UNIT I: 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.

1. Comments

Definition

Comments are lines in a program that are ignored by the compiler or interpreter during execution. They are used to explain the code, make it more readable, and assist other developers in understanding its purpose.

Features

- **Improve Readability:** Comments describe what the code does, making it easier to understand.
- **Debugging Help:** Useful for debugging by providing context or temporarily disabling code.
- **Ignored by Interpreter:** Comments have no impact on program execution.

Types of Comments

1. **Single-Line Comments**: Begin with a `#` symbol.
    - Example:

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

2. **Multi-Line Comments**: Enclosed within triple quotes `'''` or `"""`.
    - Example:

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

---

2. Keywords and Identifiers

2.1 Keywords

Definition

Keywords are reserved words in a programming language with predefined meanings. They cannot be used as variable names or identifiers.

Features

- Fixed and case-sensitive.
- Provide specific functionality or denote actions in the code.

Examples

Some Python keywords include:

- `if`, `else`, `while`, `for`, `def`, `class`, `return`.

```
if x > 0:
    print("Positive number")
```

2.2 Identifiers

Definition

Identifiers are the names used to identify variables, functions, classes, or other objects in a program.

Rules

- Must begin with a letter (A-Z or a-z) or an underscore (`_`).
- Cannot be a keyword.

- Case-sensitive.

Examples

```
variable_name = 10
print(variable_name)
```

---

3. Variables and Assignment Statements

3.1 Variables

Definition

Variables are containers for storing data values. They are dynamically typed in Python, meaning the type is assigned at runtime.

Features

- Dynamic typing.
- No explicit declaration required.

Example

```
x = 5
y = "Hello"
print(x, y)
```

3.2 Assignment Statements

Definition

Assignment statements are used to assign a value to a variable using the `=` operator.

Example

```
x = 10
y = x + 5
print(y)  # Output: 15
```

---

4. Standard Types

Definition

Python provides various standard data types used to define the type of data stored in variables.

Types

1. **Numeric Types**: `int`, `float`, `complex`.
2. **Sequence Types**: `list`, `tuple`, `range`.
3. **Text Type**: `str`.
4. **Boolean Type**: `bool`.

Example

```
x = 10   # int
y = 3.14   # float
z = "Hello"   # str
```

---

5. Other Built-in Types

Common Built-in Types

1. **NoneType**: Represents the absence of a value (`None`).

2. **Dictionary**: Key-value pairs.

   - Example:

     ```
     my_dict = {"key": "value"}
     print(my_dict["key"])   # Output: value
     ```

3. **Set and Frozenset**: Unordered collections of unique elements.

---

6. Internal Types

Definition

Internal types refer to data types used by the Python interpreter internally, such as code objects, stack frames, and more. These are generally not directly manipulated by users.

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).

Some Examples:

- **NoneType**: Represents `None`, used for a null or absent value.
- **Frame Type**: Holds execution context for functions, useful in debugging.
- **Function Type**: Represents all functions and lambdas.

---

7. Operators

Definition

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.

Types of Operators

1. **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%`, `**`, `//`.
    - Example:

    ```
    result = 5 + 3
    print(result)  # Output: 8
    ```

2. **Comparison Operators**: `==`, `!=`, `>`, `<`, `>=`, `<=`.
3. **Assignment Operators**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `//=`
4. **Logical Operators**: `and`, `or`, `not`.
5. **Bitwise Operators**: `&`, `|`, `^`, `~`, `<<`, `>>`.
6. **Membership Operators**: `in`, `not in`
7. **Identity Operators**: `is`, `is not`

---

8. Built-in Functions

Definition

Built-in functions are predefined functions provided by Python that can be used directly.

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.

Examples

1. **Input and Output Functions**: `print()`, `input()`.
2. **Type Conversion Functions**: `int()`, `float()`, `str()`.
3. **Mathematical Functions**: `abs()`, `pow()`, `round()`.

