development, create new libraries, and provide support to other users. Extensive Documentation and Tutorials: Python's official documentation and numerous online resources offer comprehensive information and tutorials for learning and using the language.

1. Dynamic Typing:

Flexible Variable Types: Python allows you to assign different data types to variables at runtime, making the language more flexible and dynamic.

1. Object-Oriented Programming (OOP):

Supports OOP Concepts: Python fully supports OOP principles, including classes, objects, inheritance, and polymorphism.

1. Memory Management:

Automatic Garbage Collection: Python's built-in garbage collector automatically manages memory allocation and deallocation, reducing the risk of memory leaks.

1. Efficiency and Performance:

Optimized Interpreter: Python's interpreter has been optimized for performance, and there are also tools and libraries available for accelerating Python code. These features, combined with its ease of learning and use, make Python a popular choice for programmers of all levels, from beginners to experienced professionals.

Describe the role of predefined keywords in python and provide examples of how they are used in a program.

In Python, keywords are reserved words that have specific meanings within the language. They cannot be used as variable names or identifiers. These keywords serve as the building blocks of Python programs, defining the structure, control flow, and operations.

Examples of Predefined Keywords and Their Uses:

1.Control Flow Keywords:

if, else, elif: Used for conditional statements to execute different code blocks based on certain conditions.

In [ ]:
```python
# code for if
age = 25

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

In [2]:
```python
#code for if-else
age = 10

if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
                                          #output: You are a minor.
```

```
You are a minor.
```

In [3]:
```python
#code for elif
age = 15

if age >= 18:
    print("You are an adult.")
elif age >= 13:
    print("You are a teenager.")
                                          #output: You are a teenager.
```

```
You are a teenager.
```

for: Used for iterating over sequences (like lists, tuples, or strings).

In [4]:
```
#code for "for loop"
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print(fruit)
                                            #output: apple
                                                    #banana
                                                    #orange
```

```
apple
banana
orange
```

while: Used for creating loops that continue as long as a given condition is true.

In [5]:
```
#code for while loop
count = 0
while count < 5:
    print(count)
    count += 1
```

```
0
1
2
3
4
```

break: Used to exit a loop prematurely.

In [6]:
```
#code for break
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num == 3:
        break
    print(num)
```

```
1
2
```

continue: Used to skip the current iteration of a loop and move to the next.

In [7]:
```
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num == 3:
        continue
    print(num)
```

```
1
2
4
5
```

Data Type Keywords:

int, float, str, bool, list, tuple, dict, set: Used to define different data types.

In [8]:
```
#code
num = 10  # int
pi = 3.14  # float
name = "Alice"  # str
is_student = True  # bool
fruits = ["apple", "banana"]  # list
```

```
coordinates = (1, 2)  # tuple
person = {"name": "Bob", "age": 30}  # dict
numbers = {1, 2, 3}  # set
```

Function Definition and Call:

def: Used to define a function. return: Used to return a value from a function.

In [9]:
```
#code
def greet(name):
    return "Hello, " + name + "!"

greeting = greet("Alice")
print(greeting)
```

```
Hello, Alice!
```

Class Definition:

class: Used to define a class for object-oriented programming.

In [11]:
```
#code
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

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

Other Keywords:

and, or, not: Used for logical operations. in, not in: Used for membership testing. is, is not: Used for identity comparison. try, except, else, finally: Used for exception handling. import, from: Used for importing modules. pass: Used as a placeholder for empty code blocks. These keywords form the foundation of Python programming and are essential for writing clear, concise, and efficient code.

compare and contrast mutable and immutable objects in python with examples.

In Python, objects can be classified into two categories: mutable and immutable. The key difference lies in their ability to be modified after creation.

Mutable Objects Mutable objects can be changed after they are created. This means that the value stored in the object can be altered.

