Introduction to Python:

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 supports multiple programming paradigms, including procedural, object-oriented, and functional programming.
- It has a large and active community that contributes to its rich ecosystem of libraries and frameworks.

Syntax:

- Python uses indentation for block structure and does not require explicit braces or semicolons.
- It emphasizes code readability with its clean and intuitive syntax.
- Statements are typically written on separate lines, and indentation is used to define code blocks.

Complexity:

- Python is designed to prioritize code readability and developer productivity over execution speed.
  - It is an interpreted language, which means it is generally slower than compiled languages like Java.
  - However, Python offers various optimization techniques, such as using built-in functions and libraries, to improve performance.

Examples:

- Python is widely used in various domains, including web development, data science, machine learning, and scripting.
  - Some popular libraries and frameworks in Python include Django (web development), NumPy (scientific computing), Pandas (data manipulation), and TensorFlow (machine learning).

Code Structure:

- Python programs are organised into modules, which are files containing Python code.
- Modules can be imported and used in other modules to reuse code.
- The entry point of a Python program is typically a module called `main.py`, which contains the main execution logic.

Example Code and Explanations:

- Here's an example of a simple Python program that prints "Hello, World!" to the console:

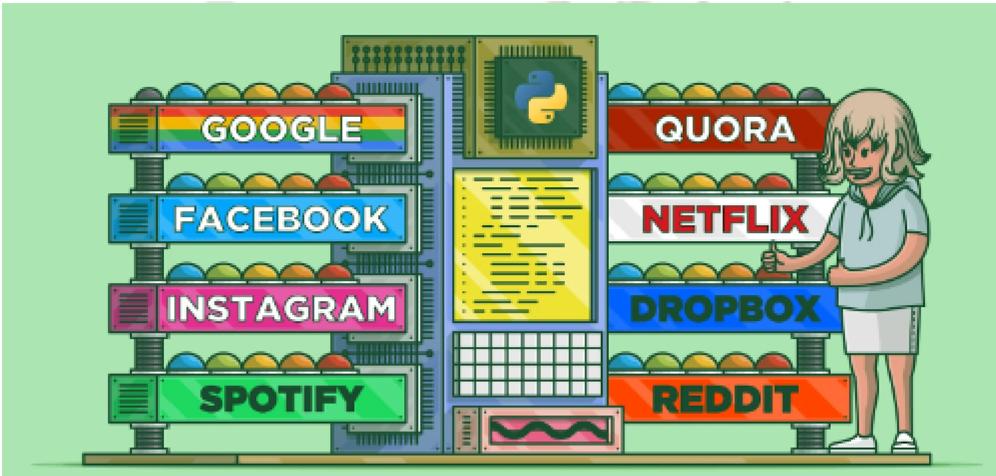
```
def say_hello():
    print("Hello, World!")

if __name__ == "__main__":
    say_hello()
```

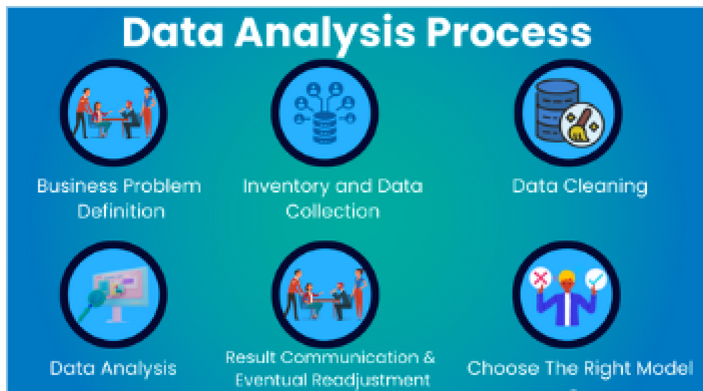
- In this code, we define a function `say_hello()` that prints the greeting. The `if __name__ == "__main__"` condition ensures that the `say_hello()` function is only called when the script is executed directly.

Real-Life Examples:

- Python is widely used in the tech industry and has been used to develop popular applications and platforms such as Instagram, Spotify, and YouTube.



- It is also used extensively in scientific research, data analysis, and machine learning projects.



Interview Questions:

- **Easy:**
  1. What is the difference between Python 2 and Python 3?
  2. How do you write a for loop in Python?
  3. What are the built-in data types in Python?
  - **Hard:**
    1. Explain the Global Interpreter Lock (GIL) in Python.
    2. What are decorators in Python, and how do they work?

Complexity Table:

- Here's a complexity table for some common operations in Python:

Operation	Time Complexity
List indexing	O(1)
List concatenation	O(k)
Dictionary lookup	O(1)
Dictionary insertion	O(1)
Set insertion	O(1)
Set union/intersection	O(len(set))
String concatenation	O(n)
Sorting	O(n log n)

VALUE LADEN

Comments in Python

Introduction:

In Python, comments are used to add explanatory notes or annotations within the code. They are ignored by the interpreter and are solely intended for human readers to understand the code better. Comments provide context, improve code readability, and can serve as documentation for future developers.

Syntax:

In Python, there are two ways to write comments:

1. **Single-line comments:** Single-line comments begin with the hash character (#) and continue until the end of the line. Anything after the # symbol is considered a comment.

   Example:

```
# This is a single-line comment
  print("Hello, World!")
```

2. **Multi-line comments:** Multi-line comments are enclosed within triple quotes (''' ''') or double quotes (""" """). These comments can span multiple lines and are often used for longer explanations or documentation.

   Example:

```
This is a multi-line comment.
   It can span across multiple lines.
   print("Hello, World!")
```

Complexity:

The complexity of comments in Python is constant (O(1)) because the time required to interpret or execute the code remains unaffected by the presence or absence of comments. Comments are not executed by the interpreter and do not impact the performance of the code.

Examples:

1. Single-line comment example:

```
# Calculate the sum of two numbers
a = 5
b = 10
sum = a + b
print("The sum is:", sum)
```

2. Multi-line comment example:

''' This program calculates the factorial of a number. The factorial is calculated using a recursive function. '''

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print("The factorial of 5 is:", result)
```

Code Structure:

Comments should be placed strategically to provide clarity and understanding. They can be added before or after a line of code or used to explain a particular block of code. It is essential to maintain a balance and avoid excessive or redundant commenting.

Example Code and Explanations:

Let's consider an example to illustrate the use of comments in Python:

```
# Calculate the area of a rectangle

# Input the dimensions
length = 5  # length of the rectangle
width = 10  # width of the rectangle

# Calculate the area
area = length * width

# Output the result
print("The area of the rectangle is:", area)
```

In the above code, comments are used to provide explanatory notes at various stages. They clarify the purpose of variables, describe the steps involved, and make the code more understandable.

Real-Life Examples:

1. Collaborative coding: When working in a team, comments help other developers understand the codebase quickly and make it easier to collaborate effectively.
2. Documentation: Comments can be used to generate code documentation automatically. Tools like Sphinx can extract comments and generate comprehensive documentation for projects.
3. Bug fixing: Comments can be added to mark potential issues or provide hints for debugging when encountering complex code.

Interview Questions:

Easy:

1. What are comments in Python, and why are they important?
2. How do you write a single-line comment in Python?
3. How can comments improve the readability and maintainability of code?

Hard:

1. Can you extract comments from Python code using external tools or libraries? Explain how.
2. In what scenarios would you consider removing or rewriting comments in a codebase? Provide examples and reasons.

Statements in Python

Introduction:

In Python, statements are the building blocks of code that carry out specific actions or operations. They are used to control the flow of execution, make decisions, and perform repetitive tasks. Python provides several types of statements, including assignment statements, control flow statements, and function definitions. Understanding how to use statements effectively is crucial for writing functional and efficient Python programs.

Syntax:

The syntax of statements in Python generally follows a specific structure. Most statements begin with a keyword, followed by any necessary arguments or expressions, and end with a colon (:). Here's a general format:

```
keyword arguments/expression:
    # Indented block of code
```

Complexity:

The complexity of statements in Python depends on the type of statement and the operations performed within it. Some statements, such as assignment statements and simple control flow statements, have constant time complexity (O(1)) as they perform basic operations. However, more complex statements, like loops and recursive functions, can have higher time complexity depending on the number of iterations or recursive calls.

Examples:

Let's explore some common types of statements in Python:

1. Assignment Statement:

The assignment statement assigns a value to a variable.

