Artificial Intelligence Diploma

Python Session 3

NeuroTech

Agenda

1. Comparison Operator
2. Logical Operator
3. Control Statements
4. Conditional Statements
5. Looping Statements
6. Control Transfer statements
7. Exception Handling

Operators

Comparison Operators

Comparison operators in Python are used to compare values and determine the relationship between them. They return Boolean values (True or False) based on whether the comparison is true or false.

Operator	Description
Equal (==)	Checks if two values are equal.
Not Equal (!=):	Checks if two values are not equal.
Greater Than (>)	Checks if the left value is greater than the right value.
Less Than (<)	Checks if the left value is less than the right value.
Greater Than or Equal To (>=):	Checks if the left value is greater than or equal to the right value.
Less Than or Equal To (<=)	Checks if the left value is less than or equal to the right value.

Logical Operators

Logical operators in Python are used to combine and manipulate Boolean values (True or False). They allow you to create more complex conditions by combining multiple conditions together.

Operator	Description
and Operator	Returns True if both operands are True.
or Operator	Returns True if at least one of the operands is True.
not Operator	Returns the opposite Boolean value of the operand.

Control Statements

NeuroTech

Control Statements

- Control statements in programming are used to control the flow of execution of a program. They allow you to specify the order in which different parts of your code are executed, based on certain conditions or criteria. Control statements are essential for making decisions, repeating actions, and creating logical structures in your code.

- There are three main types of control statements:

    1. **Conditional Statements (Selection)**: Conditional statements allow you to make decisions in your program based on certain conditions. They are used to execute different blocks of code depending on whether a specific condition is true or false. The most common conditional statement is the if statement.

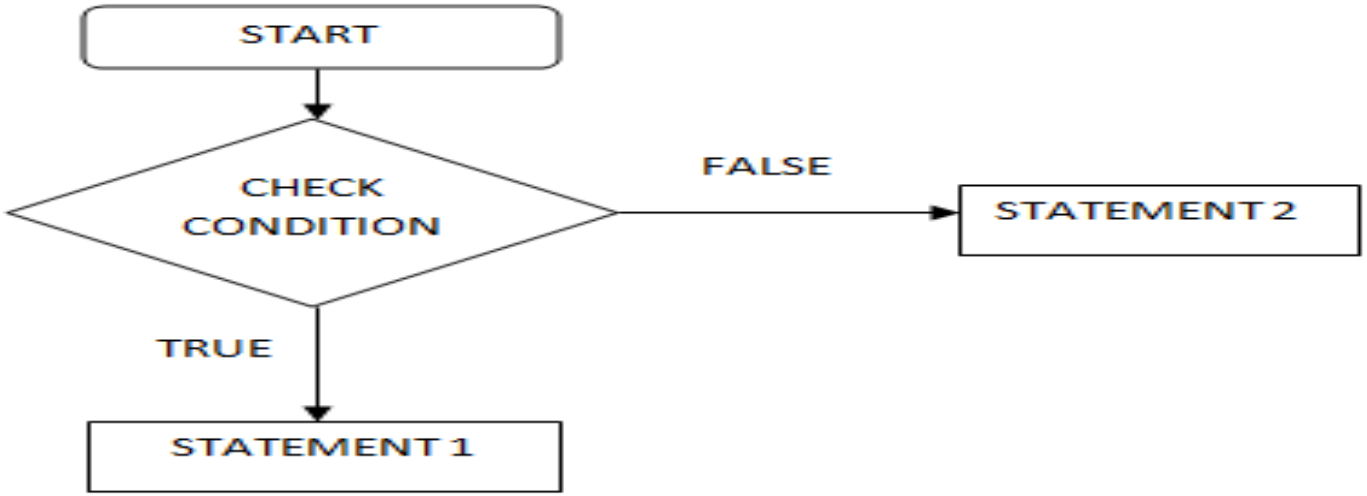
    2. **Looping Statements (Iteration)**: Looping statements allow you to repeat a block of code multiple times. They are used when you want to perform the same action or set of actions repeatedly. The two main types of loops in Python are the for loop and the while loop.

    3. **Control Transfer Statements**: Control transfer statements allow you to alter the normal flow of execution. They include the break, continue, and return statements.

Conditional Statements

Conditional Statements

- Conditional statements, also known as selection or branching statements, are an integral part of programming. They allow your code to make decisions and execute different blocks of code based on specified conditions. In Python, the primary conditional statement is the **if statement**, and it can be expanded with **elif** and **else** clauses to handle multiple conditions.



Conditional Statements

1) **if Statement**: Executes a block of code if a specified condition is True.

- Syntax:

```
if condition:
    # Code block to execute if the condition is True
```

- **'if':** The keyword that starts the if statement.
- **'condition':** An expression that evaluates to either True or False. If the condition is True, the code block following the colon is executed; otherwise, it's skipped.

- The code block following the **if** statement is indented (typically with four spaces or a tab), and it contains the code to be executed if the condition is True.

Conditional Statements

1) **if Statement**: Executes a block of code if a specified condition is True.

```
age = 20

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

2) **if-else Statement**: Executes one block of code if a condition is True and another block if it's False

```
age = 15

if age >= 18:
    print("You are an adult.")
else:
    print("You are not an adult.")
```

Conditional Statements

3) if-elif-else Statement: Handles multiple conditions by testing each one in sequence until a True condition is found, or the else block is executed if none of the conditions are True.

```
if age < 18:
    print("You are a minor.")
elif age >= 18 and age < 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

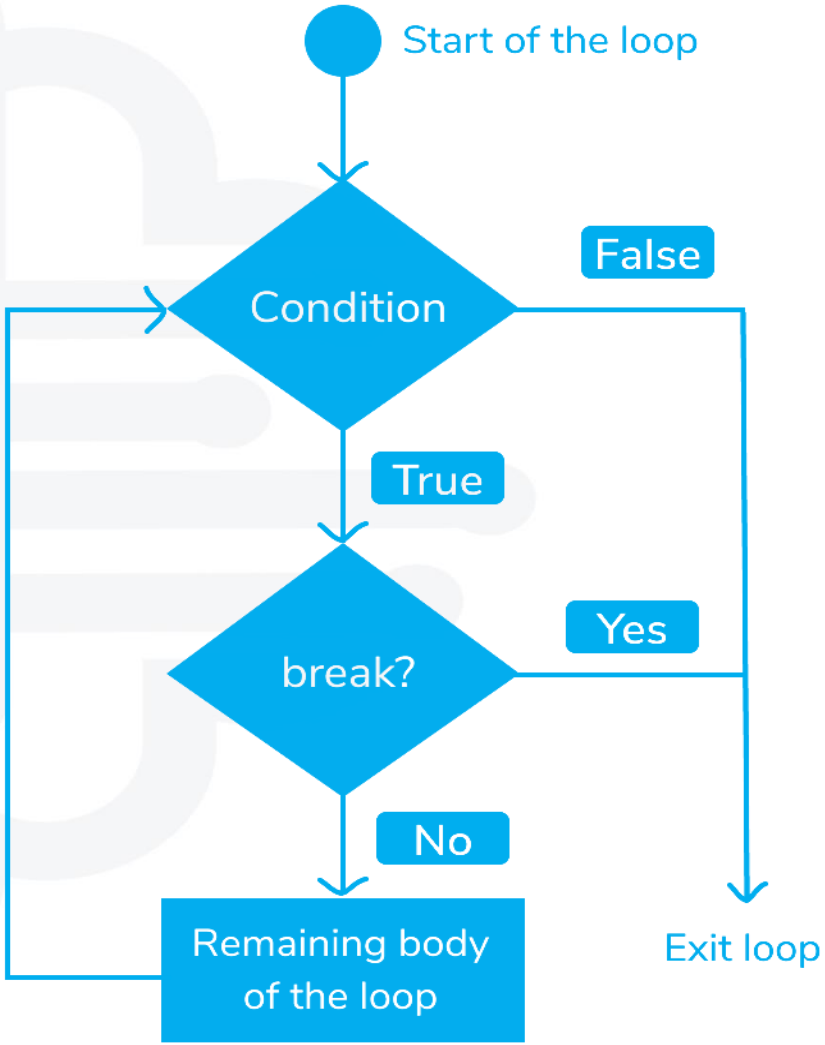
- In this example, the **if** statement is followed by multiple elif (short for "else if") statements that are evaluated in sequence. The **else** statement is the default case if none of the previous conditions are **True**.

Looping Statements

NeuroTech

Looping Statements

- Looping statements, also known as iteration or repetitive statements, allow you to execute a block of code repeatedly. They are used when you want to perform the same action or set of actions multiple times. Python provides two main types of loops: the **for** loop and the **while** loop.



Looping Statements

- **For Loop**: The for loop iterates over a sequence (such as a list, tuple, string, or range) and executes a block of code for each item in the sequence.
- **Syntax :**

```
for variable in sequence:
    # Code block to execute for each iteration
```

- **for:** The keyword that starts the for loop.
- **variable:** A variable that takes on the value of each item in the sequence during each iteration of the loop.
- **in:** The keyword used to indicate that the loop will iterate over the elements of the specified sequence.
- **sequence:** A sequence of elements, such as a list, tuple, string, or range, that the loop will iterate over.
- **::** A colon that indicates the start of the indented code block.

The code block following the for loop is indented (typically with four spaces or a tab), and it contains the code to be executed for each iteration of the loop.