Examples of mutable objects:

Lists:

In [13]:
```
#example of list
my_list = [1, 2, 3]
```

```
my_list.append(4)  # Modifying the list
print(my_list)  # Output: [1, 2, 3, 4]
```

```
[1, 2, 3, 4]
```

Dictionaries:

In [14]:
```
#example of dictionaries
my_dict = {"name": "Alice", "age": 30}
my_dict["age"] = 31  # Modifying the dictionary
print(my_dict)  # Output: {'name': 'Alice', 'age': 31}
```

```
{'name': 'Alice', 'age': 31}
```

Sets:

In [15]:
```
#example of set
my_set = {1, 2, 3}
my_set.add(4)  # Modifying the set
print(my_set)  # Output: {1, 2, 3, 4}
```

```
{1, 2, 3, 4}
```

Immutable Objects Immutable objects cannot be changed once they are created. If you want to modify an immutable object, you need to create a new object with the desired changes.

Examples of immutable objects:

Numbers (int, float, complex):

In [16]:
```
#example
x = 10
x = x + 5  # Creating a new integer object
print(x)  # Output: 15
```

```
15
```

Strings:

In [17]:
```
my_string = "Hello"
my_string = my_string + " world"  # Creating a new string object
print(my_string)  # Output: Hello world
```

```
Hello world
```

Tuples:

In [20]:
```
#example
my_tuple = (1, 2, 3)
# my_tuple[0] = 4  # This will raise an error
```

In summary:

Mutable objects are flexible and can be modified to suit changing requirements. Immutable objects are more efficient in terms of memory usage and are often used for data that doesn't need to be changed. Choosing between mutable and immutable objects depends on the specific use case and the desired behavior of your code.

Discuss the different types of operators in python and provide examples of how they are used.

Operators in Python Operators are symbols used to perform specific operations on operands (values or variables). Python supports various types of operators:

Arithmetic Operators Used for mathematical calculations. Addition: +: result = 5 + 3 Subtraction: -: result = 10 - 4 Multiplication: *: result = 2 * 6 Division: /: result = 15 / 3 Floor division: //: result = 17 // 5 (returns the quotient without the remainder) Modulus: %: result = 10 % 3 (returns the remainder) Exponentiation: **: result = 2 ** 3 (raises 2 to the power of 3)

In [21]:
```
#Examples of arithmetic operator
# Addition
result = 5 + 3
print(result)  # Output: 8

# Subtraction
result = 10 - 4
print(result)  # Output: 6

# Multiplication
result = 2 * 6
print(result)  # Output: 12

# Division
result = 15 / 3
print(result)  # Output: 5.0

# Floor division
result = 17 // 5
print(result)  # Output: 3

# Modulus
result = 10 % 3
print(result)  # Output: 1

# Exponentiation
result = 2 ** 3
print(result)  # Output: 8
```

```
8
6
12
5.0
3
1
8
```

Comparison Operators Used to compare values. Equal to: ==: result = 5 == 5 Not equal to: !=: result = 3 != 7 Greater than: >: result = 8 > 4 Less than: <: result = 2 < 9 Greater than or equal to: >=: result = 10 >= 10 Less than or equal to: <=: result = 6 <= 6

In [22]:
```
#Examples of comparison operator
# Equal to
result = 5 == 5
print(result)  # Output: True

# Not equal to
result = 3 != 7
```

```
print(result)  # Output: True

# Greater than
result = 8 > 4
print(result)  # Output: True

# Less than
result = 2 < 9
print(result)  # Output: True

# Greater than or equal to
result = 10 >= 10
print(result)  # Output: True

# Less than or equal to
result = 6 <= 6
print(result)  # Output: True
```

```
True
True
True
True
True
True
```

Assignment Operators Used to assign values to variables. Simple assignment: =: x = 5 Addition assignment: +=: x += 2 (equivalent to x = x + 2) Subtraction assignment: -=: x -= 3 Multiplication assignment: *=: x *= 4 Division assignment: /=: x /= 2 Floor division assignment: //=: x //= 3 Modulus assignment: %=: x %= 5 Exponentiation assignment: **=: x **= 2

In [23]:
```
#Examples of assignment operator
x = 5
x += 2
print(x)  # Output: 7

x -= 3
print(x)  # Output: 4

x *= 4
print(x)  # Output: 16

x /= 2
print(x)  # Output: 8

x //= 3
print(x)  # Output: 2

x %= 5
print(x)  # Output: 2

x **= 2
print(x)  # Output: 4
```

```
7
4
16
8.0
2.0
2.0
4.0
```

Logical Operators Used to combine Boolean expressions.