Example

```
x = input("Enter a number: ")
print(int(x) * 2)
```

9. 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.

9.1 Integers (`int`)

- Whole numbers without a fractional part.
- Example:

  ```
  x = 10
  ```

9.2 Floating Point Real Numbers (`float`)

- Numbers with decimal points.
- Example:

```
y = 3.14
```

9.3 Complex Numbers

- Numbers with real and imaginary parts.
- Example:

```
z = 3 + 4j
```

---

10. 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()`.

10.1 Strings

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

- String is enclosed within single (`'...'`) or double (`"..."`) quotes. They can be manipulated using string methods.
- String-specific operators:
  - Concatenation: `+`
  - Repetition: `*`
  - Example:

    ```
    text = "Hello" + " World"
    print(text)
    ```

10.2 String Built-in Methods

Definition

Python provides numerous built-in methods for string manipulation, which simplify common operations like formatting, searching, or replacing text.

Common String Methods

1. `len()`: Returns the length of the string.
   - Example:

     ```
     text = "Hello"
     print(len(text))  # Output: 5
     ```

2. `lower()` **and** `upper()`: Convert a string to lowercase or uppercase.
   - Example:

     ```
     print("Hello".lower())  # Output: hello
     print("Hello".upper())  # Output: HELLO
     ```

3. `strip()`: Removes leading and trailing whitespace.
   - Example:

```python
text = "  Hello  "
print(text.strip())  # Output: Hello
```

4. `replace()`: Replaces occurrences of a substring.
   - Example:

```python
print("Hello World".replace("World", "Python"))  # Output: Hello Python
```

5. `split()`: Splits a string into a list of substrings based on a delimiter.
   - Example:

```python
print("Hello,World".split(","))  # Output: ['Hello', 'World']
```

6. `find()`: Returns the index of the first occurrence of a substring. Returns `-1` if not found.
   - Example:

```python
print("Hello".find("e"))  # Output: 1
```

10.3 Special Features of Strings

1. Immutability

Strings are immutable, meaning their values cannot be changed after creation. Any modification creates a new string.

- Example:

```python
text = "Hello"
new_text = text.replace("H", "J")
print(text)       # Output: Hello
print(new_text)   # Output: Jello
```

2. String Slicing

Strings can be sliced to extract substrings using the syntax `string[start:end:step]`.

- Example:

```
text = "Hello"
print(text[1:4])  # Output: ell
```

---

11. Conditionals

Definition

Conditional statements control the flow of a program by executing specific blocks of code based on given conditions.

Conditional statements are used to execute certain blocks of code based on whether a given condition evaluates to `True` or `False`.

Types of Conditional Statements

11.1 `if` Statement

Executes a block of code if a condition is `True`.

- Example:

```
x = 10
if x > 5:
    print("x is greater than 5")
```

11.2 `else` Statement

Provides an alternative block of code if the condition is `False`.

- Example:

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

11.3 `elif` Statement

Tests additional conditions when the initial `if` condition is `False`.

- Example:

```python
x = 5
if x > 10:
    print("x is greater than 10")
elif x == 5:
    print("x is equal to 5")
else:
    print("x is less than 5")
```

---

12. Loops

Definition

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.

Types of Loops

12.1 `while` Loop

Repeats as long as the 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.

- Example:

```python
x = 1
while x <= 5:
    print(x)
    x += 1
```

12.2 `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.

- Example:

```
for i in range(1, 6):
    print(i)
```

---

13. Loop Control Statements

13.1 `break` Statement

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

- Example:

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)  # Output: 1 2 3 4
```

13.2 `continue` Statement

Skips the current iteration and moves to the next.

- Example:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)  # Output: 1 2 4 5
```

13.3 `pass` Statement

Used as a placeholder for code that is yet to be implemented.

- Example:

```
for i in range(5):
    pass  # No action performed
```