Looping Statements

Here are a few examples of using the for loop:

3. Looping over a range:

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

- The range() function in Python is used to generate a sequence of numbers. It is often used in conjunction with loops, especially for loops, to iterate a specific number of times or over a range of values.

- The basic syntax of the range() function is as follows:

  - range(start, stop, step)
    - **start:** The starting value of the range (inclusive). If not provided, it defaults to 0.
    - **stop:** The stopping value of the range (exclusive). This is the value at which the sequence will stop generating numbers. It is required.
    - **step:** The increment between numbers in the sequence. If not provided, it defaults to 1.

Looping Statements

1. Generating a sequence of even numbers from 2 to 10:

```
for i in range(2, 11, 2):
    print(i)
# Output: 2, 4, 6, 8, 10
```

2. Generating a sequence of numbers from 10 to 1 in reverse:

```
for i in range(10, 0, -1):
    print(i)
# Output: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Looping Statements

- **While loop**
- **Syntax:**

```
while condition:
    # Code block to execute while the condition is True
```

  - **while:** The keyword that starts the while loop.
  - **condition:** An expression that evaluates to either True or False. The loop continues to execute as long as the condition is True.
  - **::** A colon that indicates the start of the indented code block.

- The code block following the while loop is indented (typically with four spaces or a tab), and it contains the code to be executed repeatedly while the condition is True.

Looping Statements

1. Basic while loop:

```
count = 0

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

2. Using a while loop to repeatedly prompt the user for input until a valid number is entered:

```
while True:
    try:
        num = int(input("Enter a number: "))
        break  # Exit the loop if input is a valid number
    except ValueError:
        print("Invalid input. Please enter a valid number.")
```

Control Transfer Statements

Control Transfer Statements

- Control transfer statements in Python allow you to alter the normal flow of execution within loops and conditional statements. They provide ways to control how the program proceeds under certain conditions.

1. **break**: The break statement is used to exit a loop prematurely. When encountered inside a loop, it immediately terminates the loop, regardless of whether the loop's condition has been met.

```
for i in range(10):
    if i == 5:
        break  # Exit the loop when i equals 5
    print(i)
```

Control Transfer Statements

- ✓ **continue**: The continue statement is used to skip the rest of the current iteration of a loop and move on to the next iteration. It is often used when you want to skip a particular step under certain conditions.

```
for i in range(10):
    if i % 2 == 0:
        continue  # Skip even numbers
    print(i)
```

- ✓ **pass**: The pass statement is a placeholder that does nothing. It is used when you need a statement in the syntax but don't want any action to be taken. It's commonly used as a stub when you're designing functions or classes that you plan to implement later.

```
for i in range(5):
    pass  # Do nothing
```

Exception Handling

NeuroTech

Exception Handling

- Error handling, also known as exception handling, is a crucial aspect of programming that allows you to manage and gracefully handle errors that can occur during the execution of your code. Python provides mechanisms to catch and manage these errors using the **try**, **except**, **else**, and **finally** blocks.

- **Types of Exceptions:**

- Python has a wide range of built-in exceptions that represent various types of errors. Some common exceptions include:

  - **ZeroDivisionError**: Raised when division or modulo operation is performed with zero.
  - **ValueError**: Raised when a function receives an argument of the correct type but with an invalid value.
  - **TypeError**: Raised when an operation or function is applied to an object of inappropriate type.
  - **IndexError**: Raised when an index is not found in a sequence like a list or a string.
  - **KeyError**: Raised when a dictionary key is not found.
  - **FileNotFoundError**: Raised when a file operation is performed on a non-existent file.

Exception Handling

- **Handling Exceptions:**
  - You can handle exceptions using a try and except block. The try block contains the code that might raise an exception, and the except block specifies how to handle it.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Exception Handling

- **Handling Multiple Exceptions:**
  - You can handle multiple exceptions by listing them in the except block.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ZeroDivisionError, ValueError):
    print("Error occurred.")
```

Exception Handling

- **Handling All Exceptions:**
  - You can catch all exceptions using a generic except block, but this is generally discouraged because it can make debugging difficult.

```
try:
    # some code
except:
    print("An error occurred.")
```

Exception Handling

- **The else Block:**
  The else block is executed if no exceptions are raised in the try block.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid input. Please enter a valid number.")
else:
    print("Result:", result)
```

- **The finally Block:**
  The finally block is executed regardless of whether an exception is raised or not. It's often used for cleanup operations like closing files or releasing resources.

```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found.")
finally:
    if 'file' in locals():
        file.close()
```

Exception Handling

- **Raising Exceptions:**

  You can raise exceptions explicitly using the raise statement.

```
if x < 0:
    raise ValueError("x cannot be negative")
```