

# PYTHON PROGRAMMING

---

NOTES

Harsh Choudhary

# PYTHON PROGRAMMING

## Getting Started with Python Programming

**Python** is a versatile, interpreted programming language celebrated for its simplicity and readability. This guide will walk us through installing Python, running first program and exploring interactive coding—all essential steps for beginners.

## Install Python

Before starting this Python course first, you need to install Python on your computer. Most systems come with Python pre-installed. To verify if Python is available on your computer, simply open command line interface (Command Prompt on Windows or Terminal on macOS/Linux) and type:

```
python --version
```

If Python is installed, this command will display its version but if it is not installed then to install Python on our computer, follow these steps:

- **Download Python:** Go to the official Python website at <https://www.python.org/> On the homepage, we will see a "Downloads" section. Click on the "Download Python" button.



- **Choose the Version:** We will be directed to a page where we can choose the version of Python we want to download. Python usually has two main versions available: Python 3. Python 3 is the recommended version. Click on the appropriate version for your operating system (Windows, macOS, or Linux).
- **Add Python to PATH (Optional):** On Windows, we may be given the option to add Python to our system's PATH environment variable. This makes it easier to run Python from the command line. If you're not sure, it's usually safe to select this option.
- **Install Python:** Click the "Install Now" button to begin the installation. The installer will copy the necessary files to our computer.

- **Verify the Installation:** After the installation is complete, we can verify if Python was installed correctly by opening cmd (on Windows) or terminal (on macOS or Linux). Type: `python --version`. This should display the version of Python we installed.

### *Step By Step Installation Guide:*

- [Install Python on Windows](#)
- [Install Python on Linux](#)
- [Install Python on MacOS](#)

## **Create and Run your First Python Program on Terminal**

Once you have Python installed, you can run the program by following these steps:

1. Open a text editor (e.g., Notepad on Windows, TextEdit on macOS, or any code editor like VS Code, PyCharm, etc.).
2. Copy the code:- `print('Hello World')` above and paste it into the text editor.
3. Save the file with .py extension (e.g., Hello.py).
4. Open the terminal.
5. Run the program by pressing Enter.

## **Using Python's Interactive Shell**

For quick experiments, use Python's interactive shell:

### **1. Launch the Shell:**

```
python
```

or, if required:

```
python3
```

### **2. Enter Commands Directly:**

For example:

```
>>> print("Hello, World!")
```

### **3. Exit the Shell:**

```
exit()
```

## **Next Steps**

With Python installed and your first script running, continue your journey by exploring:

- [Variables and Data Types](#)
- [Loops and Conditional Statements](#)
- [Functions and Modules](#)

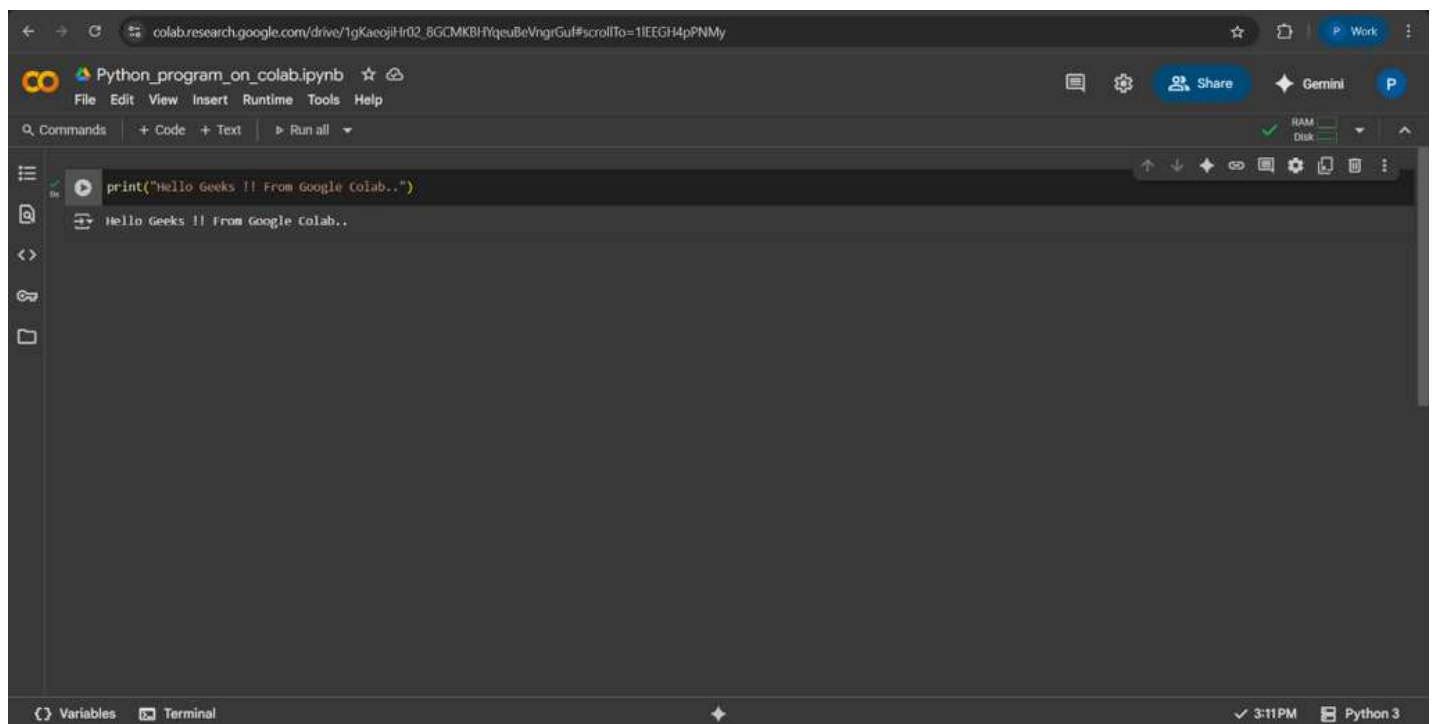
## Using Python on Google Colab

Google Colab is a popular cloud-based platform that provides an interactive Python environment for running Python code. It's especially useful for data science, machine learning, and educational purposes, as it allows you to write and execute Python code in the browser without installing anything locally.

### Steps to Get Started on Google Colab:

1. **Open Google Colab:** Go to Google Colab. You will need a Google account to access it.
2. **Create a New Notebook:** After logging into your Google account, you can create a new notebook by selecting File > New Notebook. This will create a fresh Python environment where you can start coding.
3. **Running Python Code:** In Colab, you can directly write Python code in cells and run them interactively. To run the code in a cell, click on the play button next to the cell or press Shift + Enter.

### Example:



The above image shows how easily we can work with Python on Google colab.

## Input and Output in Python

Understanding input and output operations is fundamental to Python programming. With the `print()` function, we can display output in various formats, while the `input()` function enables interaction with users by gathering input during program execution.

### Taking input in Python

Python's **`input()` function** is used to take user input. By default, it returns the user input in form of a string.

### Example:

```
name = input("Enter your name: ")  
  
print("Hello,", name, "! Welcome!")
```

## Output

*Enter your name: Data science*

*Hello, Data science ! Welcome!*

The code prompts the user to input their name, stores it in the variable "name" and then prints a greeting message addressing the user by their entered name.

## Printing Output using print() in Python

At its core, printing output in Python is straightforward, thanks to the print() function. This function allows us to display text, variables and expressions on the console. Let's begin with the basic usage of the print() function:

In this example, "Hello, World!" is a string literal enclosed within double quotes. When executed, this statement will output the text to the console.

```
print("Hello, World!")
```

## Output

*Hello, World!*

## Printing Variables

We can use the print() function to print single and multiple variables. We can print multiple variables by separating them with commas.

### Example:

```
# Single variable
```

```
s = "Bob"
```

```
print(s)
```

```
# Multiple Variables
```

```
s = "Alice"
```

```
age = 25
```

```
city = "New York"
```

```
print(s, age, city)
```

## Output

*Bob*

## Take Multiple Input in Python

We are taking multiple input from the user in a single line, splitting the values entered by the user into separate variables for each value using the [split\(\) method](#). Then, it prints the values with corresponding labels, either two or three, based on the number of inputs provided by the user.

*# taking two inputs at a time*

```
x, y = input("Enter two values: ").split()
```

```
print("Number of boys: ", x)
```

```
print("Number of girls: ", y)
```

*# taking three inputs at a time*

```
x, y, z = input("Enter three values: ").split()
```

```
print("Total number of students: ", x)
```

```
print("Number of boys is : ", y)
```

```
print("Number of girls is : ", z)
```

### Output

*Enter two values: 5 10*

*Number of boys: 5*

*Number of girls: 10*

*Enter three values: 5 10 15*

*Total number of students: 5*

*Number of boys is : 10*

*Number of girls is : 15*

## How to Change the Type of Input in Python

By default input() function helps in taking user input as string. If any user wants to take input as int or float, we just need to [typecast](#) it.

### Print Names in Python

The code prompts the user to input a string (the color of a rose), assigns it to the variable color and then prints the inputted color.

*# Taking input as string*

```
color = input("What color is rose?: ")
```

```
print(color)
```

### Output

*What color is rose?: Red*

*Red*

### **Print Numbers in Python**

The code prompts the user to input an integer representing the number of roses, converts the input to an integer using typecasting and then prints the integer value.

### **Print Float/Decimal Number in Python**

The code prompts the user to input the price of each rose as a floating-point number, converts the input to a float using typecasting and then prints the price.

```
# Taking input as float
```

```
# Typecasting to float
```

```
price = float(input("Price of each rose?: "))
```

```
print(price)
```

### **Output**

*Price of each rose?: 50.3050.3*

### **Find DataType of Input in Python**

In the given example, we are printing the type of variable x. We will determine the type of an object in Python.

```
a = "Hello World"
```

```
b = 10
```

```
c = 11.22
```

```
d = ("Tech", "for", "Tech")
```

```
e = ["Tech", "for", "Tech"]
```

```
f = {"Tech": 1, "for":2, "Tech":3}
```

```
print(type(a))
```

```
print(type(b))
```

```
print(type(c))
```

```
print(type(d))
```

```
print(type(e))
```

```
print(type(f))
```

### **Output**

```
<class 'str'>
```

```
<class 'int'>
```

```
<class 'float'>
```

```
<class 'tuple'>
```

```
<class 'list'>
```

```
<class 'dict'>
```

## Output Formatting

Output formatting in Python with various techniques including the `format()` method, manipulation of the `sep` and `end` parameters, f-strings and the versatile `%` operator. These methods enable precise control over how data is displayed, enhancing the readability and effectiveness of your Python programs.

