Python Basic Assignment

Q1. Explain the key features of python that make it a popular choice for programming.

Ans. Python has gained immense popularity as a programming language due to several key features that cater to a wide range of applications. Here are some of the most notable aspects:

1. **Readability and Simplicity**: Python emphasizes code readability and simplicity, which makes it accessible for beginners while allowing experienced developers to write clear and concise code. Its use of indentation for block structures helps avoid unnecessary complexity.

2. **Versatile Paradigms**: Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming. This flexibility allows developers to choose the best approach for their project.

3. **Extensive Libraries and Frameworks**: Python boasts a vast ecosystem of libraries and frameworks (like NumPy, pandas, Flask, Django, TensorFlow, etc.) that facilitate various tasks, from web development and data analysis to machine learning and automation, significantly accelerating the development process.

4. **Cross-Platform Compatibility**: Python is cross-platform, meaning code written on one operating system can easily run on others without modification. This capability enhances portability and collaboration.

5. **Strong Community Support**: With a large and active community, Python users benefit from a wealth of resources, tutorials, and third-party packages. The community continually contributes to Python's development and improvement, making it a dynamic language.

6. **Dynamic Typing**: Python uses dynamic typing, meaning you do not need to explicitly declare variable types. This feature allows faster prototyping and iterative development.

7. **Interpreted Language**: As an interpreted language, Python executes code line by line, which simplifies debugging and testing, as developers can see immediate results from their code changes.

8. **Integration Capabilities**: Python easily integrates with other languages and technologies, making it suitable for extending functionality with tools and software that may be written in languages like C, C++, and Java.

9. **Support for Automation**: Python is commonly used for automating tasks, including scripts for file management, data scraping, and system integration, thanks to its simplicity and accessibility.

10. **Growing Popularity in Emerging Fields**: Python's popularity is rising in data science, AI, and machine learning, thanks to libraries like TensorFlow and scikit-learn. This trend keeps it relevant across multiple domains.

These features collectively contribute to Python's reputation as a versatile and powerful programming language, making it suitable for both beginners and experienced developers across various industries.

Q2. Describe the role of predefined keywords in python and provide examples of how they are used in a program.

Ans. Predefined keywords in Python, often referred to as "reserved words," are special words that have specific meanings in the Python programming language. These keywords cannot be used as identifiers (such as variable names, function names, or class names) because they are part of Python's syntax.

Role of Predefined Keywords

1. Syntax Structure: Keywords define the structure and flow of the code. They help in controlling the execution flow, defining data types, and establishing conditions.
2. Control Flow: Keywords are essential for creating control structures such as loops and conditionals, which dictate how a program executes.
3. Data Types and Functions: Keywords help define data types (like lists and dictionaries) and built-in functions, making it easier to write efficient code.
4. Object-Oriented Programming: Keywords like class and def are crucial for creating objects and functions, respectively.
5. Exception Handling: Keywords also play a critical role in error handling within Python programs.

Examples of Predefined Keywords and Their Usage

Here are some commonly used keywords and examples of how they can be utilized in a Python program:

1. if, elif, else: Used for conditional statements.

python

Copy

```
x = 10

if x > 0:

    print("Positive")
```

```
elif x < 0:
    print("Negative")
else:
    print("Zero")
```

2. for, in: Used for looping through sequences (like lists).

python

Copy

```
numbers = [1, 2, 3, 4]
for num in numbers:
    print(num)
```

3. def: Used to define a function.

python

Copy

```
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))
```

4. class: Used to define a class in object-oriented programming.

python

Copy

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

    def bark(self):
        return f"{self.name} says Woof!"

my_dog = Dog("Buddy")
print(my_dog.bark())
```

5. try, except: Used for exception handling.

python

Copy

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You can't divide by zero!")
```

6. return: Used to return a value from a function.

python

Copy

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

sum_result = add(5, 3)
print(sum_result)
```

7. import: Used to include external modules.

python

Copy

```
import math
print(math.sqrt(16))  # Outputs: 4.0
```

8. while: Used for creating while loops.

python

Copy

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

9. break, continue: Control the flow of loops.

python

Copy

