1. **Explain the key features of python that makes it a popular choice for programming.**

(a) **Readability and Simplicity**: Python's syntax is designed to be clear and readable, making it easier to write and understand code.

(b) **Extensive Standard Library**: Python comes with a large standard library that provides modules and packages for a wide variety of tasks, from regular expressions to networking to file I/O.

(c) **High-level Language**: Python abstracts many complex details away from the programmer, which makes it easier to focus on solving problems rather than dealing with low-level details such as memory management.

(d) **High-level Language**: Python abstracts many complex details away from the programmer, which makes it easier to focus on solving problems rather than dealing with low-level details such as memory management.

(e) **High-level Language**: Python abstracts many complex details away from the programmer, which makes it easier to focus on solving problems rather than dealing with low-level details such as memory management.

(f) **Popular in Data Science and Machine Learning**: Python's simplicity and large ecosystem have made it the language of choice for data analysis, machine learning, and artificial intelligence. Libraries like NumPy, Pandas, and TensorFlow provide powerful tools for these domains.

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

Role of predefined keywords:-

(a) **Syntax Definition**: Keywords define the structure of Python programs by indicating where and how certain operations (like conditional statements, loops, function definitions, etc.) should be used.

(b) **Language Constraints**: They enforce rules about what can and cannot be done in Python code, helping maintain consistency and clarity across different programs written in the language.

(c) **Prevented Use as Identifiers**: By reserving these words, Python prevents developers from accidentally using them as variable names or function names, which could lead to confusion or errors in code execution.

Examples of predefined keywords :

(1) if, elif, else:

```
 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")
```

(2) **for, while, break, continue**:

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

(3) **def, return**:

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

3. Compare and contrast mutable and immutable objects in python with examples.

Immutable Objects:

Definition: Immutable objects in Python are objects whose state cannot be modified after they are created. Any operation that attempts to modify the object actually creates a new object with the modified value.

Examples of Immutable Objects in Python:

1. **int**: Integer objects in Python are immutable.

```
x = 5

x += 1          # output will be 6
```

2. float: Floating-point numbers are also immutable.

```
y = 3.14
```

3. tuple: Tuples are immutable sequences of arbitrary objects.

```
tup = (1, 2, 3)
```

4. **str**: Strings are immutable sequences of characters.

```
a = "Hello"
```

Mutable Objects:

Definition: Mutable objects in Python are objects whose state can be modified after they are created. This means that their content or value can change during the execution of a program.

Examples of Mutable Objects in Python:

1. **list**: Lists are mutable sequences of objects.

```
lst = [1, 2, 3]

lst.append(4)  # Modifies the list by adding 4 to the end
```

2. **dict**: Dictionaries are mutable mappings of keys to values.

```
d = {'a': 1, 'b': 2}

d['c'] = 3  # Modifies the dictionary by adding a new key-value pair
```

3. **set**: Sets are mutable unordered collections of unique items.

```
s = {1, 2, 3}

s.add(4)  # Modifies the set by adding 4 to it
```

Comparison and Contrast:

1. Modification: Immutable objects cannot be changed after creation; any operation that modifies them actually creates a new object. Mutable objects can be modified directly without creating new objects in many cases.

2. Memory Management: Immutable objects simplify memory management because the interpreter can optimize memory allocation for them. Mutable objects may require more memory management due to potential in-place modifications.

3. Thread Safety: Immutable objects are inherently thread-safe because their state cannot be changed. Mutable objects may require synchronization mechanisms in multithreaded environments to avoid data corruption.

4. Use Cases: Immutable objects are useful for representing constant values, keys in dictionaries, and ensuring data integrity. Mutable objects are suitable for situations where objects need to be modified frequently or where sharing and modifying state is required.

5. Examples: Examples of immutable objects include integers, floats, strings, and tuples. Examples of mutable objects include lists, dictionaries, and sets.

