Chapter 1

Introduction

Python is a widely used computer programming language that can do a lot of different things like building websites, creating software, automating tasks, and analyzing data. It's not specialized for one specific job, making it versatile. People like using Python because it's easy for beginners to learn.

History of Python

Python, a widely-used general-purpose programming language, was conceived by Guido van Rossum in 1991 and developed by the Python Software Foundation. Known for its emphasis on code readability and concise syntax, Python has become a popular choice among developers.

The language's origin traces back to the late 1980s when Guido Van Rossum initiated its development as a hobby project. Inspired by the ABC Programming Language, he retained its syntax and favorable features while addressing existing flaws. The name "Python" was inspired by the TV show "Monty Python's Flying Circus." Guido served as the "Benevolent Dictator for Life" (BDFL) until stepping down in July 2018. After working at Google, he is currently with Dropbox.

Python was officially released in 1991, offering concise code compared to languages like Java and C++. With a design philosophy focusing on code readability and developer productivity, Python supported classes, inheritance, core data types, exception handling, and functions from the beginning.

The language has two major versions, Python 2.x and 3.x, each with its fanbase. Python finds applications in development, scripting, automation, and software testing. Major tech companies such as Dropbox, Google, Quora, Mozilla, and IBM use Python due to its elegance and simplicity.

Python's journey to becoming the world's most popular coding language spans 30 years. Notably, a recent feature introduced at PyCon22 by the Anaconda Foundation, called "pyscript," allows Python to be written and run in browsers, similar to JavaScript. Despite its age, Python continues to captivate users, with more Google searches than prominent figures like Kim Kardashian and Donald Trump. Its enduring charm and appeal contribute to its ongoing popularity.

Here is a list of major Python versions:

1. Python 1.x Series:
    - Python 1.0 - January 26, 1994
    - Python 1.5 - December 31, 1997

2. Python 2.x Series:
    - Python 2.0 - October 16, 2000
    - Python 2.1 - April 17, 2001
    - Python 2.2 - December 21, 2001
    - Python 2.3 - July 29, 2003
    - Python 2.4 - November 30, 2004
    - Python 2.5 - September 19, 2006
    - Python 2.6 - October 1, 2008
    - Python 2.7 - July 3, 2010

Note: Python 2 reached its end of life on January 1, 2020, and is no longer officially supported.

3. Python 3.x Series:
    - Python 3.0 (also known as Python 3000 or "Py3k") - December 3, 2008
    - Python 3.1 - June 27, 2009
    - Python 3.2 - February 20, 2011
    - Python 3.3 - September 29, 2012
    - Python 3.4 - March 16, 2014
    - Python 3.5 - September 13, 2015
    - Python 3.6 - December 23, 2016
    - Python 3.7 - June 27, 2018
    - Python 3.8 - October 14, 2019
    - Python 3.9 - October 5, 2020
    - Python 3.10 - October 4, 2021
    - Python 3.11 - October 24,2022

Feature of Python

Python is a programming language known for its simplicity and versatility. Here are some key features that contribute to its popularity:

1. Easy to Learn and Use:
    - Python has a straightforward syntax similar to the English language, making it easy to learn.
    - It eliminates the need for semicolons or curly brackets, using indentation to define code blocks.
    - Recommended for beginners due to its simplicity.

2. Expressive Language:
    - Achieves complex tasks with concise code. For example, the "hello world" program can be written in just one line.
    - Requires fewer lines of code compared to languages like Java or C.

3. Interpreted Language:
    - Executes code one line at a time, making debugging easier.
    - Portable across different platforms.

4. Cross-platform Language:
    - Runs on various platforms, including Windows, Linux, UNIX, and Macintosh.
    - Enables the development of software for different platforms with the same codebase.

5. Free and Open Source:
    - Freely available on the official website (www.python.org).
    - Large global community contributes to the development of new modules and functions.
    - Anyone can download and view the source code without cost.

6. Object-Oriented Language:
    - Supports object-oriented programming concepts, including classes, objects, inheritance, polymorphism, and encapsulation.
    - Promotes reusable code and efficient application development.