```
x = 10
```

2. If Statement:

   The if statement is used to execute a block of code only if a specified condition is true.

```
if x > 5:
    print("x is greater than 5")
```

3. For Loop:

   The for loop statement iterates over a sequence (such as a list or string) and executes a block of code for each item.

```
numbers = [1, 2, 3, 4, 5]
   for num in numbers:
       print(num)
```

Code Structure:

Python code is structured using indentation. Statements within the same block must be indented at the same level. Blocks are typically defined by colons (:), followed by indented lines. Here's an example:

```
if condition:
    # Block of code
    statement1
    statement2

else:
    # Block of code
    statement3
    statement4
```

Example Code and Explanations:

Let's consider a simple example that combines multiple types of statements to calculate the sum of all even numbers from 1 to 10.

```
even_sum = 0
for num in range(1, 11):
    if num % 2 == 0:
        even_sum += num

print("Sum of even numbers:", even_sum)
```

Explanation:

- We initialise the variable `even_sum` to 0 to store the sum of even numbers.
- The `for` loop iterates over the numbers 1 to 10.
- Inside the loop, the `if` statement checks if the current number (`num`) is divisible by 2 (i.e., even).
- If the number is even, it adds it to the `even_sum` variable using the `+=` assignment operator.
- Finally, outside the loop, we print the value of `even_sum`, which gives us the desired result.

Real-Life Examples:

1. Calculating the total price of items in a shopping cart.
2. Parsing and processing data from a CSV file.
3. Implementing a game loop for a simple game.
4. Performing sentiment analysis on a large dataset of customer reviews.
5. Creating a web crawler to extract information from websites.

Interview Questions Asked:

Here are three easy and two hard interview questions related to Python statements:

Easy:

1. What is the purpose of the `if` statement in Python?
2. How does the `for` loop differ from the `while` loop in Python?
3. Explain the role of the `else` statement in an `if-else` construct.

Hard:

1. Can you provide an example of nested `if` statements in Python? Why might you need to use them?
2. Compare and contrast the `range()` function and the `enumerate()` function in Python. How are they used in different contexts?

Complexity Table:

Here's a complexity table for some commonly used statements in Python:

Statement	Time Complexity (Average Case)
Assignment	O(1)
If-else	O(1)
For Loop	O(n)
While Loop	O(n)
Nested Statements	Varies
Function Definition	O(1)

Note: The time complexity can vary based on the specific operations performed within the statements and the size of the input data.

Variables and Identifiers

Introduction:

Variables and identifiers are fundamental concepts in programming languages. They are used to represent and store data in memory for manipulation and processing. In this context, a variable is a named storage location that holds a value, while an identifier is the name given to a variable, function, class, or other programming entity.

Syntax:

In most programming languages, identifiers follow specific rules and conventions. Some common syntax rules for identifiers include:

1. The first character must be a letter or an underscore (_).
2. Subsequent characters can be letters, numbers, or underscores.
3. Identifiers are case-sensitive, meaning "myVariable" and "myvariable" are treated as different identifiers.

Complexity:

The complexity of variables and identifiers lies in understanding their scope and visibility within a program. Variables have different scopes, such as global scope (accessible throughout the program) and local scope (limited to a specific block of code). Identifiers must be chosen carefully to ensure they are meaningful and avoid conflicts with reserved keywords or existing identifiers.

Examples:

Here are some examples of valid identifiers:

1. age
2. my_variable
3. firstName
4. num123
5. _privateVariable

And here are some examples of invalid identifiers:

1. 123 num (starts with a number)

2. my-variable (contains a hyphen)

3. class (a reserved keyword in many programming languages)

Code Structure:

A typical code structure involving variables and identifiers may include variable declaration, initialization, and usage. Here's an example code snippet in Python:

```
# Variable declaration and initialization
name = "John"
age = 25

# Variable usage
print("Name:", name)
print("Age:", age)
```

Explanation:

In the above code, two variables, `name` and `age`, are declared and initialised. The variable `name` is assigned the value "John," and the variable `age` is assigned the value 25. The `print` statements then output the values of these variables.

Real-Life Examples:

Variables and identifiers are extensively used in real-life programming scenarios. Some examples include:

1. Storing user input: In a web application, an identifier can be used to store a user's name or email address for further processing.

2. Mathematical calculations: A variable can be used to store the result of a mathematical operation, such as the sum of two numbers.

3. Data manipulation: Identifiers can represent data structures like arrays or lists, allowing programmers to access and modify their contents easily.

Interview Questions:

Here are three easy and two hard interview questions related to variables and identifiers:

Easy:

1. What is the difference between a global variable and a local variable?

2. What is the purpose of a variable declaration?

3. How do you choose meaningful identifiers for variables?

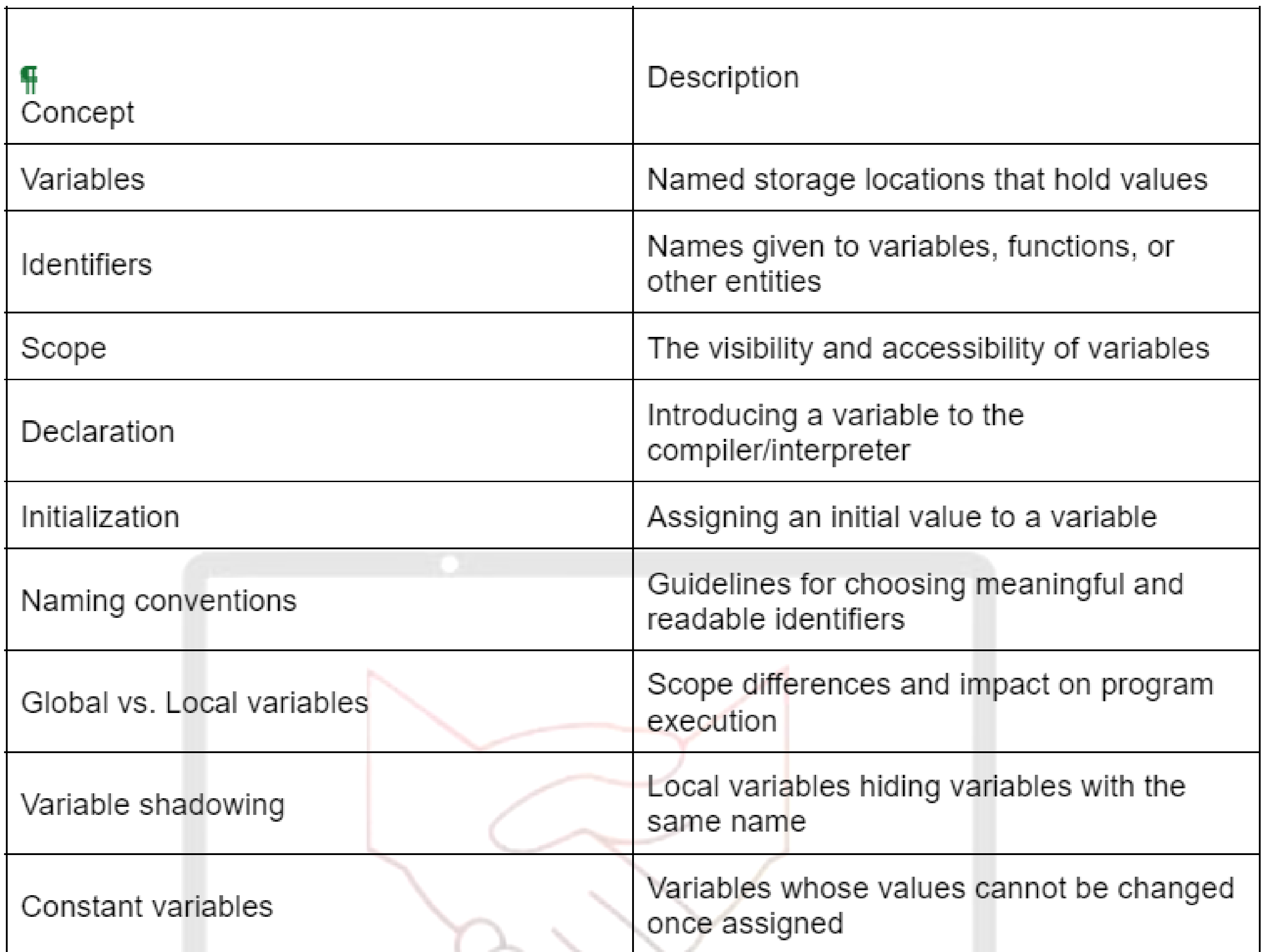
Hard:

1. Explain the concept of variable shadowing.

2. What is a constant variable, and how is it different from a regular variable?

Complexity Table:

Here's a complexity table to summarise the concepts discussed:



¶ Concept	Description
Variables	Named storage locations that hold values
Identifiers	Names given to variables, functions, or other entities
Scope	The visibility and accessibility of variables
Declaration	Introducing a variable to the compiler/interpreter
Initialization	Assigning an initial value to a variable
Naming conventions	Guidelines for choosing meaningful and readable identifiers
Global vs. Local variables	Scope differences and impact on program execution
Variable shadowing	Local variables hiding variables with the same name
Constant variables	Variables whose values cannot be changed once assigned

Input/Output formatting

Introduction

The process of specifying the structure and presentation of data when it is being inputted or outputted in a program. It involves defining the syntax and rules for how data should be formatted and displayed to the user or other components of the program. Proper input/output formatting is important for ensuring data integrity, readability, and interoperability between different systems or components.

Syntax:

The syntax for input/output formatting depends on the programming language or framework being used. Most programming languages provide built-in functions or libraries that handle input/output operations and provide formatting options. For example, in Python, the `print` function can be used to output formatted text, and the `input` function can be used to accept user input. These functions often support various formatting options, such as specifying the width, precision, alignment, and type of data being formatted.

Complexity:

The complexity of input/output formatting can vary depending on the requirements of the program and the complexity of the data being processed. Simple formatting, such as printing a single string or number, is relatively straightforward. However, as the complexity of the data and formatting requirements increase, the complexity of the formatting code also increases. This includes handling different data types, formatting tables or structured data, handling user input validation, and managing error conditions.

Examples:

Here are a few examples to illustrate different aspects of input/output formatting:

1. Basic output formatting in Python:

```
name = "John"
age = 25
print("Name: {}, Age: {}".format(name, age))
```

Output: **`Name: John, Age: 25`**

2. Formatting floating-point numbers in Java:

```
double pi = 3.14159;
System.out.printf("The value of pi is approximately %.2f", pi);
```

Output: **`The value of pi is approximately 3.14`**

3. User input formatting and validation in C++:

```
#include <iostream>
#include <limits>

int main() {
    int age;
    std::cout << "Enter your age: ";
    while (!(std::cin >> age)) {
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << "Invalid input. Enter your age: ";
    }
    std::cout << "Your age is: " << age << std::endl;
    return 0;
}
```

Code Structure:

The code structure for input/output formatting typically involves a combination of input operations (e.g., reading user input or data from files) and output operations (e.g., displaying data to the user or writing data to files). The formatting rules and options are applied during these operations to ensure the desired output format. This can include using format specifiers, escape sequences, conditional statements, loops, and other programming constructs to manipulate and format the data.

Example Code and Explanations:

Let's consider a Python example that demonstrates various aspects of input/output formatting:

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
height = float(input("Enter your height in meters: "))
```

```
print("Name: {}, Age: {}, Height: {:.2f} meters".format(name, age, height))
```

Explanation:

1. The `input` function is used to accept user input for the name, age, and height variables.
2. The `int` and `float` functions are used to convert the input strings to the appropriate data types (integer and floating-point number, respectively).
3. The `print` function is used to display the formatted output. The curly braces `{}` serve as placeholders for the values to be inserted, and the `format` method is used to substitute the actual values into the placeholders. The `:.2f` inside the curly braces is a format specifier that limits the floating-point number to two decimal places.

Real-Life Examples:

Input/output formatting is used in various real-life scenarios, such as:

1. User interfaces: Formatting user inputs and outputs in graphical user interfaces (GUIs) or command-line interfaces (CLIs).
2. Data serialisation: Formatting data for storage or transmission, such as converting objects or structured data into a specific file format (e.g., JSON, XML, CSV).
3. Reporting: Formatting data into readable reports or documents, including tables, charts, or graphs.
4. Web development: Formatting data for web pages, such as generating HTML tables or formatting data for API responses.
5. File parsing: Parsing and formatting data from files with specific formats, such as log files, configuration files, or data interchange formats.

Interview Questions (3 easy and 2 hard):

Here are some interview questions related to input/output formatting:

Easy:

1. How would you format a floating-point number to display two decimal places in Python?
2. Explain the importance of input validation in user input formatting.
3. How can you align text to the right side when printing in C++?

Hard:

1. Describe a scenario where you would use regular expressions for input/output formatting.
2. Explain how you would handle complex data structures, such as nested dictionaries or lists, during output formatting.

Complexity Table:

Here's a complexity table to summarise the complexity of input/output formatting operations:

Operation	Complexity
Simple text output	O(1)
Formatting numbers	O(1)
Table formatting	O(n^2) or higher
Input validation	O(n)
Handling complex data structures	O(n^2) or higher (depends on data complexity)

Note: The complexity can vary based on the specific implementation and the size of the data being processed. The above table provides a general estimation.

Operators

Operators are fundamental components in programming languages that allow us to perform various operations on data, such as arithmetic calculations, logical evaluations, and comparisons. They manipulate operands to produce a desired result.

Syntax:

Operators are used in expressions and follow a specific syntax depending on the programming language. Generally, an operator is placed between two operands. For example, in the expression "a + b," the "+" symbol is the operator, and "a" and "b" are the operands.

Complexity:

The complexity of operators can vary depending on their functionality and the data types they operate on. Some operators have a higher complexity than others, especially when dealing with complex data structures or algorithms. It's important to understand the complexity of operators to optimise the performance of your code.

Examples:

Here are some common types of operators:

1. Arithmetic Operators:
   - Addition (+)
   - Subtraction (-)
   - Multiplication (*)
   - Division (/)
   - Modulo (%)
   - Increment (++)
   - Decrement (--)
2. Assignment Operators:
   - Simple assignment (=)
   - Addition assignment (+=)
   - Subtraction assignment (-=)

- Multiplication assignment (*=)
- Division assignment (/=)

3. Comparison Operators:
    - Equal to (==)
    - Not equal to (!=)
    - Greater than (>)
    - Less than (<)
    - Greater than or equal to (>=)
    - Less than or equal to (<=)
4. Logical Operators:
    - AND (&&)
    - OR (||)
    - NOT (!)
5. Bitwise Operators:
    - AND (&)
    - OR (|)
    - XOR (^)
    - Bitwise NOT (~)
    - Left shift (<<)
    - Right shift (>>)

Code Structure:

Operators are used within expressions to perform specific operations. An expression can consist of one or more operands and one or more operators. The operators are applied to the operands according to the rules defined by the programming language.

Example Code and Explanations:

Let's consider a simple example in Python to illustrate the usage of operators:

```
# Arithmetic Operators
a = 5
b = 2

sum = a + b
difference = a - b
product = a * b
quotient = a / b
remainder = a % b

print("Sum:", sum)
print("Difference:", difference)
```

```
print("Product:", product)
print("Quotient:", quotient)
print("Remainder:", remainder)
```

Output:

```
Sum: 7
Difference: 3
Product: 10
Quotient: 2.5
Remainder: 1
```

In this example, we use arithmetic operators to perform addition, subtraction, multiplication, division, and modulo operations on two operands (`a` and `b`). The results are then printed.

Real-Life Examples:

Operators are used extensively in programming for various real-life scenarios. Here are a few examples:

1. Banking Application:

In a banking application, arithmetic operators are used to calculate interest rates, perform financial calculations, and validate transactions.

2. Gaming:

Logical operators are used in game development to control game mechanics, implement decision-making algorithms, and evaluate game conditions.

3. Search Engine:

Comparison operators are used in search engines to compare search queries with indexed content, helping to retrieve relevant results.

Interview Questions Asked (3 easy and 2 hard):

Easy:

1. What are the different types of operators?
2. Explain the difference between the "==" and "===" operators.
3. How do logical operators work? Provide an example.

Hard:

1. Explain the concept of operator overloading and provide an example.
2. What is the difference between prefix and postfix increments/decrements in programming languages?

Complexity Table:

Here's a complexity table for some common operators:

Operator	Complexity
Arithmetic	O(1)
Assignment	O(1)
Comparison	O(1)
Logical	O(1)
Bitwise	O(1)

The complexity of most operators is constant (O(1)) since they operate on individual values or a fixed number of bits. However, the overall complexity of an expression depends on the combination of operators and operands used.

Data Types and Type Conversions

Introduction:

Data types are fundamental to programming languages as they define the type of data that can be stored in a variable or manipulated in a program. Each programming language has its own set of data types, but common types include integers, floating-point numbers, characters, booleans, and strings. Type conversions, also known as type casting or type coercion, allow you to change the data type of a value from one type to another.

Syntax:

The syntax for declaring variables with specific data types varies across programming languages. Here's a general representation:

```
<datatype> <variable_name> = <value>;
```

For example, in Python, you can declare an integer variable as follows:

```
age = 25
```

Complexity:

The complexity of data types and type conversions depends on the programming language and the specific operations being performed. In general, the complexity of type conversions is minimal since they involve converting one data type to another.

Examples:

Here are a few examples of data types and type conversions in different programming languages:

1. Python:

```
# Integer to string conversion
age = 25
age_str = str(age)

# String to integer conversion
number_str = "10"
number = int(number_str)

# Floating-point to integer conversion
pi = 3.14
pi_int = int(pi)
```

2. JavaScript:

```
// String to integer conversion
var number_str = "10";
var number = parseInt(number_str);

// Integer to string conversion
var age = 25;
var age_str = age.toString();

// Floating-point to integer conversion
var pi = 3.14;
var pi_int = Math.floor(pi);
```

Code Structure:

The code structure for data types and type conversions generally follows a similar pattern across programming languages:

1. Declare variables with specific data types.
2. Perform type conversions using appropriate functions or operators.
3. Assign the converted value to a new variable if necessary.

Example Code and Explanations:

Let's take a closer look at the Python example code provided earlier:

```
# Integer to string conversion
age = 25
age_str = str(age)
```

In this example, the variable `age` is declared as an integer with a value of 25. To convert it to a string, the `str()` function is used, and the result is stored in the `age_str` variable.

```
# String to integer conversion
number_str = "10"
number = int(number_str)
```

Here, the variable **number_str** is initialised as a string with the value "10". To convert it to an integer, the **int()** function is used, and the converted value is stored in the **number** variable.

```
# Floating-point to integer conversion
pi = 3.14
pi_int = int(pi)
```

In this example, the variable **pi** is declared as a floating-point number with the value 3.14. To convert it to an integer, the **int()** function is used, and the converted value is stored in the **pi_int** variable.

Real-Life Examples:

Data types and type conversions are essential in real-life programming scenarios. Here are a few examples:

1. User Input: When accepting input from users, it often needs to be converted to the desired data type. For instance, converting a user's age input from a string to an integer.
2. Database Operations: When retrieving data from a database, it might be in a specific format or data type. Conversions are often required to match the data type used in the program.
3. Mathematical Calculations: Type conversions are frequently used in mathematical calculations to ensure compatible data types are used.

For example, converting a floating-point number to an integer before performing arithmetic operations.

Interview Questions Asked (3 Easy and 2 Hard):

Easy:

1. What is the difference between a string and an integer?
2. How do you convert a string to an integer in Python?
3. Explain the concept of type conversions and why they are important in programming.

Hard:

1. Describe the process of type casting in C++ and provide an example.
2. What is implicit type conversion? Give an example of its usage and potential risks.

Complexity Table:

Below is a complexity table that provides an overview of the complexity of common data types and type conversions. The complexity is represented using big O notation.

Operation	Complexity
Integer Addition	O(1)
Integer Division	O(1)
String Concatenation	O(n)
String to Integer Conversion	O(n)
Integer to String Conversion	O(n)
Floating-point to Integer Conversion	O(1)
Character to ASCII Conversion	O(1)

VALUE LADEN Dotting The Best

Note: The complexity can vary depending on the programming language and specific implementation.