### Example 1: Using Format()

```
amount = 150.75  
  
print("Amount: ${:.2f}".format(amount))
```

### Output

Amount: \$150.75

### Example 2: Using sep and end parameter

```
# end Parameter with '@'  
print("Python", end='@')  
print("hello")
```

```
# Seprating with Comma  
print('G', 'F', 'G', sep='')
```

```
# for formatting a date  
print('09', '12', '2016', sep='-')
```

```
# another example  
print('pratik', 'hello', sep='@')
```

### Output

Python@hello

GFG

09-12-2016

*pratik@hello*

### **Example 3: Using f-string**

```
name = 'Tushar'
```

```
age = 23
```

```
print(f"Hello, My name is {name} and I'm {age} years old.")
```

### **Output**

*Hello, My name is Tushar and I'm 23 years old.*

### **Example 4: Using % Operator**

We can use '%' operator. % values are replaced with zero or more value of elements. The formatting using % is similar to that of 'printf' in the C programming language.

- %d –integer
- %f – float
- %s – string
- %x –hexadecimal
- %o – octal

```
# Taking input from the user
```

```
num = int(input("Enter a value: "))
```

```
add = num + 5
```

```
# Output
```

```
print("The sum is %d" %add)
```

### **Output**

*Enter a value: 50The sum is 55*

### **Taking Conditional User Input in Python**

In Python, taking conditional user input means getting input from the user and making decisions based on that input. You usually use the input() function to get the value and then use if-else statements to check conditions.

- input() is used to take user input as a string.
- You can use int() or float() to convert it if needed.
- Use if, elif, and else to apply conditions to the input.

## Python Variables

In [Python](#), variables are used to store data that can be referenced and manipulated during program execution. A variable is essentially a name that is assigned to a value. Unlike many other programming languages, Python variables do not require explicit declaration of type. The type of the variable is inferred based on the value assigned.

Variables act as placeholders for data. They allow us to store and reuse values in our program.

### Example:

```
# Variable 'x' stores the integer value 10
```

```
x = 5
```

```
# Variable 'name' stores the string "Samantha"
```

```
name = "Samantha"
```

```
print(x)
```

```
print(name)
```

### Output

```
5
```

```
Samantha
```

In this article, we'll explore the concept of variables in Python, including their syntax, characteristics and common operations.

### Table of Content

- [Rules for Naming Variables](#)
- [Assigning Values to Variables](#)
- [Multiple Assignments](#)
- [Type Casting a Variable](#)
- [Getting the Type of Variable](#)
- [Object Reference in Python](#)
- [Delete a Variable Using del Keyword](#)

### Rules for Naming Variables

To use variables effectively, we must follow Python's naming rules:

- Variable names can only contain letters, digits and underscores (`_`).
- A variable name cannot start with a digit.
- Variable names are case-sensitive (`myVar` and `myvar` are different).
- Avoid using [Python keywords](#) (e.g., `if`, `else`, `for`) as variable names.

### Valid Example:

```
age = 21
```

```
_colour = "lilac"
```

```
total_score = 90
```

### Invalid Example:

```
1name = "Error" # Starts with a digit
class = 10      # 'class' is a reserved keyword
user-name = "Doe" # Contains a hyphen
```

## Assigning Values to Variables

### Basic Assignment

Variables in Python are assigned values using the = [operator](#).

```
x = 5
y = 3.14
z = "Hi"
```

### Dynamic Typing

Python variables are dynamically typed, meaning the same variable can hold different types of values during execution.

```
x = 10
x = "Now a string"
```

### Multiple Assignments

Python allows multiple variables to be assigned values in a single line.

### Assigning the Same Value

Python allows assigning the same value to multiple variables in a single line, which can be useful for initializing variables with the same value.

```
a = b = c = 100
print(a, b, c)
```

### Output

```
100 100 100
```

### Assigning Different Values

We can assign different values to multiple variables simultaneously, making the code concise and easier to read.

```
x, y, z = 1, 2.5, "Python"
print(x, y, z)
```

### Output

```
1 2.5 Python
```

## Type Casting a Variable

[Type casting](#) refers to the process of converting the value of one data type into another. Python provides several built-in functions to facilitate casting, including `int()`, `float()` and `str()` among others.

### Basic Casting Functions

- **`int()`** - Converts compatible values to an integer.
- **`float()`** - Transforms values into floating-point numbers.
- **`str()`** - Converts any data type into a string.

### Examples of Casting:

```
# Casting variables
s = "10" # Initially a string
n = int(s) # Cast string to integer
cnt = 5
f = float(cnt) # Cast integer to float
age = 25
s2 = str(age) # Cast integer to string
```

#### # Display results

```
print(n)
print(f)
print(s2)
```

### Output

```
10
5.0
25
```

## Getting the Type of Variable

In Python, we can determine the type of a variable using the `type()` function. This built-in function returns the type of the object passed to it.

### Example Usage of `type()`

```
# Define variables with different data types
n = 42
f = 3.14
s = "Hello, World!"
li = [1, 2, 3]
d = {'key': 'value'}
bool = True
```

```
# Get and print the type of each variable
print(type(n))
```

```
print(type(f))
print(type(s))
print(type(li))
print(type(d))
print(type(bool))
```

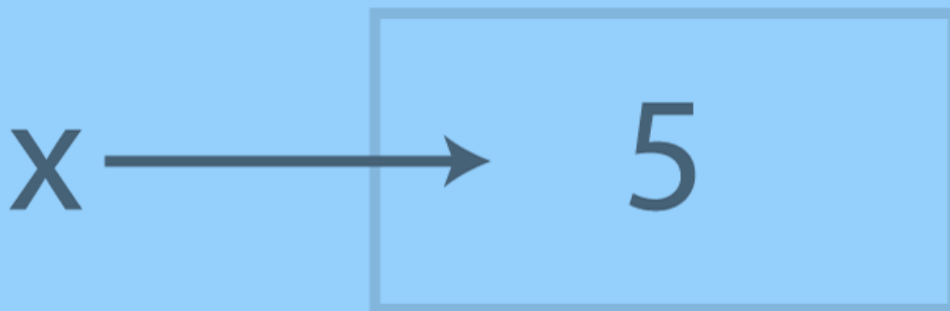
### Output

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'list'>
<class 'dict'>
<class 'bool'>
```

### Object Reference in Python

Let us assign a variable `x` to value 5.

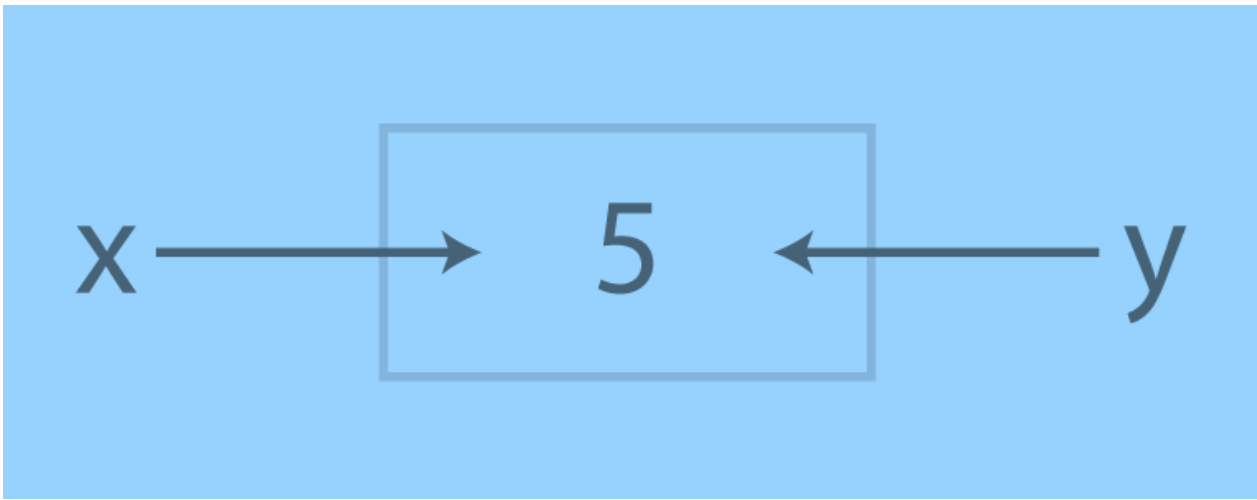
```
x = 5
```



When `x = 5` is executed, Python creates an object to represent the value `5` and makes `x` reference this object.

Now, if we assign another variable `y` to the variable `x`.

```
y = x
```



### Explanation:

- Python encounters the first statement, it creates an object for the value `5` and makes `x` reference it. The second statement creates `y` and references the same object as `x`, not `x` itself. This is called a *Shared Reference*, where multiple variables reference the same object.

### Delete a Variable Using del Keyword

We can remove a variable from the namespace using the [del](#) keyword. This effectively deletes the variable and frees up the memory it was using.

### Example:

```
# Assigning value to variable
```

```
x = 10
```

```
print(x)
```

```
# Removing the variable using del
```

```
del x
```

```
# Trying to print x after deletion will raise an error
```

```
# print(x) # Uncommenting this line will raise NameError: name 'x' is not defined
```

### Explanation:

- `del x` removes the variable `x` from memory.
- After deletion, trying to access the variable `x` results in a `NameError`, indicating that the variable no longer exists.

### Practical Examples

#### 1. Swapping Two Variables

Using multiple assignments, we can swap the values of two variables without needing a temporary variable.

```
a, b = 5, 10
```

```
a, b = b, a
print(a, b)
```

### Output

```
10 5
```

## 2. Counting Characters in a String

Assign the results of multiple operations on a string to variables in one line.