7. Extensible:
    - Allows integration with languages like C/C++, enabling the use of existing code in Python programs.

8. Large Standard Library:
    - Provides a vast range of libraries for various fields, such as machine learning, web development, and scripting.
    - Popular frameworks include TensorFlow, Pandas, NumPy, Keras, Django, Flask, and Pyramids.

9. GUI Programming Support:
    - Supports the development of graphical user interfaces for desktop applications using libraries like PyQT5, Tkinter, and Kivy.

10. Integrated:
    - Easily integrates with languages like C, C++, and Java, allowing for smooth code execution and debugging.

11. Embeddable:
    - Allows the use of code from other programming languages in Python source code.
    - Python source code can be embedded into other languages.

12. Dynamic Memory Allocation:
    - Does not require specifying variable data types.
    - Allocates memory to variables automatically at runtime, simplifying the coding process.

Chapter 2

Constraint and Variable

The foundation of a programming language lies in its ability to manage and manipulate data elements stored within designated blocks. In this context, distinctive names are assigned to Variables, Constants, and Literals, each serving a specific purpose and offering unique functionalities. This article delves into the realm of Python Constants, Variables, and Literals, providing insights into their types and accompanied by illustrative examples.

Python Variables:

A Variable, within the programming landscape, is a designated location identified by a name to store data dynamically during program execution. In Python, variables serve as containers capable of holding values of any data type. Essentially, when a variable is created, it reserves a portion of memory whose size is determined by the assigned value and data type. Notably, Python employs Identifiers to denote unique names for these variables, following the syntax: **`variable_name = data_values`**. The variable name comprises a combination of letters, numbers, and underscores. An important Python characteristic is the absence of explicit data type declaration; the interpreter dynamically allocates memory based on the assigned value's data type. This flexibility allows for the modification of variable values throughout the program's execution.

Rules to be followed while declaring a Variable name

- A Variable's name cannot begin with a number. Either an alphabet or the underscore character should be used as the first character.
- Variable names are case-sensitive and can include alphanumeric letters as well as the underscore character.
- Variable names cannot contain reserved terms.
- The equal to sign '=', followed by the variable's value, is used to assign variables in Python.

Adhering to certain rules is essential when declaring variable names in Python. Here are the key guidelines:

1. Initial Character:
- The variable name cannot begin with a number. It must start with an alphabet letter or an underscore (`_`).

# Valid variable names
my_variable = 42
_example_variable = "Hello"
# Invalid variable names
1st_variable = 10 # Starts with a number
My variable = 10

2. Case Sensitivity:
   - Variable names in Python are case-sensitive, meaning uppercase and lowercase letters are distinct.

example_variable = 42
Example_Variable = "Hello"

# These are two different variables

3. Character Set:
   - Variable names can include alphanumeric letters (both uppercase and lowercase) as well as the underscore character (`_`).
     my_variable1 = 42
     My_Variable_2 = "Hello"

4. Reserved Terms:
   - Variable names cannot be the same as reserved keywords or terms in Python. For example, you cannot name a variable `if`, `for`, `while`, etc.
     # Invalid variable name
     if = 10 # 'if' is a reserved keyword

5. Assignment Operator:
   - The equal sign (`=`) is used to assign values to variables in Python.
     my_variable = 42

These rules are crucial for maintaining clarity and consistency in your code. Following these guidelines ensures that your variable names are valid and do not conflict with Python's syntax or reserved terms.

Assigning and declaring variable in python

There are few different methods to assign data elements to a Variable. The most common ones are described below

Simple Declaration and Assignment:

- In this straightforward approach, data values are directly assigned to variables in the declaration statement.

```
num = 10
numlist = [1, 3, 5, 7, 9]
string_var = 'Hello World'

print(num)
print(numlist)
print(string_var)
```

Output:

```
10
[1, 3, 5, 7, 9]
Hello World
```

In this example, three variables (`num`, `numlist`, and `string_var`) are created and assigned an integer value, a list of integers, and a string of characters, respectively. The `print` statements showcase the assigned values.

This method provides a clear and concise way to initialize and assign values to variables in Python, making the code easy to read and understand.