AND: and: result = True and False OR: or: result = True or False NOT: not: result = not True

In [24]:
```
#examples of logical operator
result = True and False
print(result)  # Output: False

result = True or False
print(result)  # Output: True

result = not True
print(result)  # Output: False
```

```
False
True
False
```

Bitwise Operators Used to perform operations on individual bits of binary numbers.

AND: &: result = 5 & 3 OR: |: result = 5 | 3 XOR: ^: result = 5 ^ 3 NOT: ~: result = ~5 Left shift: <<: result = 5 << 2 Right shift: >>: result = 5 >> 2

In [25]:
```
#examples of bitwise operator
result = 5 & 3
print(result)  # Output: 1

result = 5 | 3
print(result)  # Output: 7

result = 5 ^ 3
print(result)  # Output: 6

result = ~5
print(result)  # Output: -6

result = 5 << 2
print(result)  # Output: 20

result = 5 >> 2
print(result)  # Output: 1
```

```
1
7
6
-6
20
1
```

Membership Operators Used to check if a value is present in a sequence.

In: in: result = "a" in "apple" Not in: not in: result = "z" not in "apple"

In [26]:
```
#examples of membership operator
result = "a" in "apple"
print(result)  # Output: True

result = "z" not in "apple"
print(result)  # Output: True
```

```
True
True
```

Identity Operators Used to check if two objects are the same.

Is: is: result = x is y Is not: is not: result = x is not y

In [29]:
```
#example of identity operator
x = 5
y = 5
result = x is y
print(result)  # Output: True

x = [1, 2, 3]
y = [1, 2, 3]
result = x is y
print(result)  # Output: False (different objects, even though they have the same v
```

```
True
False
```

Explain the concept of type casting in python with examples

Type Casting in Python: Type casting, also known as type conversion, is the process of changing the data type of a variable or expression from one type to another. Python provides two primary methods for type casting: 1. Explicit Type Casting: Involves manually specifying the desired data type using specific functions: int(): Converts to an integer. float(): Converts to a floating-point number. str(): Converts to a string. bool(): Converts to a Boolean (True or False). list(): Converts to a list. tuple(): Converts to a tuple. dict(): Converts to a dictionary. set(): Converts to a set.

Examples

Integer Conversion (int()):

In [2]:
```
#code
x = int(3.14)  # x becomes 3
y = int("10")  # y becomes 10
z = int(True)  # z becomes 1
w = int(False)  # w becomes 0
```

Floating-Point Conversion (float()):

In [3]:
```
#code
x = float(10)  # x becomes 10.0
y = float("3.14")  # y becomes 3.14
z = float(True)  # z becomes 1.0
w = float(False)  # w becomes 0.0
```

Boolean Conversion (bool()):

In [4]:
```
#code
x = bool(10)  # x becomes True
y = bool(0)  # y becomes False
z = bool(3.14)  # z becomes True
w = bool("Hello")  # w becomes True
v = bool("")  # v becomes False
```

List Conversion (list()):

In [5]:
```
#code
x = list("Hello")  # x becomes ["H", "e", "l", "l", "o"]
y = list((1, 2, 3))  # y becomes [1, 2, 3]
z = list({1: "a", 2: "b"})  # z becomes [[1, "a"], [2, "b"]]
```

Tuple Conversion (tuple()):

In [6]:
```
#code
x = tuple("Hello")  # x becomes ("H", "e", "l", "l", "o")
y = tuple([1, 2, 3])  # y becomes (1, 2, 3)
z = tuple({1: "a", 2: "b"})  # z becomes ((1, "a"), (2, "b"))
```

Dictionary Conversion (dict()):

In [7]:
```
x = dict([("a", 1), ("b", 2)])  # x becomes {"a": 1, "b": 2}
y = dict(a=1, b=2)  # y becomes {"a": 1, "b": 2}
```

Set Conversion (set()):

In [8]:
```
#code
x = set("Hello")  # x becomes {"H", "e", "l", "o"}
y = set([1, 2, 3, 2])  # y becomes {1, 2, 3}
```

1. Implicit Type Casting:

Python automatically performs type casting in certain situations without explicit intervention. This typically occurs when arithmetic operations or comparisons involve different data types. For example:

In [11]:
```
#code
x = 10
y = 3.14

# Implicit type casting
z = x + y  # z becomes 13.14 (integer converted to float)
print(z)
```

```
13.14
```

Important Considerations: Not all type conversions are possible or meaningful. For example, converting a string to an integer might raise an error if the string doesn't represent a valid number. Be mindful of potential data loss or precision changes when converting between different numeric types. Understanding implicit type casting can help avoid unexpected behavior in your code.