```
for i in range(10):
    if i == 5:
        break
```

```
   print(i)  # Outputs: 0, 1, 2, 3, 4
```

10. with: Used to wrap the execution of a block with methods defined by a context manager.

python

Copy

```
with open('file.txt', 'r') as file:
   content = file.read()
```

Summary

Predefined keywords in Python are fundamental to writing valid code. They define the operations and logic that can be applied within Python, forming the backbone of the language's syntax and behaviour. Understanding how to use these keywords effectively is crucial for developing proficient programming skills in Python.

Q 3. Compare and contrast mutable and immutable objects in python with examples.
Ans. In Python, objects are categorized into two types based on their mutability: **mutable** and **immutable** objects. Understanding the distinction between these two types is fundamental for efficient programming and memory management in Python.

Mutable Objects

Mutable objects are those that can be modified after their creation. This means you can change their content without changing their identity (memory address).

Examples of Mutable Objects:

1. **Lists**: You can add, remove, or change elements in a list.

   ```python
   my_list = [1, 2, 3]
   print(my_list)  # Outputs: [1, 2, 3]

   my_list.append(4)
   print(my_list)  # Outputs: [1, 2, 3, 4]

   my_list[0] = 10
   print(my_list)  # Outputs: [10, 2, 3, 4]
   ```

2. **Dictionaries**: Key-value pairs can be added, altered, or removed.

   ```python
   my_dict = {'a': 1, 'b': 2}
   print(my_dict)  # Outputs: {'a': 1, 'b': 2}

   my_dict['c'] = 3
   print(my_dict)  # Outputs: {'a': 1, 'b': 2, 'c': 3}

   my_dict['a'] = 10
   print(my_dict)  # Outputs: {'a': 10, 'b': 2, 'c': 3}
   ```

3. **Sets**: You can add or remove elements from a set.

   ```python
   my_set = {1, 2, 3}
   print(my_set)  # Outputs: {1, 2, 3}

   my_set.add(4)
   print(my_set)  # Outputs: {1, 2, 3, 4}

   my_set.remove(2)
   print(my_set)  # Outputs: {1, 3, 4}
   ```

Immutable Objects

Immutable objects cannot be altered once they are created. Any modification to an immutable object results in the creation of a new object instead of changing the existing one.

Examples of Immutable Objects:

1. **Strings**: Strings cannot be changed after they are created.
   ```python
   my_string = "Hello"
   print(my_string)  # Outputs: Hello

   # Trying to modify the string
   # my_string[0] = 'h'  # This will raise a TypeError

   new_string = my_string.replace("H", "h")
   print(new_string)  # Outputs: hello
   ```

2. **Tuples**: Tuples cannot be changed or have elements added or removed.
   ```python
   my_tuple = (1, 2, 3)
   print(my_tuple)  # Outputs: (1, 2, 3)

   # Trying to modify the tuple
   # my_tuple[0] = 10  # This will raise a TypeError

   new_tuple = my_tuple + (4,)
   print(new_tuple)  # Outputs: (1, 2, 3, 4)
   ```

3. **Frozensets**: Similar to sets, but they are immutable.
   ```python
   my_frozenset = frozenset([1, 2, 3])
   print(my_frozenset)  # Outputs: frozenset({1, 2, 3})

   # Trying to alter the frozenset
   # my_frozenset.add(4)  # This will raise an AttributeError
   ```

```

### Comparison Summary

| Feature | Mutable Objects | Immutable Objects |
|---|---|---|
| **Modification** | Can be changed in place | Cannot be modified after creation |
| **New Object** | No new object created on change | New object created on change |
| **Examples** | Lists, Dictionaries, Sets | Strings, Tuples, Frozensets |
| **Performance** | May have performance overhead for resizing and copying | Generally faster due to fixed size |
| **Use Cases** | Suitable for collections that require frequent updates | Suitable for fixed collections, keys in dictionaries |

### Usage Considerations

1. **Data Integrity**: Use immutable objects when you want to ensure that the data does not change, thus preventing accidental modifications.

2. **Performance**: For datasets that need frequent updates, mutable objects are generally more efficient.

3. **Function Arguments**: Be cautious with mutable objects when passing them to functions, as changes within the function can affect the original data. Immutable objects, on the other hand, are safe from such mutations.

Understanding mutability is essential to effectively use Python and optimize data handling in your applications.

Q 4. Discuss the different types of operators in python and provide examples of how they are used.
Ans. In Python, operators are special symbols that indicate the operations to be performed on operands. They can be classified into several categories based on their functionality. Here are the main types of operators in Python along with examples of how they are used:

### 1. **Arithmetic Operators**

These operators are used to perform mathematical operations.
```

- **Addition (`+`)**: Adds two operands.

- **Subtraction (`-`)**: Subtracts the second operand from the first.

- **Multiplication (`*`)**: Multiplies two operands.

- **Division (`/`)**: Divides the first operand by the second, returning a float.

- **Floor Division (`//`)**: Divides and returns the largest integer less than or equal to the result.

- **Modulus (`%`)**: Returns the remainder of the division.

- **Exponentiation (`**`)**: Raises the first operand to the power of the second.

Example:

```python
a = 10

b = 3

print(a + b)   # Outputs: 13

print(a - b)   # Outputs: 7

print(a * b)   # Outputs: 30

print(a / b)   # Outputs: 3.33333...

print(a // b)  # Outputs: 3

print(a % b)   # Outputs: 1

print(a ** b)  # Outputs: 1000
```

2. **Comparison Operators**

These operators compare two values and return a Boolean result (`True` or `False`).

- **Equal to (`==`)**: Checks if two values are equal.

- **Not equal to (`!=`)**: Checks if two values are not equal.

- **Greater than (`>`)**: Checks if the left operand is greater than the right.

- **Less than (`<`)**: Checks if the left operand is less than the right.

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

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

Example:

```python
x = 5
y = 10
print(x == y)   # Outputs: False
print(x != y)   # Outputs: True
print(x > y)    # Outputs: False
print(x < y)    # Outputs: True
print(x >= 5)   # Outputs: True
print(x <= 10)  # Outputs: True
```

3. **Logical Operators**

These operators are used to combine conditional statements.

- **AND (`and`)**: Returns `True` if both operands are true.
- **OR (`or`)**: Returns `True` if at least one of the operands is true.
- **NOT (`not`)**: Returns `True` if the operand is false.

Example:

```python
a = True
b = False
print(a and b)  # Outputs: False
print(a or b)   # Outputs: True
print(not a)    # Outputs: False
```

4. **Assignment Operators**

These operators are used to assign values to variables. They often incorporate arithmetic operations.

- **Assign (`=`)**: Assigns a value to a variable.
- **Add and assign (`+=`)**: Adds and assigns the result.
- **Subtract and assign (`-=`)**: Subtracts and assigns the result.
- **Multiply and assign (`*=`)**: Multiplies and assigns the result.
- **Divide and assign (`/=`)**: Divides and assigns the result.
- **Floor divide and assign (`//=`)**: Floor divides and assigns.
- **Modulus and assign (`%=`)**: Applies modulus and assigns.
- **Exponent and assign (`**=`)**: Raises and assigns.

Example:
```python
x = 5
x += 2  # Equivalent to x = x + 2
print(x)  # Outputs: 7
x *= 3  # Equivalent to x = x * 3
print(x)  # Outputs: 21
```

5. **Bitwise Operators**
Bitwise operators act on bits and perform bit-level operations.

- **AND (`&`)**: Performs a bitwise AND.
- **OR (`|`)**: Performs a bitwise OR.
- **XOR (`^`)**: Performs a bitwise XOR.
- **NOT (`~`)**: Inverts all bits.
- **Left Shift (`<<`)**: Shifts bits to the left.
- **Right Shift (`>>`)**: Shifts bits to the right.

Example:
```python
a = 5  # In binary: 0101
```