Changing the value of a Variable

The values assigned to variables can be changed at any time. It's akin to thinking of a variable as a bag where you can replace items whenever needed. Here's a simple example to illustrate this concept:

```
val = 50
print("Initial value:", val)

val = 100   # assigning a new value
print("Updated value:", val)
```

Output:

```
Initial value: 50
Updated value: 100
```

In this example, the variable `val` is initially assigned a value of 50. Subsequently, it is reassigned a new value of 100. This flexibility allows for the dynamic modification of variable values, providing adaptability and versatility in your Python programs.

Assign multiple values to multiple Variables

Certainly! In Python, you can assign multiple values to multiple variables in a single declaration statement, providing a concise and expressive way to initialize variables. Here's an example:

```
name, age, city = 'David', 27, 'New York'

print(name)
print(age)
print(city)
```

Output:

```
David
27
New York
```

In this example, three variables (`name`, `age`, and `city`) are declared and assigned values in a single line. This method is not only efficient but also enhances code readability. Additionally, you can also assign a single value to multiple variables, as demonstrated in the following example:

```
x = y = z = 10

print(x)
print(y)
print(z)
```

Output:

```
10
10
10
```

Here, the value `10` is assigned to all three variables (`x`, `y`, and `z`) in a single line, showcasing the flexibility and simplicity of Python variable assignments.

Constraint In Python

In Python, a constant is a variable whose value remains unchanged throughout the program's execution. Although Python does not have explicit constant types, a convention is followed to use uppercase letters for variable names that should be treated as constants. Let's explore the concept of Python constants along with the rules for declaring and assigning values to them.

Rules for Declaring Python Constants:

- Constants and variable names in Python should consist of a combination of lowercase (a-z) or uppercase (A-Z) characters, numbers (0-9), or underscores (_).
- Constant names should always be written in UPPERCASE, following a convention like `CONSTANT_NAME`.
- Constant names should not start with digits.
- Except for underscores (_), avoid using additional special characters (e.g., !, #, ^, @, $) when declaring a constant.
- Choose meaningful and descriptive names for constants, promoting code readability.

Example of Assigning Values to Constants:
Constants are typically declared in a separate module and assigned values. Here's an example using a constant module (`constant.py`) and importing it into the main file (`main.py`):

```
# constant.py file
PI = 3.14
GRAVITY = 9.8

# main.py file
import constant as const

print('Value of PI:', const.PI)
print('Value of Gravitational force:', const.GRAVITY)
```

Output:

```
Value of PI: 3.14
Value of Gravitational force: 9.8
```

In this example, `constant.py` contains two constants (`PI` and `GRAVITY`), and these constants are imported and used in the `main.py` file. This approach helps organize constants in a modular way, making the code more maintainable and readable.

Variable as a label

In Python, the term "variable as a label" refers to the concept that a variable is essentially a label or a name given to a memory location where a value is stored. Unlike some other programming languages, Python variables are not containers that hold values but rather references or labels that point to an object in memory.

Consider the following example:
x = 10

In this case, `x` is a variable, but more accurately, it is a label pointing to the memory location where the value `10` is stored. If you later assign a different value to `x`, the label is simply redirected to the new memory location.

x = 20

Now, `x` is a label pointing to a different memory location where the value `20` is stored. This dynamic nature of variables in Python allows for flexibility but also requires an understanding of how the labels work.

This concept becomes more evident when dealing with mutable objects like lists:
list1 = [1, 2, 3]
list2 = list1

Here, both `list1` and `list2` are labels pointing to the same list object in memory. If you modify the list through one label, the changes are reflected when accessed through the other label.

list1.append(4)
print(list2) # Output: [1, 2, 3, 4]

Understanding variables as labels is crucial for writing Python code effectively, especially when working with complex data structures and mutable objects.

Data type in python:

1. String

   A String in Python is a sequence of characters and is immutable, meaning it cannot be changed after creation. Strings are versatile and widely used for storing and manipulating text data, such as names, addresses, and more.

   Creating a String:

Strings in Python can be created using single quotes (`'`), double quotes (`"`), or triple quotes (`'''` or `"""`). Triple quotes are particularly useful for creating multiline strings.

```
# Creating a String with single Quotes
string_single = 'Hello World'

# Creating a String with double Quotes
string_double = "I'm a Developer"

# Creating a String with triple Quotes
string_triple = '''I'm a Python Developer "KCC"'''

# Creating a multiline String with triple Quotes
multiline_string = '''Our
                Country is
                Nepal'''
```

Accessing Characters in a String:
In Python, individual characters in a string can be accessed using indexing. Negative indices refer to characters from the end of the string.

```
first_char = string_single[0]      # Accessing the first character
last_char = string_single[-1]      # Accessing the last character
```

Remember that attempting to access an index outside the range will result in an `IndexError`. Only integers are allowed as indices, and using other types will cause a `TypeError`.

2. Integers

In Python, an integer is a data type that represents whole numbers without any decimal points. Integers can be positive or negative, and they have no upper or lower limit (except for practical constraints like available memory). The `int` type in Python is used to represent integers.

Here are some examples of integers in Python:

```
a = 5       # positive integer
b = -3      # negative integer
c = 0       # zero

# Integers can be used in various operations
sum_result = a + b
```

```
product_result = a * b
```

In Python, integers can be used in arithmetic operations like addition, subtraction, multiplication, and division. If a division operation is performed between two integers in Python 3, it will return a floating-point result, even if the division is exact.

```
division_result = a / b  # Result will be a float in Python 3
```

Python 2 behaved differently in this regard, where division of integers resulted in an integer value (truncating the decimal part). In Python 3, if you want integer division (floor division), you can use the `//` operator:

```
integer_division_result = a // b  # Result will be an integer
```

Integers in Python are very flexible and are used extensively in various mathematical and computational tasks.

3. Floats

In Python, a float is a data type that represents real numbers (numbers with decimal points or exponential notation). The `float` type is used to represent floating-point numbers.

Here are some examples of floats in Python:

```
x = 3.14      # a simple float
y = -0.5      # a negative float
z = 2.0e3     # a float using exponential notation (2.0 * 10^3)

# Floats can be used in various operations
sum_result = x + y
product_result = x * z
```

Floats in Python can be used in arithmetic operations similar to integers. However, it's important to be aware of certain behaviors associated with floating-point arithmetic, such as precision limitations.

```
division_result = x / y  # Result will be a float

# Beware of precision limitations
print(0.1 + 0.2)  # Result might not be exactly 0.3 due to floating-point representation
```

Floating-point numbers are represented using binary fractions, which can sometimes result in small inaccuracies due to the limitations of computer hardware.

Floats are particularly useful when dealing with mathematical operations that involve non-integer numbers, such as scientific calculations or any scenario where precision is important.

It's also worth noting that in Python, the `int` type and `float` type are distinct. If you perform an operation that involves both an `int` and a `float`, the result will be a `float`.
result = 5 + 2.0 # Result will be a float

Understanding the characteristics of floating-point numbers is crucial when working with numerical data in Python to avoid unexpected behavior due to precision issues.

Operators in python:

Operator a use to perform operation on value and variable

1. **Arithmetic Operator**:

   Use to perform common mathematical operation
   Certainly! Here's a compressed overview of arithmetic operators in Python:

   - Addition (`+`):Adds two operands.
     result = 5 + 3   # Result: 8
   - Subtraction (`-`): Subtracts the right operand from the left operand.
     result = 7 - 2   # Result: 5
   - Multiplication (`*`): Multiplies two operands.
     *result = 4 * 6   # Result: 24*

   - Division (`/`): Divides the left operand by the right operand (result is a float).
     result = 10 / 2   # Result: 5.0
   - Floor Division (`//`): Divides with the result as the largest integer less than or equal to the division.
     result = 10 // 3   # Result: 3
   - Modulus (`%`): Returns the remainder of the division.
     result = 10 % 3   # Result: 1
   - Exponentiation (`**`): Raises the left operand to the power of the right operand.
     *result = 2 ** 3   # Result: 8*

Notes
Remember, parentheses can be used to control the order of operations.
*result = (4 + 2) * 3   # Result: 18*

2. Assignment Operator:

In Python, the assignment operator (`=`) is used to give a variable a value. For example:
x = 5   # x now holds the value 5
name = "John" # name now holds the string "John"

You can also use compound assignment for shortcuts:
count = 10
count += 5   # Increment count by 5

Other compound assignment operators include `-=`, `*=`, `/=`, `//=`, and `%=` for subtraction, multiplication, division, floor division, and modulus, respectively.

3. **Comparison Operator:**

Comparison operators in Python help compare values and produce Boolean results:

1. Equal to (`==`): Checks if values are equal.
  result = 5 == 5   # True

2. Not equal to (`!=`): Checks if values are not equal.
  result = 5 != 3   # True

3. Greater than (`>`): Checks if the left value is greater.
  result = 7 > 4   # True

4. Less than (`<`): Checks if the left value is less.
  result = 2 < 5   # True

5. Greater than or equal to (`>=`): Checks if the left value is greater than or equal to the right value.
  result = 6 >= 6   # True

6. Less than or equal to (`<=`): Checks if the left value is less than or equal to the right value.
  result = 3 <= 3   # True

These operators are essential for making decisions in Python programs based on the relationships between values.

4. **Logical Operator:**

Logical operators in Python help combine and evaluate conditions:

1. Logical AND (`and`): Returns `True` if both conditions are true.
   result = (5 > 3) and (7 < 10)   # True

2. Logical OR (`or`): Returns `True` if at least one condition is true.
   result = (5 > 3) or (7 < 5)   # True

3. Logical NOT (`not`): Returns the opposite of the condition.
   result = not (5 > 3)   # False

These operators are useful for combining and negating conditions to make decisions in Python programs.

5. Identity Operator

In Python, identity operators check if two variables refer to the same object:

1. is: Returns `True` if both variables share the same identity.
   x = [1, 2, 3]
   y = [1, 2, 3]
   result = x is y   # False (different objects)

2. is not: Returns `True` if both variables do not share the same identity.
   a = "hello"
   b = "hello"
   result = a is not b   # False (Same objects)

These operators compare the memory location of objects, not their values.

6. **Membership Operator**:

In Python, membership operators check if a value is in a sequence:

1. in: Returns `True` if the value is in the sequence.
   result = 3 in [1, 2, 3, 4]   # True

2. not in: Returns `True` if the value is not in the sequence.

```
result = "z" not in "hello"   # True
```
These operators are handy for checking membership in lists, strings, or other iterable types.

7. Bitwise Operator:

In Python, bitwise operators manipulate individual bits:

1. AND (`&`): Sets each bit to 1 if both are 1.
```
result = 5 & 3   # 1
```

2. OR (`|`): Sets each bit to 1 if at least one is 1.
```
result = 5 | 3   # 7
```

3. XOR (`^`): Sets each bit to 1 if only one is 1.
```
result = 5 ^ 3   # 6
```

4. NOT (`~`): Flips the bits.
```
result = ~5   # -6
```

5. Left Shift (`<<`): Shifts bits to the left.
```
result = 5 << 1   # 10
```

6. Right Shift (`>>`): Shifts bits to the right.
```
result = 5 >> 1   # 2
```

Bitwise operators are used for low-level bit manipulation in integer values.

Comment In Python

In Python, comments are used to provide explanatory notes within the code. Comments are ignored by the Python interpreter during the execution of the program. There are two types of comments in Python:

1.Single-line Comments:

- Created using the `#` symbol.
```
# This is a single-line comment
x = 5  # This is a comment at the end of a line
```

2. Multi-line Comments:

- Created using triple-quotes (`'''` or `"""`).

```
'''
This is a multi-line comment.
It can span multiple lines.
'''
"""
Another way to create a multi-line comment.
"""
```

Comments are valuable for explaining the purpose of code, documenting functionality, or temporarily disabling code for testing purposes. They contribute to code readability and understanding.

Indentation

In Python, indentation is used to define the structure and scope of code blocks. Unlike many other programming languages that use braces `{}` or keywords like `begin` and `end` to delimit blocks, Python relies on indentation to indicate the beginning and end of blocks of code. Here are the key points about indentation in Python:

1. Indentation Level:
- Indentation is typically done using four spaces for each level of indentation. However, the important thing is to be consistent with the number of spaces used throughout the code.