```
word = "Python"
length = len(word)
print("Length of the word:", length)
```

### Output

```
Length of the word: 6
```

## Scope of a Variable

In Python, the scope of a variable defines where it can be accessed in the program. There are two main types of scope: [local and global](#).

### Local Variables:

- Defined within a function or block, accessible only inside that scope.
- Destroyed once the function/block ends.
- Temporary, used for short-term data.

### Global Variables:

- Defined outside functions, accessible throughout the program.
- To modify within a function, use the [global keyword](#).
- Persist in memory for the program's duration, useful for shared data.

*To learn about it in detail, refer to [local and global](#).*

## Python Variables - Python

### What is the scope of a variable in Python?

*The scope of a variable determines where it can be accessed. Local variables are scoped to the function in which they are defined, while global variables can be accessed throughout the program.*

### Can we change the type of a variable after assigning it?

*Yes, Python allows dynamic typing. A variable can hold a value of one type initially and be reassigned a value of a different type later.*

### What happens if we use an undefined variable?

Using an undefined variable raises a `NameError`. Always initialize variables before use.

## How can we delete a variable in Python?

We can delete a variable in Python using the `del` keyword:

```
x = 10
del x
#print(x)  # Raises a NameError since 'x' has been deleted
```

## Python Keywords

Keywords in Python are reserved words that have special meanings and serve specific purposes in the language syntax. Python keywords cannot be used as the names of variables, functions, and classes or any other identifier.

## Getting List of all Python keywords

We can also get all the keyword names using the below code.

```
import keyword

# printing all keywords at once using "kwlist()"
print("The list of keywords is : ")
print(keyword.kwlist)
```

**Output:**

```
The list of keywords are:
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

## How to Identify Python Keywords ?

- **With Syntax Highlighting** - Most of IDEs provide syntax-highlight feature. You can see Keywords appearing in different color or style.
- **Look for SyntaxError** - This error will encounter if you have used any keyword incorrectly. Note that keywords can not be used as identifiers (variable or a function name).

## What Happens if We Use Keywords as Variable Names ?

In Python, keywords are reserved words that have special meanings and cannot be used as variable names. If you attempt to use a keyword as a variable, Python will raise a **SyntaxError**. Let's look at an example:

```
for = 10
print(for)
```

**Output**

```
Hangup (SIGHUP)
File      "/home/guest/sandbox/Solution.py",      line      1
for      =                                         10
^
SyntaxError: invalid syntax
```

Let's categorize all keywords based on context for a clearer understanding.

Category	Keywords
Value Keywords	<u>True</u> , <u>False</u> , <u>None</u>
Operator Keywords	<u>and</u> , <u>or</u> , <u>not</u> , <u>is</u> , <u>in</u>
Control Flow Keywords	<u>if</u> , <u>else</u> , <u>elif</u> , <u>for</u> , <u>while</u> , <u>break</u> , <u>continue</u> , <u>pass</u> , <u>try</u> , <u>except</u> , <u>finally</u> , <u>raise</u> , <u>assert</u>
Function and Class	<u>def</u> , <u>return</u> , <u>lambda</u> , <u>yield</u> , <u>class</u>
Context Management	<u>with</u> , <u>as</u>
Import and Module	<u>import</u> , <u>from</u>
Scope and Namespace	<u>global</u> , <u>nonlocal</u>
Async Programming	<u>async</u> , <u>await</u>

## Difference Between Keywords and Identifiers

Keywords	Identifiers
Reserved words in Python that have a specific meaning.	Names given to variables, functions, classes, etc.
Cannot be used as variable names.	Can be used as variable names (if not a keyword).
Examples: if, else, for, while	Examples: x, number, sum, result
Part of the Python syntax.	User-defined, meaningful names in the code.
They cannot be redefined or changed.	Can be defined and redefined by the programmer.

## Difference Between Variables and Keywords

Variables	Keywords
Used to store data.	Reserved words with predefined meanings in Python.
Can be created, modified, and deleted by the programmer.	Cannot be modified or used as variable names.
Examples: x, age, name	Examples: if, while, for
Hold values that are manipulated in the program.	Used to define the structure of Python code.
Variable names must follow naming rules but are otherwise flexible.	Fixed by Python language and cannot be altered.

## Python Operators

In Python programming, Operators in general are used to perform operations on values and variables. These are standard symbols used for logical and arithmetic operations. In this article, we will look into different types of **Python operators**.

- **OPERATORS:** These are the special symbols. Eg- + , \* , / , etc.
- **OPERAND:** It is the value on which the operator is applied.

## Types of Operators in Python

### Arithmetic Operators in Python

Python [Arithmetic operators](#) are used to perform basic mathematical operations like **addition**, **subtraction**, **multiplication** and **division**.

In Python 3.x the result of division is a floating-point while in Python 2.x division of 2 integers was an integer. To obtain an integer result in Python 3.x floored (`// integer`) is used.

### Example of Arithmetic Operators in Python:

```
# Variables
```

```
a = 15
```

```
b = 4
```

```
# Addition
```

```
print("Addition:", a + b)
```

```
# Subtraction
```

```
print("Subtraction:", a - b)
```

```
# Multiplication
```

```
print("Multiplication:", a * b)
```

```
# Division
```

```
print("Division:", a / b)
```

```
# Floor Division
```

```
print("Floor Division:", a // b)
```

```
# Modulus
```

```
print("Modulus:", a % b)
```

```
# Exponentiation
```

```
print("Exponentiation:", a ** b)
```

### Output

```
Addition: 19
```

```
Subtraction: 11
```

```
Multiplication: 60
```

```
Division: 3.75
```

```
Floor Division: 3
```

```
Modulus: 3
```

```
Exponentiation: 50625
```

**Note:** Refer to [Differences between / and //](#) for some interesting facts about these two Python operators.

### Comparison of Python Operators

In Python [Comparison of Relational operators](#) compares the values. It either returns **True** or **False** according to the condition.

### Example of Comparison Operators in Python

Let's see an example of Comparison Operators in Python.

```
a = 13
```

```
b = 33
```

```
print(a > b)
```

```
print(a < b)
```

```
print(a == b)
```

```
print(a != b)
```

```
print(a >= b)
```

```
print(a <= b)
```

### Output

```
False
True
False
True
False
True
```

## Logical Operators in Python

Python [Logical operators](#) perform **Logical AND**, **Logical OR** and **Logical NOT** operations. It is used to combine conditional statements.

The precedence of Logical Operators in Python is as follows:

1. Logical not
2. logical and
3. logical or

### Example of Logical Operators in Python:

```
a = True
b = False
print(a and b)
print(a or b)
print(not a)
```

### Output

```
False
True
False
```

## Bitwise Operators in Python

Python [Bitwise operators](#) act on bits and perform bit-by-bit operations. These are used to operate on binary numbers.

Bitwise Operators in Python are as follows:

1. Bitwise NOT
2. Bitwise Shift
3. Bitwise AND
4. Bitwise XOR
5. Bitwise OR

### Example of Bitwise Operators in Python:

```
a = 10  
b = 4
```

```
print(a & b)  
print(a | b)  
print(~a)  
print(a ^ b)  
print(a >> 2)  
print(a << 2)
```

#### Output

```
0  
14  
-11  
14  
2  
40
```

### Assignment Operators in Python

Python [Assignment operators](#) are used to assign values to the variables. This operator is used to assign the value of the right side of the expression to the left side operand.

### Example of Assignment Operators in Python:

```
a = 10  
b = a  
print(b)  
b += a  
print(b)  
b -= a  
print(b)  
b *= a  
print(b)  
b <<= a  
print(b)
```

#### Output

```
10  
20  
10  
100
```

## Identity Operators in Python

In Python, **is** and **is not** are the [identity operators](#) both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

**is**                *True*                *if*                *the*                *operands*                *are*                *identical*  
**is not** *True if the operands are not identical*

### Example of Identity Operators in Python:

```
a = 10
b = 20
c = a
```

```
print(a is not b)
print(a is c)
```

### Output

```
True
True
```

## Membership Operators in Python

In Python, **in** and **not in** are the [membership operators](#) that are used to test whether a value or variable is in a sequence.

**in**                *True*                *if*                *value*                *is*                *found*                *in*                *the*                *sequence*  
**not in** *True if value is not found in the sequence*

### Examples of Membership Operators in Python:

```
x = 24
y = 20
list = [10, 20, 30, 40, 50]
```

```
if (x not in list):
    print("x is NOT present in given list")
else:
    print("x is present in given list")
```

```
if (y in list):
    print("y is present in given list")
else:
    print("y is NOT present in given list")
```

### Output

```
x is NOT present in given list
y is present in given list
```

## Ternary Operator in Python

in Python, **Ternary operators** also known as conditional expressions are operators that evaluate something based on a condition being true or false. It was added to Python in version 2.5.

It simply allows testing a condition in a **single line** replacing the multiline if-else making the code compact.

**Syntax :** *[on\_true] if [expression] else [on\_false]*

### Examples of Ternary Operator in Python:

```
a, b = 10, 20
min = a if a < b else b

print(min)
```

### Output

```
10
```

## Precedence and Associativity of Operators in Python

In Python, Operator precedence and associativity determine the priorities of the operator.

### Operator Precedence in Python

This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

#### Example:

```
expr = 10 + 20 * 30
print(expr)
name = "Alex"
age = 0

if name == "Alex" or name == "John" and age >= 2:
    print("Hello! Welcome.")
else:
    print("Good Bye!!")
```

### Output

```
610
Hello! Welcome.
```

## Operator Associativity in Python

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left to Right or from Right to Left.

### Example:

```
print(100 / 10 * 10)
print(5 - 2 + 3)
print(5 - (2 + 3))
print(2 ** 3 ** 2)
```

### Output

```
100.0
6
0
512
```

### Python Operator Exercise Questions

Below are two Exercise Questions on Python Operators. We have covered arithmetic operators and comparison operators in these exercise questions. For more exercises on Python Operators visit the page mentioned below.

#### Q1. Code to implement basic arithmetic operations on integers

```
num1 = 5
num2 = 2

sum = num1 + num2
difference = num1 - num2
product = num1 * num2
quotient = num1 / num2
remainder = num1 % num2

print("Sum:", sum)
print("Difference:", difference)
print("Product:", product)
print("Quotient:", quotient)
print("Remainder:", remainder)
```

### Output

```
Sum: 7
Difference: 3
Product: 10
Quotient: 2.5
Remainder: 1
```

## Q2. Code to implement Comparison operations on integers

```
num1 = 30
num2 = 35

if num1 > num2:
    print("The first number is greater.")
elif num1 < num2:
    print("The second number is greater.")
else:
    print("The numbers are equal.")
```

### Output

```
The second number is greater.
```

## Python Data Types

Python Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, Python data types are classes and variables are instances (objects) of these classes. The following are the standard or built-in data types in Python:

- **Numeric** - [int](#), [float](#), [complex](#)
- **Sequence Type** - [string](#), [list](#), [tuple](#)
- **Mapping Type** - [dict](#)
- **Boolean** - [bool](#)
- **Set Type** - [set](#), [frozenset](#)
- **Binary Types** - [bytes](#), [bytearray](#), [memoryview](#)

This code assigns variable 'x' different values of few Python data types - **int**, **float**, **list**, **tuple** and **string**. Each assignment replaces the previous value, making 'x' take on the data type and value of the most recent assignment.

```
# int, float, string, list and set
x = 50
x = 60.5
x = "Hello World"
x = ["tech "]
```

## 1. Numeric Data Types in Python

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number. These values are defined as [Python int](#), [Python float](#) and [Python complex](#) classes in [Python](#).

- **Integers** - This value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.

- **Float** - This value is represented by the float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- **Complex Numbers** - A complex number is represented by a complex class. It is specified as *(real part) + (imaginary part)j*. **For example** - 2+3j

```
a = 5
```

```
print(type(a))
```

```
b = 5.0
```

```
print(type(b))
```

```
c = 2 + 4j
```

```
print(type(c))
```

### Output

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

## 2. Sequence Data Types in Python

The sequence Data Type in Python is the ordered collection of similar or different Python data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence data types of Python:

- Python String
- Python List
- Python Tuple

### String Data Type

Python Strings are arrays of bytes representing Unicode characters. In Python, there is no character data type Python, a character is a string of length one. It is represented by str class.

Strings in Python can be created using single quotes, double quotes or even triple quotes. We can access individual characters of a String using index.

```
s = 'Welcome to the tech World'
```

```
print(s)
```

```
# check data type
```

```
print(type(s))
```

```
# access string with index
```

```
print(s[1])
```

```
print(s[2])
```

```
print(s[-1])
```

## Output

```
Welcome to the tech World
<class 'str'>
e
l
d
```

## List Data Type

[Lists](#) are just like arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.

### Creating a List in Python

Lists in Python can be created by just placing the sequence inside the square brackets[].

**# Empty list**

```
a = []
```

**# list with int values**

```
a = [1, 2, 3]
print(a)
```

**# list with mixed int and string**

```
b = ["Tech", "For", "Tech", 4, 5]
print(b)
```

## Output

```
[1, 2, 3]
['Tech', 'For', 'Tech', 4, 5]
```

## Access List Items

In order to access the list items refer to the index number. In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

```
a = ["Tech", "For", "Tech"]
print("Accessing element from the list")
print(a[0])
print(a[2])

print("Accessing element using negative indexing")
print(a[-1])
print(a[-3])
```

## Output

```
Accessing element from the list
Tech
Tech
Accessing element using negative indexing
Tech
Tech
```

## Tuple Data Type

Just like a list, a [tuple](#) is also an ordered collection of Python objects. The only difference between a tuple and a list is that tuples are immutable. Tuples cannot be modified after it is created.

### Creating a Tuple in Python

In Python Data Types, [tuples](#) are created by placing a sequence of values separated by a 'comma' with or without the use of parentheses for grouping the data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, lists, etc.).

**Note:** Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing '**comma**' to make it a tuple.

# initiate empty tuple

```
tup1 = ()
```

```
tup2 = ('Tech', 'For')
```

```
print("\nTuple with the use of String: ", tup2)
```

## Output

```
Tuple with the use of String:  ('Tech', 'For')
```

**Note** - The creation of a Python tuple without the use of parentheses is known as Tuple Packing.

### Access Tuple Items

In order to access the tuple items refer to the index number. Use the index operator [ ] to access an item in a tuple.

```
tup1 = tuple([1, 2, 3, 4, 5])
```

# access tuple items

```
print(tup1[0])
```

```
print(tup1[-1])
```

```
print(tup1[-3])
```

## Output

```
1
5
```

### 3. Boolean Data Type in Python

Python Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). However non-Boolean objects can be evaluated in a Boolean context as well and determined to be true or false. It is denoted by the class bool.

**Example:** The first two lines will print the type of the boolean values True and False, which is `<class 'bool'>`. The third line will cause an error, because true is not a valid keyword in Python. Python is case-sensitive, which means it distinguishes between uppercase and lowercase letters.

```
print(type(True))
print(type(False))
print(type(true))
```

**Output:**

```
<class 'bool'>
<class 'bool'>
Traceback (most recent call last):
  File "/home/7e8862763fb66153d70824099d4f5fb7.py", line 8, in
    print(type(true))
NameError: name 'true' is not defined
```

### 4. Set Data Type in Python

In Python Data Types, [Set](#) is an unordered collection of data types that is iterable, mutable, and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

#### Create a Set in Python

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a **'comma'**. The type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

**Example:** The code is an example of how to create sets using different types of values, such as **strings**, **lists**, and mixed values

```
# initializing empty set
s1 = set()

s1 = set("Tech")
print("Set with the use of String: ", s1)

s2 = set(["Tech", "For", "Tech"])
print("Set with the use of List: ", s2)
```

**Output**

```
Set with the use of String: {'s', 'o', 'F', 'G', 'e', 'k', 'r'}
Set with the use of List: {'Tech', 'For'}
```

## Access Set Items

Set items cannot be accessed by referring to an index, since sets are unordered the items have no index. But we can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in the keyword.

```
set1 = set(["Tech", "For", "Tech"])
print(set1)
```

```
# loop through set
for i in set1:
    print(i, end=" ")
```

```
# check if item exist in set
print("Tech" in set1)
```

## Output

```
{'Tech', 'For'}
Tech For True
```

## 5. Dictionary Data Type

A dictionary in Python is a collection of data values, used to store data values like a map, unlike other Python Data Types that hold only a single value as an element, a Dictionary holds a key: value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon :, whereas each key is separated by a 'comma'.

### Create a Dictionary in Python

Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable. The dictionary can also be created by the built-in function **dict()**.

**Note** - Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.

```
# initialize empty dictionary
d = {}
```

```
d = {1: 'Tech', 2: 'For', 3: 'Tech'}
print(d)
```

```
# creating dictionary using dict() constructor
d1 = dict({1: 'Tech', 2: 'For', 3: 'Tech'})
print(d1)
```

## Output

```
{1: 'Tech', 2: 'For', 3: 'Tech'}  
{1: 'Tech', 2: 'For', 3: 'Tech'}
```

## Accessing Key-value in Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets. Using **get() method** we can access the dictionary elements.

```
d = {1: 'Tech', 'name': 'For', 3: 'Tech'}
```

```
# Accessing an element using key  
print(d['name'])
```

```
# Accessing a element using get  
print(d.get(3))
```

## Output

```
For  
Tech
```

## Python Data Type Exercise Questions

Below are two exercise questions on Python Data Types. We have covered list operation and tuple operation in these exercise questions. For more exercises on Python data types visit the page mentioned below.

### Q1. Code to implement basic list operations

```
fruits = ["apple", "banana", "orange"]  
print(fruits)  
fruits.append("grape")  
print(fruits)  
fruits.remove("orange")  
print(fruits)
```

## Output

```
['apple', 'banana', 'orange']  
['apple', 'banana', 'orange', 'grape']  
['apple', 'banana', 'grape']
```

### Q2. Code to implement basic tuple operation

```
coordinates = (3, 5)  
print(coordinates)  
print("X-coordinate:", coordinates[0])  
print("Y-coordinate:", coordinates[1])
```

## Output

```
(3, 5)
```

```
X-coordinate: 3
```

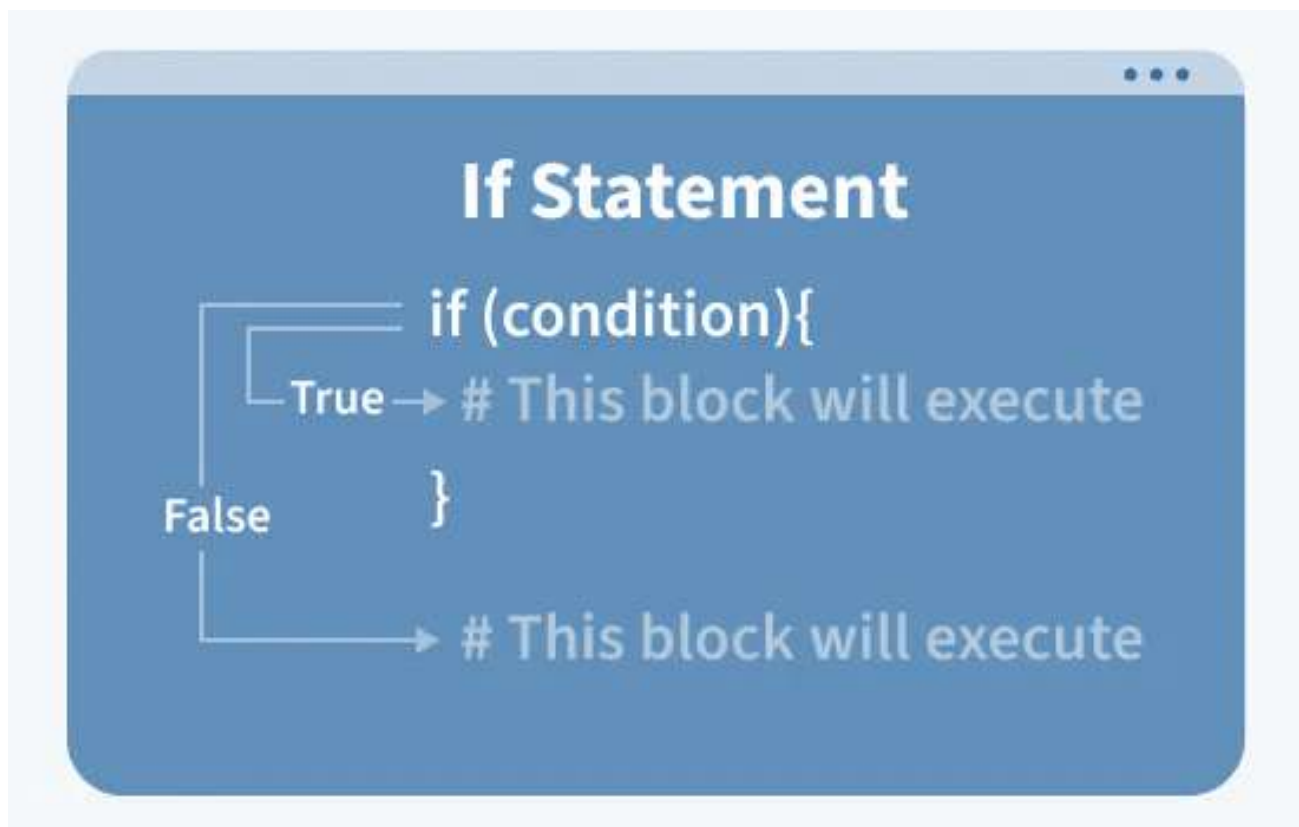
```
Y-coordinate: 5
```

## Conditional Statements in Python

Conditional statements in [Python](#) are used to execute certain blocks of code based on specific conditions. These statements help control the flow of a program, making it behave differently in different situations.

### If Conditional Statement in Python

If statement is the simplest form of a conditional statement. It executes a block of code if the given condition is true.



If Statement

#### Example:

```
age = 20
```

```
if age >= 18:  
    print("Eligible to vote.")
```

#### Output

```
Eligible to vote.
```

#### Short Hand if

Short-hand if statement allows us to write a single-line if statement.

### Example:

```
age = 19
if age > 18: print("Eligible to Vote.")
```

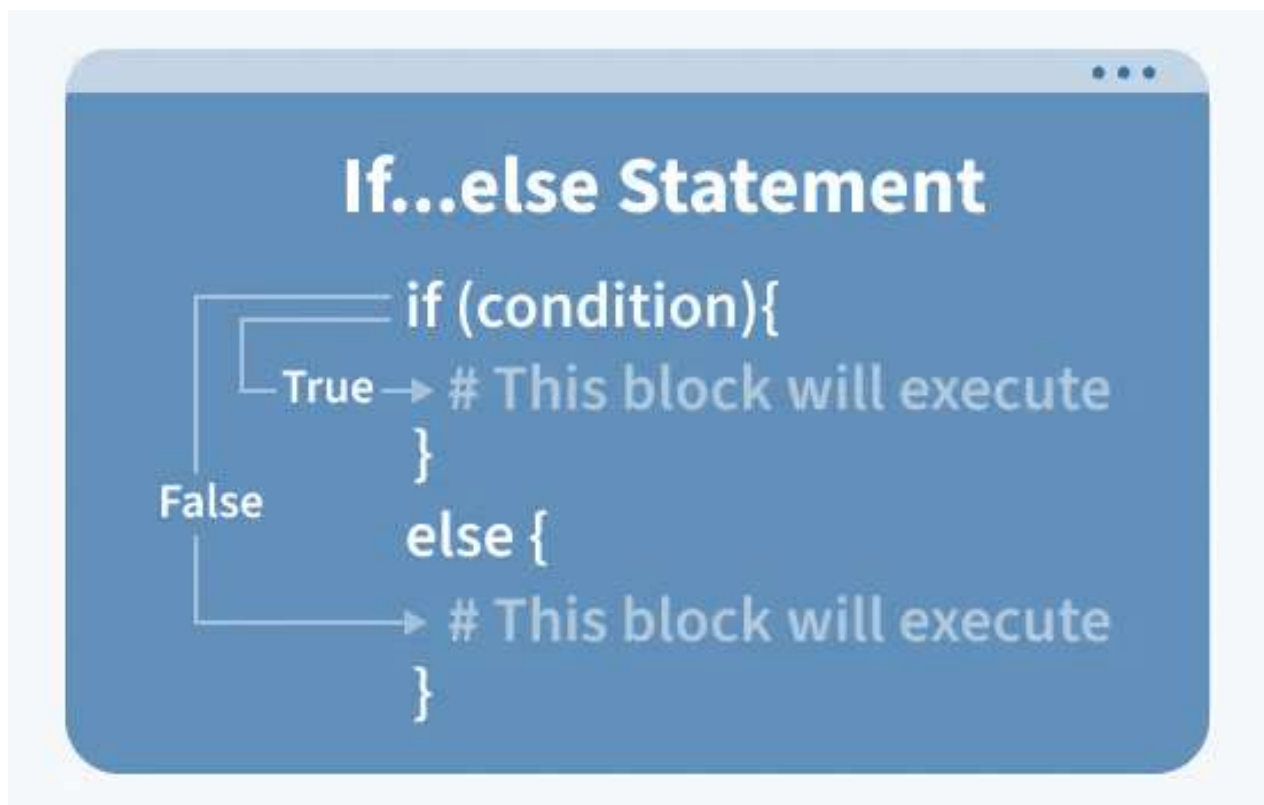
### Output

```
Eligible to Vote.
```

This is a compact way to write an if statement. It executes the print statement if the condition is true.

### If else Conditional Statements in Python

Else allows us to specify a block of code that will execute if the condition(s) associated with an if or elif statement evaluates to False. Else block provides a way to handle all other cases that don't meet the specified conditions.



### If Else Statement

#### Example:

```
age = 10

if age <= 12:
    print("Travel for free.")
else:
```

```
print("Pay for ticket.")
```

### Output

```
Travel for free.
```

### Short Hand if-else

The short-hand if-else statement allows us to write a single-line if-else statement.

### Example:

```
marks = 45  
res = "Pass" if marks >= 40 else "Fail"
```

```
print(f"Result: {res}")
```

### Output

```
Result: Pass
```

**Note:** This method is also known as ternary operator. Ternary Operator essentially a shorthand for the if-else statement that allows us to write more compact and readable code, especially for simple conditions.

### elif Statement

[elif statement in Python](#) stands for "else if." It allows us to check multiple conditions , providing a way to execute different blocks of code based on which condition is true. Using elif statements makes our code more readable and efficient by eliminating the need for multiple nested if statements.

### Example:

```
age = 25  
  
if age <= 12:  
    print("Child.")  
elif age <= 19:  
    print("Teenager.")  
elif age <= 35:  
    print("Young adult.")  
else:  
    print("Adult.")
```

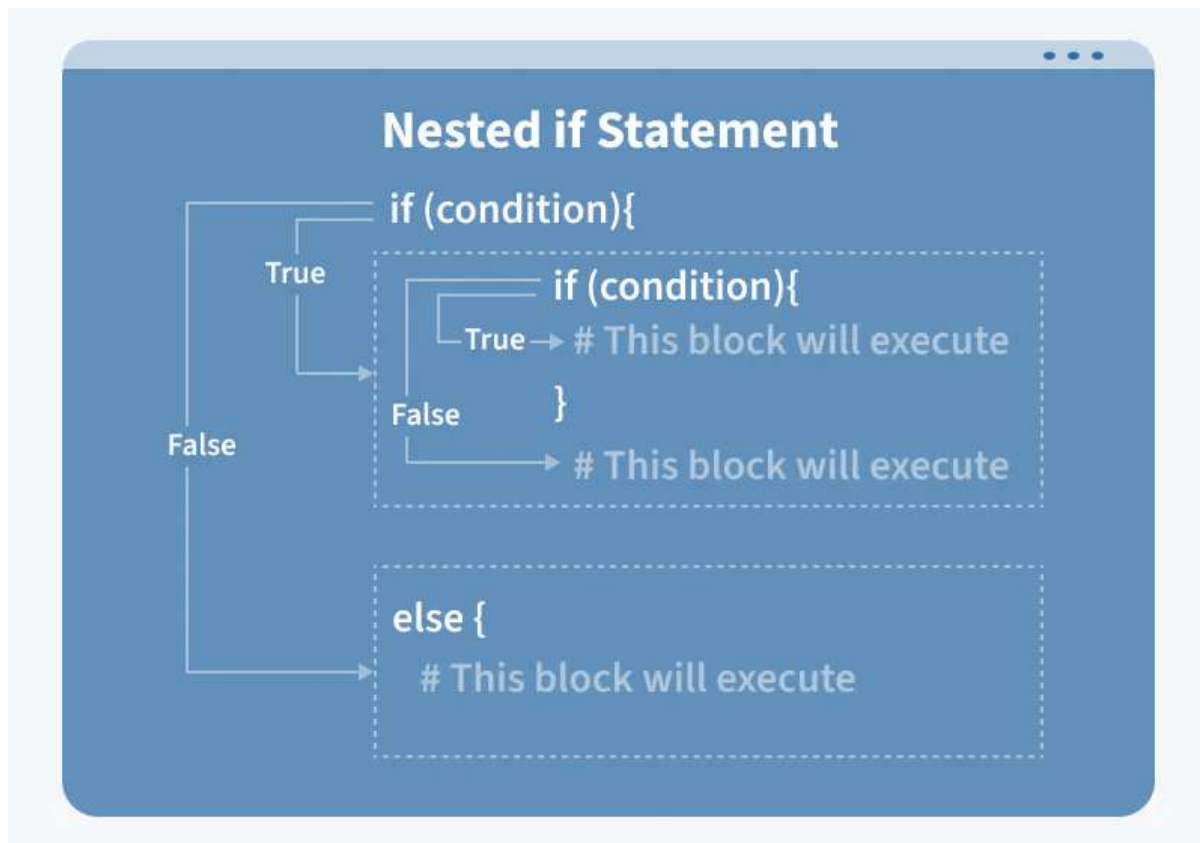
### Output

```
Young adult.
```

The code checks the value of age using if-elif-else. Since age is 25, it skips the first two conditions (age <= 12 and age <= 19), and the third condition (age <= 35) is True, so it prints "Young adult."

## Nested if..else Conditional Statements in Python

Nested if..else means an if-else statement inside another if statement. We can use nested if statements to check conditions within conditions.



Nested If Else

age = 70

is\_member = True

```
if age >= 60:
    if is_member:
        print("30% senior discount!")
    else:
        print("20% senior discount.")
else:
    print("Not eligible for a senior discount.")
```

**Output**

```
30% senior discount!
```

## Ternary Conditional Statement in Python

A ternary conditional statement is a compact way to write an if-else condition in a single line. It's sometimes called a "conditional expression."

### Example:

```
# Assign a value based on a condition
age = 20
s = "Adult" if age >= 18 else "Minor"

print(s)
```

### Output

```
Adult
```

Here:

- If age >= 18 is True, status is assigned "Adult".
- Otherwise, status is assigned "Minor".

### Match-Case Statement in Python

match-case statement is Python's version of a switch-case found in other languages. It allows us to match a variable's value against a set of patterns.

### Example:

```
number = 2

match number:
    case 1:
        print("One")
    case 2 | 3:
        print("Two or Three")
    case _:
        print("Other number")
```

### Output:

```
Two or Three
```

### Loops in Python - For, While and Nested Loops

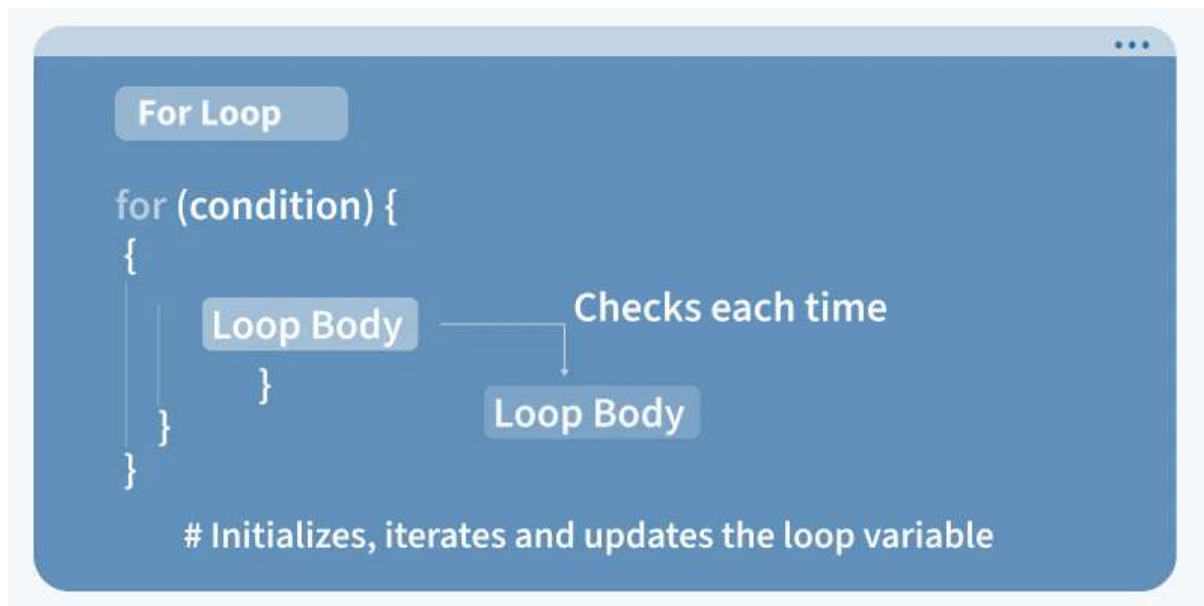
Loops in Python are used to repeat actions efficiently. The main types are For loops (counting through items) and While loops (based on conditions). In this article, we will look at Python loops and understand their working with the help of examples.

### For Loop in Python

**For loops** is used to iterate over a sequence such as a list, tuple, string or range. It allow to execute a block of code repeatedly, once for each item in the sequence.

## Syntax:

```
For iterator_var in sequence:  
    statements(s)
```



## Example:

```
n = 4  
for i in  
    range(0,  
        n):  
    print(i)
```

## Output

```
0  
1  
2  
3
```

**Explanation:** This code prints the numbers from 0 to 3 (inclusive) using a for loop that iterates over a range from 0 to n-1 (where n = 4).

## Example:

Iterating Over List, Tuple, String and Dictionary Using for Loops in Python

```
li = ["tech", "for", "tech"]  
for i in li:  
    print(i)
```

```
tup = ("tech", "for", "tech")  
for i in tup:  
    print(i)
```

```
s = "Tech"  
for i in s:  
    print(i)
```

```
d = dict({'x':123, 'y':354})  
for i in d:  
    print("%s %d" % (i, d[i]))
```

```
set1 = {1, 2, 3, 4, 5, 6}  
for i in set1:  
    print(i),
```

## Output

```
tech
for
tech
tech
for
tech
G
e
e
k
s
x 123
y 354
1
2
3
4
5
6
```

## Iterating by the Index of Sequences

We can also use the index of elements in the sequence to iterate. The key idea is to first calculate the length of the list and then iterate over the sequence within the range of this length.

```
li = ["tech", "for", "tech"]
for index in range(len(li)):
    print(li[index])
```

## Output

```
tech
for
tech
```

**Explanation:** This code iterates through each element of the list using its index and prints each element one by one. The **range(len(list))** generates indices from 0 to the length of the list minus 1.

## Using else Statement with for Loop in Python

We can also combine else statement with for loop like in while loop. But as there is no condition in for loop based on which the execution will terminate so the else block will be executed immediately after for block finishes execution.

```
li = ["tech", "for", "tech"]
for index in range(len(li)):
    print(li[index])
else:
    print("Inside Else Block")
```

### Output

```
tech
for
tech
Inside Else Block
```

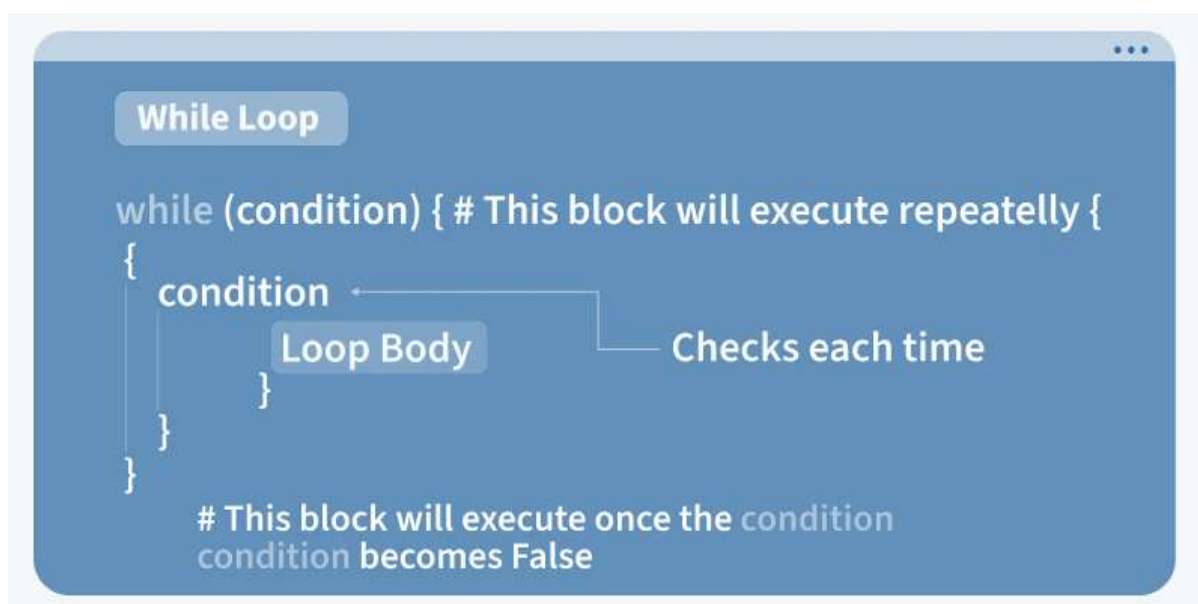
**Explanation:** The code iterates through the list and prints each element. After the loop ends it prints "Inside Else Block" as the else block executes when the loop completes without a break.

## While Loop in Python

In Python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed.

### Syntax:

```
while expression:
    statement(s)
while loop explanation
```



All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

### Example:

This example demonstrates a basic while loop in Python. The loop runs as long as the condition `cnt < 3` is true. It increments the counter by 1 on each iteration and prints "Hello Geek" three times.

```
cnt = 0
while (cnt < 3):
    cnt = cnt + 1
    print("Hello TECH")
```

### Output

```
Hello TECH
Hello TECH
Hello TECH
```

### Using else statement with While Loop in Python

Else clause is only executed when our while condition becomes false. If we break out of the loop or if an exception is raised then it won't be executed.

### Syntax of While Loop with else statement:

<i>while</i>			<i>condition:</i>
<i>#</i>	<i>execute</i>	<i>these</i>	<i>statements</i>
<i>else:</i>			
<i># execute these statements</i>			

### Example:

The code prints "Hello Geek" three times using a 'while' loop and then after the loop it prints "In Else Block" because there is an "else" block associated with the 'while' loop.

```
cnt = 0
while (cnt < 3):
    cnt = cnt + 1
    print("Hello TECH")
else:
    print("In Else Block")
```

### Output

```
Hello TECH
Hello TECH
Hello TECH
In Else Block
```

## Infinite While Loop in Python

If we want a block of code to execute infinite number of times then we can use the while loop in Python to do so.

