Chapter - 2

What are Comments?

Comments enhance the readability of the code and help the programmers to understand the code very carefully. Python single-line comment starts with the hashtag symbol (#) with no white spaces and lasts till the end of the line.
Python does not provide the option for multiline comments.

Why to use them?

1. Enhance code readability
2. Explaining code to other

What are Strings?

A String is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once you have created a string, you cannot change it.

Eg: print("How are you today!")

Operations on Strings

1. **Concatenation**: Combining two or more strings into one.

```
str1 = "Hello"
```

```
str2 = "World"

result = str1 + " " + str2

print(result)  # Output: Hello World
```

2. **Indexing**: Accessing individual characters in a string by their position.

```
text = "Python"

print(text[0])  # Output: P

print(text[3])  # Output: h
```

3. **Slicing**: Extracting a substring from a string.

```
text = "Python"

print(text[2:5])  # Output: tho
```

4. **Length**: Finding the length of a string.

```
text = "Python"

print(len(text))  # Output: 6
```

5. **Lowercase and Uppercase**: Converting the case of characters in a string.

```
text = "Hello World"
```

```
print(text.lower())   # Output: hello world

print(text.upper())   # Output: HELLO WORLD
```

6. **Splitting**: Breaking a string into a list of substrings based on a delimiter.

```
text = "apple,banana,orange"

fruits = text.split(",")

print(fruits)  # Output: ['apple', 'banana', 'orange']
```

7. **Stripping**: Removing leading and trailing whitespace characters from a string.

```
text = "   Hello   "

stripped_text = text.strip()

print(stripped_text)  # Output: Hello
```

8. **Finding Substrings**: Searching for substrings within a string

```
text = "Hello World"

print(text.find("World"))  # Output: 6
```

9. **Replacing:** Replacing occurrences of a substring with another substring.

```
text = "Hello World"
```

```
new_text = text.replace("World", "Universe")

print(new_text)  # Output: Hello Universe
```

10. **Checking Substring Existence**: Checking if a substring exists within a string.

```
text = "Hello World"

print("World" in text)  # Output: True

print("Python" in text)  # Output: False
```

Datatypes

In Python, data types define the type of data that a variable can hold. Here are some of the basic data types in Python along with examples:

1. **Integer (`int`)**: Represents whole numbers without any decimal points.

```
num = 10
```

2. **Float (`float`)**: Represents numbers with decimal points.

```
num = 3.14
```

3. **String (`str`)**: Represents sequences of characters, enclosed within single, double, or triple quotes.

```
text = "Hello, World!"
```

4. **Boolean (`bool`)**: Represents a binary value, either `True` or `False`

```
is_raining = True
```

5. **List (`list`)**: Represents an ordered collection of items, which can be of different data types. Lists are mutable.

```
numbers = [1, 2, 3, 4, 5]
```

6. **Dictionary (`dict`)**: Represents a collection of key-value pairs, where each key is associated with a value. Dictionaries are unordered.

```
person = {'name': 'John', 'age': 30, 'city': 'New York'}
```

7. **Set (`set`)**: Represents an unordered collection of unique items. Sets do not allow duplicate elements.

```
unique_numbers = {1, 2, 3, 4, 5}
```

8. **NoneType (`None`)**: Represents the absence of a value or a null value.

```
result = None
```

Charsets

In Python, a character set refers to a collection of characters, often grouped together for specific purposes or operations.

1. **ASCII (American Standard Code for Information Interchange)**:

ASCII is a character encoding standard that represents text in computers. It uses 7 bits to represent each character, providing a total of 128 different characters.

Example:

```
# ASCII representation of characters

char = 'A'

print(ord(char))  # Output: 65
```

2. **Unicode**:

Unicode is a standard for encoding characters in most of the world's writing systems. It supports a vast range of characters, including symbols, emojis, and characters from various languages. In Python, strings are Unicode by default.

Example:

```
# Unicode representation of characters

char = '😊'
```

```
print(ord(char))  # Output: 128522
```

Operators

Operators in Python are symbols or keywords that perform operations on operands. Operands are the values or variables that the operator acts upon.