2. Indentation for Code Blocks:
- Indentation is used to define the scope of control flow statements (like `if`, `for`, `while`) and function or class definitions.

```
if condition:
    # Code inside the if block
    print("Condition is true")

for item in iterable:
    # Code inside the for loop
    print(item)
```

3. Consistent Indentation:
- All lines within the same block must have the same level of indentation. Inconsistent indentation will result in a `IndentationError`.

```
# Incorrect indentation, will result in an error
if condition:
print("Indented incorrectly")
```

4. Blank Lines:
- Blank lines are used to separate blocks of code but do not influence the indentation level.

```
def function_one():
    # Code for function_one
    pass

# Blank line separating function_one and function_two
def function_two():
    # Code for function_two
    pass
```

5. No Need for End Delimiters:
  - Python does not use braces or keywords to indicate the end of a block. The end of a block is determined by a decrease in indentation.

```
if condition:
    # Code inside the if block
    print("Condition is true")
# End of the if block (due to the decrease in indentation)
```

Consistent and readable indentation is crucial in Python for maintaining code clarity and ensuring that the code is correctly interpreted by the Python interpreter.

Input and output in python

In Python, input and output operations are crucial for interacting with the user and displaying results. Here's a brief overview:
1. `input()` Function:
  - Used to take user input from the console.
  - Returns the input as a string.

```
name = input("Enter your name: ")
age = input("Enter your age: ")
```

2. Type Conversion for Numeric Input:
  - If you need numeric input, you may need to convert the string input to the desired numeric type.

```
age = int(input("Enter your age: "))
```

Output in Python:

1. `print()` Function:
  - Used to display output on the console.
  - Can print multiple values separated by commas.

```
print("Hello, " + name + "! You are", age, "years old.")
```

2. Formatted Output:
- Using f-strings (formatted string literals) for more readable output.

```
print(f"Hello, {name}! You are {age} years old.")
```

3. `sep` and `end` Parameters:
- `sep` specifies the separator between values in `print()`.
- `end` specifies the character at the end of the line.

```
print("Hello", "World", sep=", ", end="!")
```

4. Formatted Output with `format()`:
- Another way to format strings using the `format()` method.

```
print("Hello, {}! You are {} years old.".format(name, age))
```

Input and output operations are essential for building interactive programs, and understanding how to use `input()` and `print()` effectively is fundamental in Python programming.

Styling Your Code:

Style guidelines in Python help maintain consistency in code, making it more readable and understandable. The most widely followed style guide for Python is PEP 8 (Python Enhancement Proposal 8). Here are some key points from PEP 8:

1. Indentation:
- Use 4 spaces per indentation level.
- Avoid tabs.

```
# Good
def function():
    if x:
        do_something()

# Bad (avoid tab is use)
def function():
	if x:
		do_something()
```

2. Maximum Line Length:
- Limit all lines to a maximum of 79 characters for code and 72 for docstrings.
- For docstrings or comments, limit lines to 72 characters.

```
# Good
```

```
def function_with_long_name(
        parameter1, parameter2, parameter3, parameter4):
    # function body

# Bad
def function_with_long_name(parameter1, parameter2, parameter3, parameter4):
  # More  than 79 characters
```

3. Imports:
- Imports should usually be on separate lines.
- Wildcard imports (`from module import *`) are strongly discouraged.

```
# Good
import module
from module import function

# Bad
import module, another_module  # Multiple imports on one line
from module import *  # Wildcard import
```

4. Whitespace in Expressions and Statements:
- Avoid extraneous whitespace in the following situations:
  - Immediately inside parentheses, brackets, or braces.
  - Immediately before a comma, semicolon, or colon.
  - Immediately before the open parenthesis that starts the argument list of a function call.

```
# Good
spam(ham[1], {eggs: 2})
if x == 4: print(x, y); x, y = y, x

# Bad
spam( ham[ 1 ], { eggs: 2 } )  # Extraneous whitespace
```

5. Comments:
- Comments should be complete sentences.
- Use two spaces after a sentence-ending period in multi-sentence comments.

```
# Good
# This is a good comment

# Bad
#badcomment
```