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

In Python, operators are special symbols or keywords that perform operations on variables and values. Python supports various types of operators, which can be categorized as follows:

1. **Arithmetic Operators**

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Addition (`+`): Adds two operands.

```
a = 10
b = 5
result = a + b        # result = 15
```

Subtraction (`-`): Subtracts right operand from the left operand.

```
a = 10
b = 5
result = a - b      # result = 5
```

Multiplication (`*`): Multiplies two operands.

```
a = 10
b = 5
result = a * b       # result = 50
```

Division (`/`): Divides left operand by the right operand (always results in a float).

```
a = 10
```

```
b = 5

result = a / b       # result = 2.0
```

Floor Division (`//`): Divides left operand by the right operand and returns the floor value (integer division).

```
a = 10

b = 3

result = a // b        # result = 3
```

Modulus (`%`): Returns the remainder of the division of left operand by the right operand.

```
a = 10

b = 3

result = a % b      # result = 1
```

Exponentiation (`**`): Performs exponentiation (left operand raised to the power of right operand).

```
a = 2

b = 3

result = a ** b      # result = 8
```

2. Comparison (Relational) Operators

Comparison operators are used to compare values. They return either `True` or `False` .

Equal to (`==`): True if operands are equal.

```
a = 10

b = 5

result = (a == b)       # result = False
```

Not equal to (`!=`): True if operands are not equal.

```
a = 10

b = 5

result = (a != b)       # result = True
```

Greater than (`>`): True if left operand is greater than the right operand.

```
a = 10

b = 5

result = (a > b)        # result = True
```

Less than (`<`): True if left operand is less than the right operand.

```
a = 10

b = 5

result = (a < b)          # result = False
```

Greater than or equal to (`>=`): True if left operand is greater than or equal to the right operand.

```
a = 10

b = 5

result = (a >= b)       # result = True
```

Less than or equal to (`<=`): True if left operand is less than or equal to the right operand.

```
a = 10
b = 5
result = (a <= b)          # result = False
```

3. Logical Operators

Logical operators are used to combine conditional statements.

Logical AND (`and`): True if both operands are true.

```
a = True
b = False
result = (a and b)     # result = False
```

Logical OR (`or`): True if either of the operands is true.

```
a = True
b = False
result = (a or b)          # result = True
```

Logical NOT (`not`): True if operand is false (complements the operand).

```
a = True
result = not a         # result = False
```

4. Assignment Operators

Assignment operators are used to assign values to variables.

Assignment (`=`): Assigns right operand to left operand.

```
a = 10
```

Compound Assignment (`+=`, `-=`, `*=`, `/=`, etc.): Perform arithmetic operation on the variable and then assign the result to the variable.

```
a = 5

a += 2  # equivalent to a = a + 2,    result = 7
```

5. Bitwise Operators

Bitwise operators perform bit-level operations on integers.

Bitwise AND (`&`), Bitwise OR (`|`), Bitwise XOR (`^`): Perform bitwise operations between corresponding bits of operands.

```
a = 3  # Binary: 0011

b = 5  # Binary: 0101

result_and = a & b  # result_and = 1 (Binary: 0001)

result_or = a | b   # result_or = 7 (Binary: 0111)

result_xor = a ^ b  # result_xor = 6 (Binary: 0110)
```

Bitwise NOT (`~`): Inverts all bits of the operand.

```
a = 3  # Binary: 0011

result_not = ~a  # result_not = -4 (Binary: 1100 in 2's complement form)
```

Bitwise Left Shift (`<<`), Bitwise Right Shift (`>>`): Shift bits left or right by a specified number of positions.

```
a = 8  # Binary: 1000

result_left_shift = a << 2  # result_left_shift = 32 (Binary: 100000)

result_right_shift = a >> 1  # result_right_shift = 4 (Binary: 100)
```

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

In Python, type casting (or type conversion) refers to the process of converting a variable from one data type to another. Python provides several built-in functions for this purpose, allowing you to explicitly convert variables from one type to another as needed.

Types of Type Casting in Python

1. Implicit Type Conversion (Automatic)

Python automatically converts data types in certain situations, such as when you perform operations between different types. For example:

```
a = 10  # integer

b = 5.5  # float


c = a + b  # c will be automatically converted to float (15.5)
```

Here, `a` (integer) is implicitly converted to a `float` before performing the addition with `b`.

2. Explicit Type Conversion (Manual)

Explicit type conversion is done using built-in functions such as `int()`, `float()`, `str()`, etc., which convert a value from one type to another explicitly.