Conditional Statements - if, if else, elif, nested if

Introduction:

Conditional statements are fundamental constructs in programming that allow the execution of different code blocks based on certain conditions. They help control the flow of a program by evaluating expressions and executing specific blocks of code accordingly. In this context, we will discuss four types of conditional statements: if, if-else, elif, and nested if.

Syntax:

The syntax for each type of conditional statement is as follows:

1. if statement:

```
if condition:
    # code block
```

2. if-else statement:

```
if condition:
    # code block executed if condition is True
else:
    # code block executed if condition is False
```

3. elif statement (short for "else if"):

```
if condition1:
    # code block executed if condition1 is True
```

```
elif condition2:
    # code block executed if condition1 is False and condition2 is True
else:
    # code block executed if both condition1 and condition2 are False
```

4. Nested if statement:

```
if condition1:
    # code block executed if condition1 is True
    if condition2:
        # code block executed if both condition1 and condition2 are True
    else:
        # code block executed if condition1 is True and condition2 is False
else:
    # code block executed if condition1 is False
```

Complexity:

The complexity of conditional statements is generally considered to be constant time complexity (O(1)), as the execution time does not depend on the size of the input. However, the complexity can increase if there are multiple conditions or nested if statements.

Examples:

1. if statement:

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

2. if-else statement:

```
x = -5
if x > 0:
    print("x is positive")
else:
    print("x is non-positive")
```

3. elif statement:

```
x = 0
if x > 0:
    print("x is positive")
```

```
elif x < 0:
    print("x is negative")
else:
    print("x is zero")
```

4. Nested if statement:

```
x = 5
if x > 0:
    if x % 2 == 0:
        print("x is positive and even")
    else:
        print("x is positive and odd")
else:
    print("x is non-positive")
```

Code structure:

The code structure of conditional statements follows a hierarchical approach. Each condition is evaluated sequentially, and the corresponding code block is executed if the condition evaluates to True. If none of the conditions are True (in the case of if-elif-else), the else block is executed.

Example code and explanations:

Let's consider an example to demonstrate the use of conditional statements:

```
# Example: Check if a number is positive, negative, or zero

num = float(input("Enter a number: "))

if num > 0:
    print("The number is positive.")
elif num < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

Explanation:

1. The user is prompted to enter a number.
2. The input is converted to a float using the `float()` function.
3. The if-elif-else statement is used to determine if the number is positive, negative, or zero.
4. If the number is greater than 0, the first if condition evaluates to True, and the corresponding print statement executes.
5. If the number is less than 0, the first condition is False, and the elif condition is evaluated. If it is True, the corresponding print statement executes.

6. If both the if and elif conditions are False, the else block is executed, which prints that the number is zero.

Real-Life Examples:

1. Weather Application:
    - If the temperature is above 30 degrees Celsius, display a message for a hot day.
- If the temperature is between 20 and 30 degrees Celsius, display a message for a pleasant day.
- If the temperature is below 20 degrees Celsius, display a message for a cool day.

2. Online Shopping:
- If the total order amount is greater than $100, apply a 10% discount.
- If the total order amount is between $50and$100, apply a 5% discount.
- If the total order amount is less than $50, no discount is applied.

Interview Questions Asked (3 easy and 2 hard):

Easy Questions:

1. Write a Python program to check if a given number is even or odd using an if statement.
2. Write a Python program to find the maximum of three numbers using if-else statements.
3. Write a Python program to check if a given year is a leap year or not using if-else statements.

Hard Questions:

1. Write a Python program to check if a given string is a palindrome using nested if statements.
2. Write a Python program to simulate a simple calculator that performs addition, subtraction, multiplication, or division based on user input using if-elif-else statements.

Complexity table:

Type of Conditional Statement	Time Complexity (Worst Case)
if	O(1)
if-else	O(1)
elif	O(1)
nested if	O(1)

The time complexity of each conditional statement is considered constant because the execution time is not dependent on the size of the input or the number of conditions. Each condition is evaluated sequentially until a True condition is found.

In summary, conditional statements provide a powerful tool for controlling the flow of a program based on different conditions. They can be used to handle various scenarios and make decisions dynamically, leading to more flexible and interactive programs.