6. Function and Variable Names:

- Function names should be lowercase, with words separated by underscores.
- Variable names follow the same convention.

```
# Good
def my_function():
    my_variable = 42

# Bad
def MyFunction():  # Camel case
    MyVariable = 42  # Camel case
```

7. Whitespace in Expressions and Statements:

- Avoid extraneous whitespace in the following situations:
  - Immediately inside parentheses, brackets, or braces.
  - Immediately before a comma, semicolon, or colon.
  - Immediately before the open parenthesis that starts the argument list of a function call.

```
# Good
spam(ham[1], {eggs: 2})
if x == 4: print(x, y); x, y = y, x

# Bad
spam( ham[ 1 ], { eggs: 2 } )  # Extraneous whitespace
```

Chapter 3

If Statement:

The `if` statement in Python is a conditional statement that allows you to execute a block of code if a specified condition is `True`. Here's an overview of its syntax and usage:

Syntax:

```
if condition:
    # Code block executed if the condition is True
    statement1
    statement2
    # ...
```
Example:
```
x = 5
if x > 0:
    print("x is positive")
```

In this example, the `if` statement checks whether the variable `x` is greater than 0. If the condition is `True`, the block of code inside the `if` block is executed. If the condition is `False`, the block of code is skipped, and the program continues with the next statement after the `if` block.

The `if` statement is fundamental for introducing conditional logic in Python, allowing you to control the flow of your program based on specific conditions.

`if-else` Statement:

The `if-else` statement in Python is used for conditional execution. It allows you to specify two blocks of code: one to be executed if a specified condition is `True`, and another to be executed if the condition is `False`. Here's an overview:

Syntax:
```
if condition:
    # Code block executed if the condition is True
    statement1
    statement2
    # ...

else:
    # Code block executed if the condition is False
    statement3
    statement4
    # ...
```

Example:
```
x = -2
if x > 0:
    print("x is positive")
```

```
else:
    print("x is non-positive")
```

In this example, the `if` statement checks whether the variable `x` is greater than 0. If the condition is `True`, the block of code inside the `if` block is executed. If the condition is `False`, the block of code inside the `else` block is executed.

The `if-else` statement is fundamental for implementing branching logic in Python, providing the ability to handle different scenarios based on the evaluation of a condition.

`if-elif-else` Statement:

The `elif` (short for "else if") statement in Python is used in conjunction with the `if` statement to handle multiple conditions sequentially. It allows you to test additional conditions if the previous `if` or `elif` conditions are not satisfied. Here's an overview of its syntax and usage:

Syntax:

```
if condition1:
    # Code block executed if condition1 is True
    statement1
    statement2
elif condition2:
    # Code block executed if condition1 is False and condition2 is True
    statement3
    statement4
elif condition3:
    # Code block executed if both condition1 and condition2 are False, and condition3 is True
    statement5
    statement6
else:
    # Code block executed if none of the above conditions are True
    statement7
    statement8
```

Example:

```
score = 85

if score >= 90:
    print("A")
elif 80 <= score < 90:
    print("B")
elif 70 <= score < 80:
    print("C")
```

```
else:
    print("F")
```

In this example, the `elif` statements provide additional conditions to be checked sequentially. If the first condition (`score >= 90`) is not satisfied, it moves to the next `elif` statement (`80 <= score < 90`), and so on. If none of the conditions is met, the code block under `else` is executed.

The `elif` statement is valuable for handling multiple cases in a structured and readable way, avoiding nested `if` statements.

Switch Case:

Certainly! The new `match` statement introduced in Python 3.10 provides a clean and concise way to implement switch-like behavior. Here's an easy explanation:

```
lang = input("What's the programming language you want to learn? ")

match lang:
    case "JavaScript":
        print("You can become a web developer.")
    case "Python":
        print("You can become a Data Scientist")
    case "PHP":
        print("You can become a backend developer")
    case "Solidity":
        print("You can become a Blockchain developer")
    case "Java":
        print("You can become a mobile app developer")
    case _:
        print("The language doesn't matter, what matters is solving problems.")
```

With the `match` statement, you can easily express a switch-like structure. The underscore (`_`) serves as a wildcard for the default case, and you no longer need explicit `break` statements as it's handled automatically. This leads to cleaner and more readable code compared to multiple `elif` statements.