Examples of Explicit Type Casting:

Converting to Integer (`int()`):

Convert a float or string containing a number to an integer.

```
a = 10.5

b = int(a)  # b will be 10, converting float to int
```

Converting to Float (`float()`):

Convert an integer or string containing a number to a float.

```
a = 10
b = float(a)  # b will be 10.0, converting int to float
```

Converting to String (`str()`):

Convert an integer, float, or any other type to a string.

```
a = 10
b = str(a)  # b will be '10', converting int to str
```

Converting to Boolean (`bool()`):

Convert an expression or value to a boolean.

```
a = 0
b = bool(a)   # b will be False, since 0 is considered False
```

Converting to List (`list()`), Tuple (`tuple()`), Set (`set()`), etc.:

Convert sequences (like tuples, lists) or other iterable objects to different data types.

```
a = (1, 2, 3)  # tuple
b = list(a)    # b will be [1, 2, 3], converting tuple to list
```

Explicit Conversion Between Different Data Types:

For example, converting a string containing a number to an integer or float.

```
a = "123"
b = int(a)    # b will be 123
c = float(a)  # c will be 123.0
```

6.How do conditional statements works in python ?Illustrate with examples.

Conditional statements in Python allow you to execute certain blocks of code based on whether a condition evaluates to true or false. The primary conditional statements in Python are `if`, `else`, and `elif` (short for "else if").

Syntax of Conditional Statements

if condition:

    # block of code to be executed if the condition is true

elif condition:

    # block of code to be executed if the condition is true (optional)

else:

    # block of code to be executed if none of the above conditions are true (optional)

Examples of Conditional Statements

1. Basic `if` Statement

Example : Checking if a number is positive or negative

```
num = 10
if num > 0:
    print("Number is positive")
```

2. `if-else` Statement

Example : Checking if a number is positive or negative using if-else

```
num = -5
if num > 0:
    print("Number is positive")
else:
    print("Number is negative")
```

3. `if-elif-else` Statement

Example : Checking the grade based on percentage using if-elif-else

```
percentage = 75

if percentage >= 90:
    grade = 'A'
elif percentage >= 80:
    grade = 'B'
elif percentage >= 70:
    grade = 'C'
elif percentage >= 60:
    grade = 'D'
else:
    grade = 'F'

print(f"Your grade is {grade}")
```

Nested `if` Statements

You can also nest conditional statements within each other to create more complex decision-making logic:

Example : Nested if statements

```
num = 15

if num >= 0:
    if num == 0:
        print("Zero")
    else:
   print("Positive number")
else:
    print("Negative number")
```

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

In Python, loops are used to execute a block of code repeatedly as long as a specified condition is true. Python supports two main types of loops: `for` loops and `while` loops. Each type has its own use cases and can be nested within each other to create more complex iterations.

1. `for` Loop

A `for` loop is used to iterate over a sequence (such as a list, tuple, string, or any iterable object) and execute a block of code for each element in the sequence.

Syntax:

for element in sequence:

```
    # block of code to be executed for each element
```

Example: Iterating over a list

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

    print(fruit)
```

Output:

apple

banana

cherry

Example: Iterating over a String

Example: Iterating over a string

```
name = "Python"

for char in name:

    print(char)
```

Output:

P

y

t

h

o

n

2. `while` Loop

A `while` loop is used to repeatedly execute a block of code as long as a specified condition is true.

Syntax:

while condition:

```
    # block of code to be executed as long as condition is true
```

Example: Using a `while` Loop to Print Numbers

Example : Using a while loop to print numbers from 1 to 5

```
num = 1
while num <= 5:
    print(num)
    num += 1
```

Output:

1

2

3

4

5

Nested Loops

Both `for` and `while` loops can be nested within each other to perform more complex iterations.

Example: Nested `for` Loops

Example: Nested for loops to create a multiplication table

```
for i in range(1, 5):
    for j in range(1, 11):
        print(i * j, end="\t")
    print()    # to move to the next line after each inner loop completes
```

Output:

```
1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
```

Link :-

https://docs.google.com/document/d/1k34ydD_fjdZsyI5c5hAZ2NP2TpEhUsBLdyAIMNDwGeo/edit