1. **Arithmetic Operators**: These operators perform arithmetic operations like addition, subtraction, multiplication, division, etc.

```
x = 10

y = 3

print(x + y)  # Addition: Output = 13

print(x - y)  # Subtraction: Output = 7

print(x * y)  # Multiplication: Output = 30

print(x / y)  # Division: Output = 3.3333...

print(x // y) # Floor Division: Output = 3

print(x % y)  # Modulus: Output = 1

print(x ** y) # Exponentiation: Output = 1000
```

2. **Comparison Operators**: These operators compare the values of two operands and return True or False.

```
x = 10

y = 3

print(x > y)   # Output: True

print(x < y)   # Output: False

print(x == y)  # Output: False

print(x != y)  # Output: True

print(x >= y)  # Output: True

print(x <= y)  # Output: False
```

3. **Logical Operators**: These operators perform logical operations like AND, OR, and NOT.

```
x = True

y = False

print(x and y)  # Output: False
print(x or y)   # Output: True

print(not x)    # Output: False
```

4. Assignment Operators: These operators are used to assign values to variables.

```
x = 10

x += 5   # Equivalent to: x = x + 5

x -= 3   # Equivalent to: x = x - 3

x *= 2   # Equivalent to: x = x * 2

x /= 4   # Equivalent to: x = x / 4
```

5. **Membership Operators**: These operators test for membership in a sequence.

```
my_list = [1, 2, 3, 4, 5]

print(3 in my_list)   # Output: True

print(6 not in my_list)   # Output: True
```

Expressions

In Python, an expression is a combination of values, variables, operators, and function calls that evaluates to a single value. Expressions can be simple or complex, but they always result in a value.

Eg: x = 10 and y = 5

x + y =15 is an expression.

Precedence and Associativity

In Python, precedence and associativity are two concepts that govern the order in which operators are evaluated in expressions.

Precedence determines the order of operations in an expression. Operators with higher precedence are evaluated before operators with lower precedence. For example, multiplication (*) has higher precedence than addition (+), so in the expression `2 + 3 * 4`, the multiplication operation `3 * 4` is evaluated first, resulting in `12`, and then the addition operation `2 + 12` is evaluated, resulting in `14`.

Associativity determines the grouping of operators with the same precedence level. It specifies whether operators are evaluated from left to right (left-associative) or from right to left (right-associative) when they have the same precedence. For example, in the expression `2 - 3 - 4`, the subtraction operations are evaluated from left to right, resulting in `-1 - 4`, and then `-5`.

Non-associative Operators

Non-associative operators are operators for which the order of evaluation cannot be determined solely based on the precedence of the operators.

One example of a non-associative operator is the exponentiation operator (`**`) in Python. Unlike most arithmetic operators, which are left-associative, the exponentiation operator is right-associative. However, it's considered non-associative because multiple exponentiation operations without parentheses cannot be directly evaluated without clarification.

```
result = 2 ** 3 ** 2
```

In this example, the expression `2 ** 3 ** 2` could be interpreted as either `(2 ** 3) ** 2` or `2 ** (3 ** 2)`, resulting in different values (`512` or `64`). To avoid ambiguity, it's recommended to use parentheses to specify the desired order of evaluation:

Using parentheses to specify the order of evaluation

```
result = (2 ** 3) ** 2  # Result: 64
```

Functions

A function in Python is a block of reusable code that performs a specific task. Functions help in modularizing code, making it more readable, maintainable, and reusable.

1. **Built-in Functions**: These functions are pre-defined in Python and are available for use without the need for import statements. Examples include `print()`, `len()`, `max()`, `min()`, etc.

```
  print("Hello, World!")  # Output: Hello, World!
```

2. **User-defined Functions**: These functions are created by the user to encapsulate specific functionality. They start with the `def` keyword followed by the function name, parameters, and the body of the function.

```
  def greet(name):

      return "Hello, " + name + "!"

  print(greet("Alice"))  # Output: Hello, Alice!
```

Modules

A module in Python is a file containing Python definitions, statements, and functions. The file name is the module name with the `.py` extension. Modules help in organizing related code into files, making it easier to manage and reuse.

1. **Built-in Modules**: These are modules that come pre-installed with Python and can be imported and used directly. Examples include `math`, `random`, `os`, `sys`, etc.

```
   import math

   print(math.sqrt(25))   # Output: 5
```

2. **Third-party Modules**: These are modules developed by third-party developers and are not part of the Python standard library. They need to be installed separately using tools like `pip`. Examples include `numpy`, `pandas`, `matplotlib`, etc.

`