Additional Notes: Python also provides functions like complex() to convert to complex numbers and chr() and ord() for converting between characters and integers. Custom classes can define their own type conversion methods using **int**, **float**, **str**, etc.

How do conditional statements work in python? illustrate with examples

Conditional Statements in Python

Conditional statements allow your Python code to make decisions based on certain conditions. They are essential for controlling the flow of your program. Python provides the following conditional statements:

if Statement:

Executes a block of code if a specified condition is true.

syntax:- if condition: # Code to be executed if condition is true

In [13]:
```
#Example
age = 18
if age >= 18:
    print("You are an adult.")
```

```
You are an adult.
```

if-else Statement:

Executes one block of code if a condition is true and another block if it's false.

syntax:- if condition: # Code to be executed if condition is true else: # Code to be executed if condition is false

In [14]:
```
#example
age = 17
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

```
You are a minor.
```

if-elif-else Statement: Provides multiple conditions to check, executing the first block that meets the condition. If none of the conditions are true, the else block is executed.

syntax:- if condition1: # Code to be executed if condition1 is true elif condition2: # Code to be executed if condition2 is true else: # Code to be executed if none of the conditions are true

In [16]:
```
#Example
grade = 90
if grade >= 90:
    print("Excellent!")
elif grade >= 80:
    print("Very good!")
elif grade >= 70:
    print("Good!")
else:
    print("Needs improvement.")
```

```
Excellent!
```

Nested Conditional Statements:

You can nest conditional statements within each other to create more complex decision-making structures.

In [18]:
```
#Example
age = 25
if age >= 18:
    if age >= 65:
        print("You are eligible for retirement.")
    else:
        print("You are an adult.")
else:
    print("You are a minor.")
```

```
You are an adult.
```

Key Points:

Conditions are evaluated as Boolean expressions (True or False). Indentation is crucial in Python to define the blocks of code within conditional statements. You can combine multiple conditions using logical operators (and, or, not).

Describe the different types of loops in python and their use cases with examples.

for Loop:

Iterates over a sequence of values (e.g., elements in a list, characters in a string, items in a dictionary). Useful for processing each item in a collection.

In [19]:
```
#Examples
# Iterating over a list
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print(fruit)

# Iterating over a string
name = "Python"
for char in name:
    print(char)

# Iterating over a range of numbers
for num in range(5):
    print(num)
```

```
apple
banana
orange
P
y
t
h
o
n
0
1
2
3
4
```

while Loop:

Executes a block of code as long as a specified condition is true. Useful for repeating actions until a certain condition is met.

In [20]:
```
#Examples
count = 0
while count < 5:
    print(count)
    count += 1

# Using a `while` loop to calculate the factorial of a number
num = 5
factorial = 1
while num > 0:
    factorial *= num
    num -= 1
print("Factorial of 5:", factorial)
```

```
0
1
2
3
4
Factorial of 5: 120
```

do-while Loop (Equivalent in Python using while and a break statement):

Executes a block of code at least once, then repeats as long as a condition is true. Useful when you need to perform an action at least once before checking the condition.

Examples

while True: num = int(input("Enter a positive number: ")) if num > 0: break print("Please enter a positive number.")

print("You entered:", num)

Nested Loops:

Loops within loops, used for processing multi-dimensional data structures.

In [22]:
```
#Example
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in matrix:
    for element in row:
        print(element, end=" ")
    print()
```

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

Choosing the Right Loop:

Use a for loop when you know the exact number of iterations or have a sequence to iterate over. Use a while loop when you need to repeat actions until a condition is met, and the number of iterations is uncertain. Use a do-while loop (equivalent using while and break) when you need to perform an action at least once before checking the condition.

In [ ]: