Unit 1: Python - Introduction and Overview

1. Introduction and Overview

Python is a high-level, easy-to-read programming language popular for its simplicity and versatility. Created by Guido van Rossum in 1991.

It supports multiple programming styles, like procedural, object-oriented, and functional. With a rich standard library and strong community, Python is widely used in fields like web development, data science, and AI, making it ideal for beginners and experts alike.

- **Python**: High-level, interpreted, and dynamically-typed language known for its readability.
- **Uses**: Web development, data analysis, AI, automation, etc.

1.1 Comments

Comments are used to annotate code, making it more readable and maintainable. They are ignored during the execution of the program.

- **Single-line Comments:** Start with `#` and extend to the end of the line.

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

- **Multi-line Comments:** Use triple quotes `''' ... '''` or `""" ... """`.

```
"""
This is a multi-line comment.
It can span multiple lines.
"""
print("Multi-line comment example.")
```

1.2 Keywords and Identifiers

- **Keywords:** Reserved words that have specific meanings and functions within the language. They cannot be used as identifiers (variable names, function names, etc.). Examples include `if`, `else`, `while`, `for`, `break`, `continue`, etc.
- **Identifiers:** Names given to variables, functions, classes, and other objects. They must follow these rules:

- Must start with a letter (A-Z or a-z) or an underscore (`_`).
- Can contain letters, digits, and underscores.
- Cannot be a keyword or contain special characters (`@` , `#` , `$` , etc.).

```
variable_1 = 10  # Valid identifier
1st_variable = 20  # Invalid identifier (starts with a digit)
```

1.3 Variables and Assignment Statements

- **Variables** are used to store data that can be referenced and manipulated in a program. A variable is essentially a name that points to a specific value in memory. Variables do not need to be declared with a type in Python, as it is dynamically typed.
- **Assignment Statements** are used to assign values to variables using the `=` operator.
- **Example:**

```
name = "Sandy"
age = 20
aura = 9000.50
```

1.4 Standard Data Types

- **Integer:** Represents whole numbers (e.g., `5` , `-10` , `0`).
- **Float:** Represents numbers with a fractional part (e.g., `3.14` , `-0.5`).
- **String:** A sequence of characters enclosed in single (`'...'`) or double (`"..."`) quotes.
- **Boolean:** Represents `True` or `False` .

1.5 Other Built-in Data Types

- **List:** An ordered, mutable (changeable) collection that can hold items of different types. Lists are created using square brackets `[...]` .

```
my_list = [1, 2, 3, "apple", True]
```

- **Tuple:** An ordered, immutable (unchangeable) collection, often used for storing related data. Tuples are created using parentheses `(...)` .

```
my_tuple = (1, 2, 3, "banana")
```

- **Set:** An unordered collection of unique elements. Sets are created using curly braces `{...}` or the `set()` function.

```
my_set = {1, 2, 3, "cherry"}
```

- **Dictionary:** An unordered collection of key-value pairs, where each key is unique. Dictionaries are created using curly braces `{key: value, ...}`.

```
my_dict = {"name": "Sandy", "age": 20}
```

1.6 Internal Types

Internal types refer to the types that the Python interpreter uses internally to manage data and support various operations.

These types are foundational to Python's internal mechanics and often enable functionality for higher-level types (like lists and dictionaries) and operations (like memory management, error handling, and type checking).

- **NoneType**: Represents `None`, used for a null or absent value.
- **Ellipsis Type** (`...`): A placeholder in slicing or function annotations.
- **Code Type**: Represents compiled byte-code for functions, enabling code execution.
- **Frame Type**: Holds execution context for functions, useful in debugging.
- **Traceback Type**: Stores error trace details, helpful for debugging.
- **Slice Type**: Manages slicing information for lists and other sequences.
- **Function Type**: Represents all functions and lambdas.
- **Generator Type**: Represents generators, allowing efficient iteration.
- **Module Type**: Represents imported modules.

1.7 Operators

Operators are symbols or keywords that perform operations on values (also called operands). Operators allow you to manipulate data and perform calculations, comparisons, and other operations.

1.7.1 Arithmetic Operators

These operators are used to perform mathematical operations such as addition, subtraction, multiplication, and division.

Operator	Description	Example
`+`	Addition	`5 + 3` → `8`
`-`	Subtraction	`5 - 3` → `2`
`*`	Multiplication	`5 * 3` → `15`
`/`	Division (float)	`5 / 3` → `1.67`
`%`	Modulus (remainder)	`5 % 3` → `2`
`**`	Exponentiation	`5 ** 3` → `125`
`//`	Floor Division	`5 // 3` → `1`

1.7.2 Comparison Operators

Comparison operators are used to compare two values. They return a boolean value, `True` or `False`.

Operator	Description	Example
`==`	Equal to	`5 == 3` → `False`
`!=`	Not equal to	`5 != 3` → `True`
`>`	Greater than	`5 > 3` → `True`
`<`	Less than	`5 < 3` → `False`
`>=`	Greater than or equal to	`5 >= 3` → `True`
`<=`	Less than or equal to	`5 <= 3` → `False`

1.7.3 Assignment Operators

Assignment operators are used to assign values to variables.

Operator	Description	Example
`=`	Assign	`a = 5`
`+=`	Add and assign	`a += 3` (same as `a = a + 3`)
`-=`	Subtract and assign	`a -= 3`
`*=`	Multiply and assign	`a *= 3`
`/=`	Divide and assign	`a /= 3`
`%=`	Modulus and assign	`a %= 3`
`**=`	Exponent and assign	`a **= 3`
`//=`	Floor divide and assign	`a //= 3`

1.7.4 Logical Operators

Logical operators are used to combine conditional statements.

Operator	Description	Example
`and`	Returns True if both statements are true	`(5 > 3) and (2 < 4)` → `True`
`or`	Returns True if one of the statements is true	`(5 < 3) or (2 < 4)` → `True`
`not`	Returns the opposite of the statement	`not(5 > 3)` → `False`

1.7.5 Bitwise Operators

Bitwise operators operate on binary digits (bits) of integers.

Operator	Description	Example
`&`	Bitwise AND	`5 & 3`
`` ` ``	Bitwise OR	5 `` ` `` 3
`^`	Bitwise XOR	`5 ^ 3`
`~`	Bitwise NOT	`~5`
`<<`	Bitwise Left Shift	`5 << 1`
`>>`	Bitwise Right Shift	`5 >> 1`

1.7.6 Membership Operators

Membership operators test if a sequence contains an element.

Operator	Description	Example
`in`	Returns True if a value is in a sequence	`'a' in 'apple'` → `True`
`not in`	Returns True if a value is not in a sequence	`'b' not in 'apple'` → `True`

1.7.7 Identity Operators

Identity operators check if two variables refer to the same object in memory.

Operator	Description	Example
`is`	Returns True if both variables are the same object	`a is b`
`is not`	Returns True if both variables are not the same object	`a is not b`

1.7.8 Ternary Operator

The ternary operator in Python is a one-line shorthand for an if-else statement.

- **Syntax:**

```
<value_if_true> if <condition> else <value_if_false>
```

- **Example:**

```
result = "Even" if (num % 2 == 0) else "Odd"
```

1.8. Built-in Functions

A built-in function in Python is a core function provided by the Python standard library that is directly accessible in any Python environment, as it's loaded into the namespace by default.

These are designed to handle common tasks and simplify coding by providing quick solutions for data manipulation, type conversion, mathematical calculations, input/output, and more.

Python provides many built-in functions such as `print()`, `len()`, `type()`, `input()`, and more.

- **Example:**

```
print("Hello, World!")  # Prints a message to the screen
x = len("Hello")  # Returns the length of the string
```

1.8.1 Data Type Conversion Functions

These functions allow you to convert between different data types.

Function	Description	Example
`int()`	Converts to an integer	`int("5")` → `5`
`float()`	Converts to a floating-point number	`float("3.14")` → `3.14`
`str()`	Converts to a string	`str(123)` → `"123"`
`bool()`	Converts to a boolean	`bool(1)` → `True`
`list()`	Converts to a list	`list("abc")` → `['a', 'b', 'c']`
`tuple()`	Converts to a tuple	`tuple([1, 2])` → `(1, 2)`
`set()`	Converts to a set	`set([1, 2, 2])` → `{1, 2}`
`dict()`	Converts to a dictionary	`dict(a=1, b=2)` → `{'a': 1, 'b': 2}`

1.8.2 Input and Output Functions

These functions manage input and output operations in Python.

Function	Description	Example
`print()`	Outputs data to the console	`print("Hello")`
`input()`	Takes input from the user as a string	`name = input("Enter your name: ")`

1.8.3 Mathematical Functions

Python includes functions for basic mathematical operations.

Function	Description	Example
`abs()`	Returns the absolute value	`abs(-5)` → `5`
`pow()`	Returns a number raised to a power	`pow(2, 3)` → `8`
`round()`	Rounds a number to a specified number of decimal places	`round(3.14159, 2)` → `3.14`
`max()`	Returns the largest item	`max(3, 5, 2)` → `5`
`min()`	Returns the smallest item	`min(3, 5, 2)` → `2`
`sum()`	Sums all items in an iterable	`sum([1, 2, 3])` → `6`

1.8.4 String Functions

Python has functions for manipulating strings.

Function	Description	Example
`len()`	Returns the length of an object	`len("hello")` → `5`
`str()`	Converts any data type to a string	`str(123)` → `"123"`
`ord()`	Returns the Unicode code of a character	`ord('a')` → `97`
`chr()`	Converts a Unicode code to a character	`chr(97)` → `'a'`

1.8.5 Type Checking Functions

These functions check the data type of an object.

Function	Description	Example
`type()`	Returns the type of an object	`type(123)` → `<class 'int'>`
`isinstance()`	Checks if an object is an instance of a specific type	`isinstance(123, int)` → `True`

1.8.6 Object and Collection Functions

These functions work with collections, iterables, and other objects.

Function	Description	Example
`len()`	Returns the length of an object	`len([1, 2, 3])` → `3`
`sorted()`	Returns a sorted list from an iterable	`sorted([3, 1, 2])` → `[1, 2, 3]`
`reversed()`	Returns an iterator that accesses elements in reverse	`list(reversed([1, 2, 3]))` → `[3, 2, 1]`
`enumerate()`	Returns an enumerate object	`list(enumerate(['a', 'b']))` → `[(0, 'a'), (1, 'b')]`
`zip()`	Aggregates elements from multiple iterables	`list(zip([1, 2], ['a', 'b']))` → `[(1, 'a'), (2, 'b')]`

1.8.7 Functional Programming Functions

Python supports functional programming with functions that work on iterables.

Function	Description	Example
`map()`	Applies a function to all items in an iterable	`list(map(str, [1, 2, 3]))` → `['1', '2', '3']`
`filter()`	Filters items in an iterable based on a condition	`list(filter(lambda x: x > 0, [-1, 0, 1]))` → `[1]`
`reduce()`	Applies a rolling computation to items in an iterable	Requires `from functools import reduce`

1.8.8 Utility Functions

These are useful functions for miscellaneous tasks.

Function	Description	Example
`id()`	Returns the memory address of an object	`id("hello")`
`help()`	Displays documentation for an object	`help(print)`
`dir()`	Returns a list of attributes and methods for an object	`dir(list)`
`eval()`	Parses and evaluates a string expression	`eval("2 + 2")` → `4`
`exec()`	Executes Python code dynamically	`exec("print('Hello')")`

2. Introduction to Numbers

Numbers are one of the basic data types used to store numeric values. Python supports various types of numbers, including integers, floating-point numbers, and complex numbers.

2.1 Integers (`int`)

Integers represent whole numbers without any decimal point. They can be positive, negative, or zero. Python supports arbitrarily large integers, meaning integers can grow as large as your memory allows.

- **Example:**

```
a = 5
b = -10
```

2.2 Floating Point Real Numbers (`float`)

Floating-point numbers (or simply "floats") represent real numbers with a decimal point. They are used to handle numbers that have fractions or decimal parts.

Floats can also represent very large or very small values by using scientific notation.

- **Example:**

```
pi = 3.14159
```

2.3 Complex Numbers

Complex numbers have a real and an imaginary part, represented as `a + bj`, where:

- `a` is the real part.
- `b` is the imaginary part (with `j` as the imaginary unit in Python).

Complex numbers are used in advanced mathematical calculations, particularly in engineering and scientific computations.

- **Example:**

```
num1 = 2 + 3j      # 2 is the real part, 3 is the imaginary part
```

3. Sequences and Strings

3.1 Sequences

A sequence is an ordered collection of items, where each item has a specific position (or index). Sequences allow you to access elements by their position, and they support operations such as indexing, slicing, and iterating over the elements.

Common types of sequences in Python include:

1. **Strings**: A sequence of characters. Example: `"Hello, World!"`
2. **Lists**: An ordered, mutable (modifiable) collection of items. Example: `[1, 2, 3, 4, 5]`
3. **Tuples**: An ordered, immutable (non-modifiable) collection of items. Example: `(1, 2, 3, 4, 5)`
4. **Ranges**: A sequence of numbers, often used in loops. Example: `range(1, 10)`

Key Characteristics of Sequences

- **Indexing**: Each item in a sequence has an index, starting from 0. You can access individual items using their index.
- **Slicing**: You can retrieve a subset of the sequence using slicing.
- **Iteration**: Sequences can be looped over using a `for` loop.
- **Length**: You can get the number of items in a sequence using `len()`.

3.2 Strings

Strings are a sequence of characters. String is enclosed within single (`'...'`) or double (`"..."`) quotes. They can be manipulated using string methods.

Strings are commonly used to represent text, such as words, sentences, or any combination of letters, symbols, and numbers.

Key Characteristics of Strings in Python

1. **Immutable**: Strings cannot be changed once they are created. Any modification to a string results in a new string being created.
2. **Indexed**: Each character in a string has a unique index (starting from `0` for the first character) that can be accessed to retrieve individual characters.
3. **Supports Various Operations**: Python provides several built-in functions and operators to work with strings, including concatenation, slicing, and various string methods.

Example:

```
s1 = "Hello, World!"
```

3.3 String-Only Operators

These are operators that work specifically on strings:

1. **Concatenation (`+`)**: Combines two strings into one.

```
greeting = "Hello, " + "world!"
# Output: "Hello, world!"
```

2. **Repetition (`*`)**: Repeats a string a specified number of times.

```
laugh = "ha"
result = laugh * 3
# Output: "hahaha"
```

3. **Membership (`in` and `not in`)**: Checks if a substring exists within a string.

```
text = "Python programming"
print("Python" in text)     # Output: True
print("Java" not in text)   # Output: True
```

4. **Comparison Operators (`==`, `!=`, `<`, `>`, `<=`, `>=`)**: Compares two strings lexicographically.

```
print("apple" == "apple")   # Output: True
print("apple" < "banana")   # Output: True
```

3.4 String Built-in Methods

In python, Strings come with a wide array of built-in methods that help manipulate, format, and analyze text data.

3.4.1 Case and Capitalization Methods

Method	Description	Example
`capitalize()`	Capitalizes the first letter of the string.	`"hello".capitalize()` → `"Hello"`
`casefold()`	Converts the string to lowercase (useful for comparisons).	`"HELLO".casefold()` → `"hello"`
`lower()`	Converts the string to lowercase.	`"HELLO".lower()` → `"hello"`
`upper()`	Converts the string to uppercase.	`"hello".upper()` → `"HELLO"`
`title()`	Capitalizes the first letter of each word in the string.	`"hello world".title()` → `"Hello World"`
`swapcase()`	Swaps the case of all letters in the string.	`"Hello".swapcase()` → `"hELLO"`

3.4.2 Trimming and Whitespace Methods

Method	Description	Example
`strip()`	Removes leading and trailing whitespace.	`" hello ".strip()` → `"hello"`
`lstrip()`	Removes leading whitespace.	`" hello".lstrip()` → `"hello"`
`rstrip()`	Removes trailing whitespace.	`"hello ".rstrip()` → `"hello"`
`expandtabs()`	Expands tabs to spaces.	`"hello\tworld".expandtabs(4)` → `"hello world"`

3.4.3 Search and Replace Methods

Method	Description	Example
`find()`	Returns the lowest index of the substring (or `-1` if not found).	`"hello".find("e")` → `1`
`rfind()`	Returns the highest index of the substring (or `-1` if not found).	`"hello".rfind("e")` → `1`
`index()`	Similar to `find()`, but raises `ValueError` if not found.	`"hello".index("e")` → `1`
`rindex()`	Similar to `rfind()`, but raises `ValueError` if not found.	`"hello".rindex("e")` → `1`
`replace()`	Replaces a substring with another substring.	`"hello".replace("e", "a")` → `"hallo"`
`count()`	Returns the number of occurrences of a substring.	`"hello".count("l")` → `2`

3.4.4 Testing Methods

Method	Description	Example
`startswith()`	Checks if the string starts with a specified substring.	`"hello".startswith("he")` → `True`
`endswith()`	Checks if the string ends with a specified substring.	`"hello".endswith("lo")` → `True`
`isalnum()`	Checks if all characters are alphanumeric.	`"hello123".isalnum()` → `True`
`isalpha()`	Checks if all characters are alphabetic.	`"hello".isalpha()` → `True`
`isdigit()`	Checks if all characters are digits.	`"12345".isdigit()` → `True`
`isnumeric()`	Checks if all characters are numeric.	`"12345".isnumeric()` → `True`
`isspace()`	Checks if the string contains only whitespace characters.	`" ".isspace()` → `True`
`isupper()`	Checks if all characters are uppercase.	`"HELLO".isupper()` → `True`
`islower()`	Checks if all characters are lowercase.	`"hello".islower()` → `True`
`istitle()`	Checks if the string is in title case.	`"Hello World".istitle()` → `True`

3.4.5 Encoding and Decoding Methods

Method	Description	Example
`encode()`	Encodes the string to a bytes object using a specified encoding.	`"hello".encode("utf-8")` → `b'hello'`
`decode()`	Decodes a bytes object to a string using a specified encoding.	`b'hello'.decode("utf-8")` → `"hello"`

3.4.6 String Formatting and Alignment Methods

Method	Description	Example
`format()`	Formats the string using placeholders.	`"Hello, {}".format("world")` → `"Hello, world"`
`center()`	Centers the string within a specified width.	`"hello".center(10, "-")` → `"--hello---"`
`ljust()`	Left-justifies the string within a specified width.	`"hello".ljust(10, "-")` → `"hello-----"`
`rjust()`	Right-justifies the string within a specified width.	`"hello".rjust(10, "-")` → `"-----hello"`
`zfill()`	Pads the string with zeros from the left to a specified width.	`"42".zfill(5)` → `"00042"`

3.4.7 Splitting and Joining Methods

Method	Description	Example
`split()`	Splits the string into a list of substrings.	`"hello world".split()` → `["hello", "world"]`
`rsplit()`	Splits the string from the right.	`"hello world".rsplit(" ", 1)` → `["hello", "world"]`
`splitlines()`	Splits the string into lines.	`"hello\nworld".splitlines()` → `["hello", "world"]`
`join()`	Joins a list of strings into a single string.	`"-".join(["hello", "world"])` → `"hello-world"`

3.4.8 Other Useful Methods

Method	Description	Example
`len()`	Returns the length of the string.	`len("hello")` → `5`
`ord()`	Returns the Unicode code point of a character.	`ord('a')` → `97`
`chr()`	Returns the character that corresponds to a Unicode code point.	`chr(97)` → `'a'`
`format_map()`	Similar to `format()`, but uses a dictionary.	`"Hello, {name}".format_map({"name": "Sandy"})` → `"Hello, Sandy"`

3.5 Special Features of Strings

1. **Immutable**: Strings in Python are immutable, meaning once a string is created, its value cannot be changed. Any operation that modifies a string results in a new string object.

```
s = "Hello"
s[0] = 'h'  # Raises TypeError: 'str' object does not support item assignment
```

2. **Concatenation**: You can concatenate strings using the `+` operator or by using the `.join()` method.

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2  # Output: "Hello World"
```

3. **Repetition**: The `*` operator allows you to repeat strings.

```
s = "abc" * 3  # Output: "abcabcabc"
```

4. **Indexing/Slicing**: Strings can be indexed and sliced like lists. Indexing starts at 0 for the first character.

```
s = "Hello"
print(s[0])  # Output: 'H'
print(s[-1])  # Output: 'o'
print(s[1:4])  # Output: 'ell'
```

5. **Multiline**: We can create multiline strings using triple quotes (`'''` or `"""`).

```
multiline_str = """This is
a multiline
string."""
```

6. **Escape Sequences**: Strings can contain special characters like newlines (`\n`), tabs (`\t`), etc., using escape sequences.

```
s = "Hello\nWorld"  # Output:
# Hello
# World
```

7. **Methods**: Python offers methods like `.upper()` , `.lower()` , `.replace()` , `.split()` , `.find()` .

```
s = "hello world"
print(s.upper())  # Output: "HELLO WORLD"
print(s.split())  # Output: ['hello', 'world']
```

8. **f-strings**: Format strings using `{}` inside a string prefixed with `f` .

```
name = "Sandy"
age = 20
greeting = f"Hello, my name is {name} and I am {age} years old."
print(greeting) # Output: Hello, my name is Sandy and I am 20 years old.
```

9. **`.format()`** : Another way to format strings (before f-strings).

```
name = "Sandy"
age = 20
greeting = "Hello, my name is {} and I am {} years old.".format(name, age)
print(greeting) # Output: Hello, my name is Sandy and I am 20 years old.
```

10. **Encoding/Decoding**: We can convert strings to bytes using encoding (`encode()`), and convert bytes back to strings using decoding (`decode()`).

```
s = "Hello"
encoded_str = s.encode('utf-8')  # Convert string to bytes
decoded_str = encoded_str.decode('utf-8')  # Convert bytes back to string
```

11. **Comparison**: Strings can be compared using relational operators like `==` , `!=` , `<` , `>` , etc.

```
s1 = "apple"
s2 = "banana"
print(s1 < s2)  # Output: True (because "apple" comes before "banana"
lexicographically)
```

12. **Raw Strings**: Raw strings (denoted by prefixing the string with `r`) treat backslashes (`\`) as literal characters rather than escape characters. This is especially useful for regular expressions or file paths.

```
raw_str = r"C:\Users\mrsandy\Documents"
print(raw_str)  # Output: C:\Users\mrsandy\Documents
```

13. **Traversal**: Loop through characters using a `for` loop.

```
s = "Hello"
for char in s:
    print(char)
```

4. Conditional Statements

Conditional statements are used to execute certain blocks of code based on whether a given condition evaluates to `True` or `False` . They allow your program to make decisions and control the flow of execution.

4.1 if Statement

The `if` statement is used to test a condition. If the condition evaluates to `True` , the code inside the `if` block will be executed. If it evaluates to `False` , the code inside the `if` block will be skipped.

- **Syntax:**

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

4.2 else Statement

The `else` statement is used to execute a block of code when the `if` condition is false.

- **Syntax:**

```
if condition:
    # Code to execute if condition is True
else:
    # Code to execute if condition is False
```

4.3 elif Statement

The `elif` (short for "else if") statement allows you to check multiple conditions.

It follows an `if` statement, and if the `if` condition is `False`, the program checks the `elif` condition. You can have multiple `elif` conditions.

- **Syntax:**

```
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition2 is True
else:
    # Code to execute if no condition is True
```

4.4 Nested if Statements:

You can also have conditional statements inside other conditional statements, which is called nesting.

- **Example:**

```
age = 20
if age >= 18:
    if age >= 21:
        print("You can drink alcohol in the USA.")
    else:
        print("You can vote, but not drink alcohol in the USA.")
else:
    print("You are too young.")
```

5. Loops

Loops are used to execute a block of code repeatedly as long as a specified condition is met. Loops allow you to automate repetitive tasks and iterate over sequences like lists, tuples, dictionaries, and ranges.

5.1 for Loop

The `for` loop is used to iterate over a sequence (like a list, tuple, or string) or a range of numbers. It executes a block of code for each item in the sequence.

- **Syntax:**

```
for variable in sequence:
    # Code to execute for each item in the sequence
```

- **Example:**

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

Iterating with `range()`: The `range()` function generates a sequence of numbers. It is commonly used to run a loop a specific number of times.

- **Example:**

```
# Looping through a range of numbers
for i in range(5):
    print(i)
```

5.2 while Loop

The `while` loop executes a block of code as long as a condition is `True`. The condition is evaluated before each iteration, and if it is `False` at the start, the code inside the loop will not be executed.

- **Syntax:**

```
while condition:
    # Code to execute as long as condition is True
```

- **Example:**

```
# Asking for user input until they provide a valid positive number
number = -1
while number <= 0:
    number = int(input("Please enter a positive number: "))
    if number <= 0:
        print("That's not a positive number. Try again.")

print(f"Thank you! You entered: {number}")
```

5.3 Nested Loops:

You can place loops inside other loops, which is known as a **nested loop**. This is useful when you need to iterate over multiple sequences or perform complex iteration.

- **Example:**

```
# Nested for loop
for i in range(3):
    for j in range(2):
        print(f"i = {i}, j = {j}")
```

6. Other Flow Statements

6.1 break Statement

The `break` statement is used to exit the loop prematurely, regardless of the loop's condition.

- **Example:**

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

6.2 continue Statement

The `continue` statement skips the current iteration of the loop and moves to the next iteration.

- **Example:**

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)  # Prints only odd numbers
```

6.3 pass Statement

The `pass` statement is a null operation; it is used as a placeholder when a statement is required syntactically but no action is needed.

- **Example:**

```
for letter in "Python":
    if letter == 'h':
        pass
    print(letter) # Prints Pyton
```