```
b = 3  # In binary: 0011
print(a & b)  # Outputs: 1 (0001)
print(a | b)  # Outputs: 7 (0111)
print(a ^ b)  # Outputs: 6 (0110)
print(~a)     # Outputs: -6 (inverts bits)
print(a << 1) # Outputs: 10 (1010)
print(a >> 1) # Outputs: 2 (0010)
```

6. **Membership Operators**

These operators test for membership in sequences such as strings, lists, or tuples.

- **In (`in`)**: Returns `True` if a value is found in the sequence.
- **Not in (`not in`)**: Returns `True` if a value is not found in the sequence.

Example:

```python
my_list = [1, 2, 3, 4]
print(2 in my_list)       # Outputs: True
print(5 not in my_list)   # Outputs: True
```

7. **Identity Operators**

These operators compare the memory locations of two objects.

- **Is (`is`)**: Returns `True` if both operands refer to the same object.
- **Is not (`is not`)**: Returns `True` if both operands do not refer to the same object.

Example:

```python
x = [1, 2, 3]
```

```
y = x
z = [1, 2, 3]
print(x is y)      # Outputs: True (same object)
print(x is z)      # Outputs: False (different objects with same content)
print(x == z)      # Outputs: True (values are the same, but they are different objects)
```

Summary

Python's operators provide a rich set of functionality to manipulate data. Understanding how these operators work helps you write more effective and efficient code. Each type of operator serves a different purpose, allowing for a wide range of operations to be performed across various data types.

Q 5. Explain the concept of type casting in python with examples.
Ans. Type casting in Python refers to the conversion of one data type into another. It allows you to change the type of a variable or value explicitly.

Examples:

1. Integer to Float:

python

Copy

```
num = 5
num_float = float(num)  # Converts integer to float
print(num_float)  # Outputs: 5.0
```

2. Float to Integer:

python

Copy

```
num = 5.7
num_int = int(num)  # Converts float to integer, truncating the decimal
print(num_int)  # Outputs: 5
```

3. String to Integer:

python

Copy

```
num_str = "10"
```

num_int = int(num_str) # Converts string to integer

print(num_int) # Outputs: 10

4. Integer to String:

python

Copy

num = 20

num_str = str(num) # Converts integer to string

print(num_str) # Outputs: '20'

Type casting is useful for ensuring the correct data type is used in operations and functions.

Q 6. How do conditional statements work in python? Illustrate with examples.
Ans. Conditional statements in Python allow you to execute certain blocks of code based on specific conditions. The main conditional statements are if, elif, and else.

Structure:

- if: Evaluates a condition; if true, executes the block of code.
- elif: (optional) Checks another condition if the previous if was false.
- else: (optional) Executes if all previous conditions were false.

Examples:

1. Basic if Statement:

python

Copy

```
x = 10

if x > 5:

    print("x is greater than 5")  # Outputs: x is greater than 5
```

2. Using if, elif, and else:

python

Copy

```
x = 5

if x > 5:

    print("x is greater than 5")

elif x == 5:

    print("x is equal to 5")  # Outputs: x is equal to 5
```

```
else:
    print("x is less than 5")
```

3. Multiple Conditions:

```
python
Copy
x = 0
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")  # Outputs: Zero
```

Conditional statements enable decision-making in your code based on dynamic conditions.

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

Ans- In Python, there are primarily two types of loops: for loops and while loops. Each serves different purposes depending on the use case.

1. for Loop

The for loop iterates over a sequence (like a list, tuple, or string) or a range of numbers.

Use Case: When the number of iterations is known in advance or you're iterating over a collection.

Example:

```
python
Copy
# Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
# Outputs:
# apple
# banana
# cherry
# Using range()
for i in range(5):
    print(i)
# Outputs: 0 1 2 3 4
```

2. while Loop

The while loop continues executing as long as a specified condition is true.

Use Case: When the number of iterations is not known and depends on a dynamic condition.

Example:

python

Copy

```
count = 0
while count < 5:
    print(count)
    count += 1
# Outputs: 0 1 2 3 4
```

Summary

- for loops are used for iterating over a sequence or a predefined number of iterations.
- while loops are used when the termination condition is dynamic and not known beforehand.