The code given below uses a 'while' loop with the condition "**True**", which means that the loop will run infinitely until we break out of it using "**break**" keyword or some other logic.

```
while (True):  
    print("Hello Geek")
```

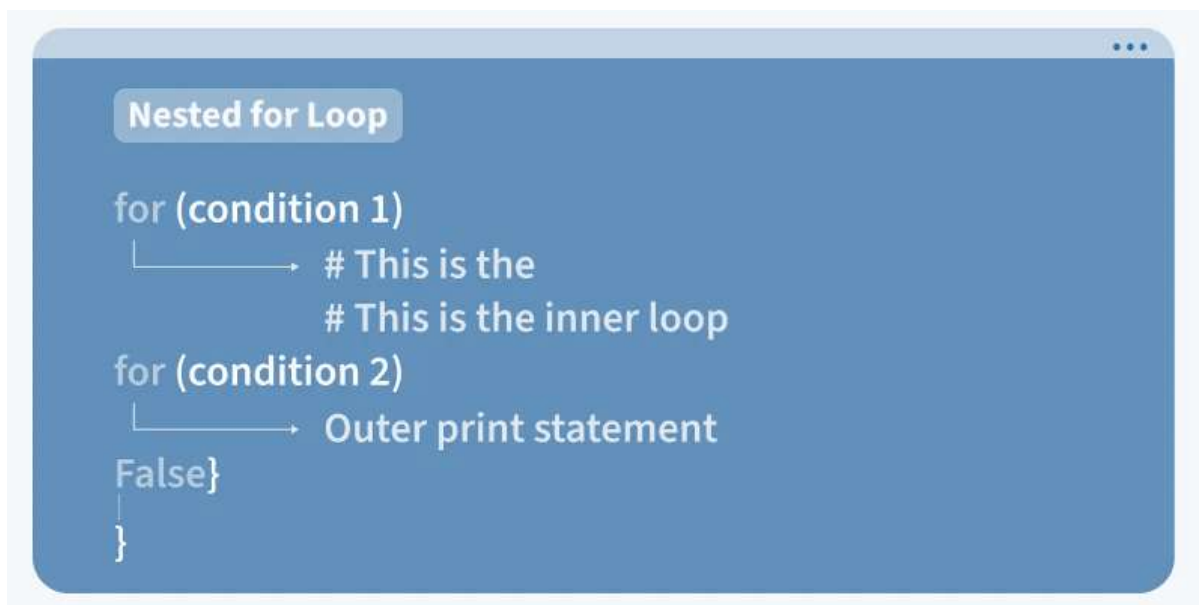
**Note:** It is suggested not to use this type of loop as it is a never-ending infinite loop where the condition is always true and we have to forcefully terminate the compiler.

## Nested Loops in Python

Python programming language allows to use one loop inside another loop which is called [nested loop](#). Following section shows few examples to illustrate the concept.

### Syntax:

<code>for</code>	<code>iterator_var</code>	<code>in</code>	<code>sequence:</code>
<code>for</code>	<code>iterator_var</code>	<code>in</code>	<code>sequence:</code>
<code>statements(s)</code>			
<code>statements(s)</code>			
Nested for loop			



The syntax for a nested while loop statement in the Python programming language is as follows:

<code>while</code>	<code>expression:</code>
<code>while</code>	<code>expression:</code>
<code>statement(s)</code>	
<code>statement(s)</code>	
Nested while loop	

### Nested While Loop

```
while (condition2) {  
    {  
        while condition2)  
        {    Statements  
        }  
    }  
}
```

A final note on loop nesting is that we can put any type of loop inside of any other type of loops in Python. For example, a for loop can be inside a while loop or vice versa.

```
from __future__ import print_function  
for i in range(1, 5):  
    for j in range(i):  
        print(i, end=' ')  
    print()
```

### Output

```
1  
2 2  
3 3 3  
4 4 4 4
```

**Explanation:** In the above code we use nested loops to print the value of i multiple times in each row, where the number of times it prints i increases with each iteration of the outer loop. The print() function prints the value of i and moves to the next line after each row.

### Loop Control Statements

[Loop control statements](#) change execution from their normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

#### Continue Statement

The [continue statement](#) in Python returns the control to the beginning of the loop.

#### Example:

```
for letter in 'tech':  
    if letter == 'e' or letter == 's':  
        continue
```

```
print('Current Letter :', letter)
```

## Break Statement

The [break statement](#) in Python brings control out of the loop.

### Example:

```
for letter in 'tech':  
    if letter == 'e' or letter == 'h':  
        break
```

```
print('Current Letter :', letter)
```

### Output

```
Current Letter : e
```

**Explanation:** break statement is used to exit the loop prematurely when a specified condition is met. In this example, the loop breaks when the letter is either 'e' or 's', stopping further iteration.

## Pass Statement

We use [pass statement](#) in Python to write empty loops. Pass is also used for empty control statements, functions and classes.

### Example:

```
for letter in 'tech':  
    pass  
print('Last Letter :', letter)
```

### Output

```
Last Letter : h
```

**Explanation:** In this example, the loop iterates over each letter in 'tech' but doesn't perform any operation, and after the loop finishes, the last letter ('s') is printed.

## How for loop works internally in Python?

Before proceeding to this section, we should have a prior understanding of Python Iterators.

Firstly, lets see how a simple for loops in Python looks like.

### Example 1:

This Python code iterates through a list called fruits, containing "apple", "orange" and "kiwi." It prints each fruit name on a separate line, displaying them in the order they appear in the list.

```
fruits = ["apple", "orange", "kiwi"]
```

```
for fruit in fruits:
```

```
    print(fruit)
```

This code iterates over each item in the fruits list and prints the item (fruit) on each iteration and the output will display each fruit on a new line.

### Example 2:

This Python code manually iterates through a list of fruits using an iterator. It prints each fruit's name one by one and stops when there are no more items in the list.

```
fruits = ["apple", "orange", "kiwi"]
```

```
iter_obj = iter(fruits)
```

```
while True:
```

```
    try:
```

```
        fruit = next(iter_obj)
```

```
        print(fruit)
```

```
    except StopIteration:
```

```
        break
```

### Output

```
apple
```

```
orange
```

```
kiwi
```

### Python Functions

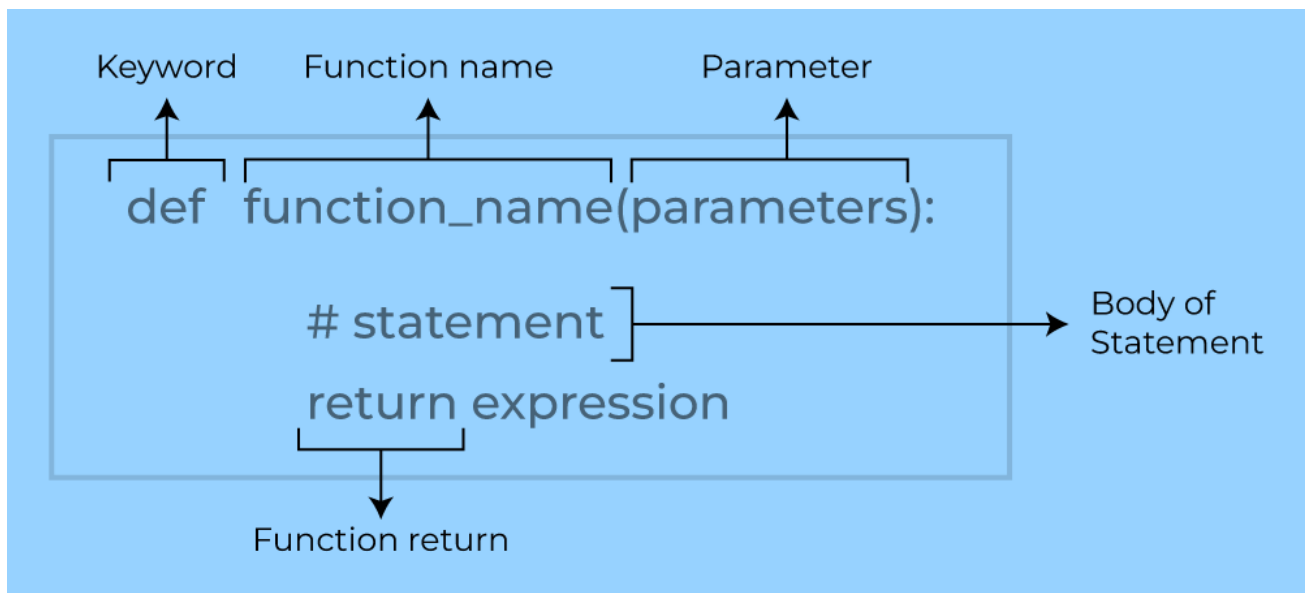
**Python Functions** is a block of statements that does a specific task. The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

### Benefits of Using Functions

- Code Reuse
- Reduced code length
- Increased readability of code

### Python Function Declaration

The syntax to declare a function is:



## Syntax of Python Function Declaration

### Types of Functions in Python

Below are the different types of functions in Python:

- **Built-in library function:** These are Standard functions in Python that are available to use.
- **User-defined function:** We can create our own functions based on our requirements.

### Creating a Function in Python

We can define a function in Python, using the **def keyword**. We can add any type of functionalities and properties to it as we require.

#### What is def ?

The **def keyword** stands for define. It is used to create a **user-defined function**. It marks the beginning of a function block and allows you to group a set of statements so they can be reused when the function is called.

#### Syntax:

```
def function_name(parameters):
```

```
    # function body
```

#### Explanation:

- **def:** Starts the function definition.
- **function\_name:** Name of the function.
- **parameters:** Inputs passed to the function (inside `()`), optional.
- **::** Indicates the start of the function body.
- **Indented code:** The function body that runs when called.

**Example:** Let's understand this with a simple example. Here, we define a function using `def` that prints a welcome message when called.

```
def fun():  
    print("Welcome to GFG")
```

For more information, refer to this article: [Python def Keyword](#)

## Calling a Function in Python

After creating a function in Python we can call it by using the name of the functions Python followed by parenthesis containing parameters of that particular function. Below is the example for calling def function Python.

```
def fun():  
    print("Welcome to GFG")
```

```
# Driver code to call a function  
fun()
```

## Output

```
Welcome to GFG
```

## Python Function Arguments

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

### Syntax for functions with arguments:

```
def          function_name(parameter:          data_type)          ->          return_type:  
    """Docstring"""  
#              body              of              the              function  
return expression
```

**data\_type** and **return\_type** are optional in function declaration, meaning the same function can also be written as:

```
def          function_name(parameter)          :  
    """Docstring"""  
#              body              of              the              function  
return expression
```

Let's understand this with an example, we will create a simple function in Python to check whether the number passed as an argument to the function is even or odd.

```
def evenOdd(x: int) ->str:  
    if (x % 2 == 0):  
        return "Even"  
    else:  
        return "Odd"
```

```
print(evenOdd(16))  
print(evenOdd(7))
```

## Output

Even

Odd

The above function can also be declared without **type\_hints**, like this:

```
def evenOdd(x):  
    if (x % 2 == 0):  
        return "Even"  
    else:  
        return "Odd"  
  
print(evenOdd(16))  
print(evenOdd(7))
```

## Output

Even

Odd

## Types of Python Function Arguments

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following function argument types in Python:

- **Default argument**
- **Keyword arguments (named arguments)**
- **Positional arguments**
- **Arbitrary arguments** (variable-length arguments \*args and \*\*kwargs)

Let's discuss each type in detail.

### Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments to write functions in Python.

```
def myFun(x, y=50):  
    print("x: ", x)  
    print("y: ", y)
```

myFun(10)

## Output

x: 10

y: 50

Like C++ default arguments, any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

## Keyword Arguments

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

```
def student(fname, lname):  
    print(fname, lname)  
  
student(fname='Tech', lname='Practice')  
student(lname='Practice', fname='Tech')
```

## Output

```
Tech Practice  
Tech Practice
```

## Positional Arguments

We used the [Position argument](#) during the function call so that the first argument (or value) is assigned to name and the second argument (or value) is assigned to age. By changing the position, or if you forget the order of the positions, the values can be used in the wrong places, as shown in the Case-2 example below, where 27 is assigned to the name and Suraj is assigned to the age.

```
def nameAge(name, age):  
    print("Hi, I am", name)  
    print("My age is ", age)
```

```
print("Case-1:")  
nameAge("Suraj", 27)
```

```
print("\nCase-2:")  
nameAge(27, "Suraj")
```

## Output

```
Case-1:  
Hi, I am Suraj  
My age is 27  
  
Case-2:  
Hi, I am 27  
My age is Suraj
```

## Arbitrary Keyword Arguments

In Python Arbitrary Keyword Arguments, `*args`, and `**kwargs` can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- `*args` in Python (Non-Keyword Arguments)
- `**kwargs` in Python (Keyword Arguments)

**Example 1:** Variable length non-keywords argument

```
def myFun(*argv):  
    for arg in argv:  
        print(arg)
```

```
myFun('Hello', 'Welcome', 'to', 'Tech')
```

### Output

```
Hello  
Welcome  
to  
Tech
```

**Example 2:** Variable length keyword arguments

```
def myFun(**kwargs):  
    for key, value in kwargs.items():  
        print("%s == %s" % (key, value))
```

```
myFun(first='Tech', mid='for', last='Tech')
```

### Output

```
first == Tech  
mid == for  
last == Tech
```

## Docstring

The first string after the function is called the Document string or [Docstring](#) in short. This is used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

The below syntax can be used to print out the docstring of a function.

**Syntax:** `print(function_name.__doc__)`

**Example:** Adding Docstring to the function

```
def evenOdd(x):
```

```
"""Function to check if the number is even or odd"""
```

```
if (x % 2 == 0):  
    print("even")  
else:  
    print("odd")
```

```
print(evenOdd.__doc__)
```

### Output

```
Function to check if the number is even or odd
```

### Python Function within Functions

A function that is defined inside another function is known as the **inner function** or **nested function**. Nested functions can access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function.

```
def f1():  
    s = 'I love Tech'
```

```
    def f2():  
        print(s)
```

```
    f2()
```

```
f1()
```

### Output

```
I love Tech
```

### Anonymous Functions in Python

In Python, an [anonymous function](#) means that a function is without a name. As we already know the `def` keyword is used to define the normal functions and the `lambda` keyword is used to create anonymous functions.

```
def cube(x): return x*x*x # without lambda
```

```
cube_l = lambda x : x*x*x # with lambda
```

```
print(cube(7))  
print(cube_l(7))
```

### Output

```
343
```

```
343
```

## Return Statement in Python Function

The **return** statement in Python is used to exit a function and send a value back to the caller. It can return any **data type**, and if multiple values are separated by commas, they are automatically packed into a **tuple**. If no value is specified, the function returns **None** by default.

### Syntax:

```
return [expression]
```

### Explanation:

- **return:** Ends the function and optionally sends a value to the caller.
- **[expression]:** Optional value to return, defaults to None if omitted.

### Example: Python Function Return Statement

```
def square_value(num):  
    """This function returns the square  
    value of the entered number"""  
    return num**2
```

```
print(square_value(2))  
print(square_value(-4))
```

### Output

```
4  
  
16
```

For more information, refer to this article: [Python return statement](#)

## Pass by Reference and Pass by Value

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function Python, a new reference to the object is created. Parameter passing in Python is the same as reference passing in Java.

# Here x is a new reference to same list lst

```
def myFun(x):  
    x[0] = 20
```

# Driver Code (Note that lst is modified  
# after function call.

```
lst = [10, 11, 12, 13, 14, 15]  
myFun(lst)  
print(lst)
```

### Output

```
[20, 11, 12, 13, 14, 15]
```

When we pass a reference and change the received reference to something else, the connection between the passed and received parameters is broken. For example, consider the below program as follows:

```
def myFun(x):  
    x = [20, 30, 40]
```

```
lst = [10, 11, 12, 13, 14, 15]  
myFun(lst)  
print(lst)
```

### Output

```
[10, 11, 12, 13, 14, 15]
```

Another example demonstrates that the reference link is broken if we assign a new value (inside the function).

```
def myFun(x):  
    x = 20
```

```
x = 10  
myFun(x)  
print(x)
```

### Output

```
10
```

**Exercise:** Try to guess the output of the following code.

```
def swap(x, y):  
    temp = x  
    x = y  
    y = temp
```

```
x = 2  
y = 3  
swap(x, y)  
print(x)  
print(y)
```

### Output

```
2  
3
```

## Recursive Functions in Python

**Recursion** in Python refers to when a function calls itself. There are many instances when you have to build a recursive function to solve **Mathematical and Recursive Problems**.

Using a recursive function should be done with caution, as a recursive function can become like a non-terminating loop. It is better to check your exit statement while creating a recursive function.

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
print(factorial(4))
```

### Output

```
24
```

## Python Lambda Functions

**Python Lambda Functions** are anonymous functions means that the function is without a name. As we already know the *def* keyword is used to define a normal function in Python. Similarly, the *lambda* keyword is used to define an anonymous function in [Python](#).

In the example, we defined a lambda function(**upper**) to convert a string to its upper case using [upper\(\)](#).

```
s1 = 'Tech'  
  
s2 = lambda func: func.upper()  
print(s2(s1))
```

### Output

```
TECH
```

This code defines a lambda function named **s2** that takes a string as its argument and converts it to uppercase using the **upper()** method. It then applies this lambda function to the string 'Tech' and prints the result.

Let's explore Lambda Function in detail:

## Python Lambda Function Syntax

**Syntax:** *lambda arguments : expression*

- **lambda:** The keyword to define the function.
- **arguments:** A comma-separated list of input parameters (like in a regular function).
- **expression:** A single expression that is evaluated and returned.

Let's see some of the practical uses of the Python lambda function.

## lambda with Condition Checking

A lambda function can include conditions using if statements.

### Example:

```
# Example: Check if a number is positive, negative, or zero  
n = lambda x: "Positive" if x > 0 else "Negative" if x < 0 else "Zero"  
  
print(n(5))
```

```
print(n(-3))
print(n(0))
```

### Output

```
Positive
Negative
Zero
```

### Explanation:

- The lambda function takes x as input.
- It uses nested if-else statements to return "Positive," "Negative," or "Zero."

### Difference Between lambda and def Keyword

lambda is concise but less powerful than def when handling complex logic. Let's take a look at short comparison between the two:

Feature	lambda Function	Regular Function (def)
Definition	Single expression with <code>lambda</code> .	Multiple lines of code.
Name	Anonymous (or named if assigned).	Must have a name.
Statements	Single expression only.	Can include multiple statements.
Documentation	Cannot have a docstring.	Can include docstrings.
Reusability	Best for short, temporary functions.	Better for reusable and complex logic.

### Example:

```
# Using lambda
sq = lambda x: x ** 2
print(sq(3))
```

```
# Using def
def sqdef(x):
    return x ** 2
print(sqdef(3))
```

### Output

```
9
9
```

As we can see in the above example, both the `sq()` function and `sqdef()` function behave the same and as intended.

## Lambda with List Comprehension

Combining lambda with [list comprehensions](#) enables us to apply transformations to data in a concise way.

### Example:

```
li = [lambda arg=x: arg * 10 for x in range(1, 5)]  
for i in li:  
    print(i())
```

### Output

```
10  
20  
30  
40
```

### Explanation:

- The lambda function squares each element.
- The list comprehension iterates through li and applies the lambda to each element.
- This is ideal for applying transformations to datasets in data preprocessing or manipulation tasks.

## Lambda with if-else

lambda functions can incorporate conditional logic directly, allowing us to handle simple decision making within the function.

### Example:

```
# Example: Check if a number is even or odd  
check = lambda x: "Even" if x % 2 == 0 else "Odd"  
  
print(check(4))  
print(check(7))
```

### Output

```
Even  
Odd
```

### Explanation:

- The lambda checks if a number is divisible by 2 ( $x \% 2 == 0$ ).
- Returns "Even" for true and "Odd" otherwise.
- This approach is useful for labeling or categorizing values based on simple conditions.

## Lambda with Multiple Statements

Lambda functions do not allow multiple statements, however, we can create two lambda functions and then call the other lambda function as a parameter to the first function.

### Example:

**# Example: Perform addition and multiplication in a single line**

```
calc = lambda x, y: (x + y, x * y)
```

```
res = calc(3, 4)
```

```
print(res)
```

### Output

```
(7, 12)
```

### Explanation:

- The lambda function performs both addition and multiplication and returns a tuple with both results.
- This is useful for scenarios where multiple calculations need to be performed and returned together.

Lambda functions can be used along with built-in functions like [filter\(\)](#), [map\(\)](#) and [reduce\(\)](#).

### Using lambda with filter()

The [filter\(\)](#) function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence "sequence", for which the function returns True.

### Example:

**# Example: Filter even numbers from a list**

```
n = [1, 2, 3, 4, 5, 6]
```

```
even = filter(lambda x: x % 2 == 0, n)
```

```
print(list(even))
```

### Output

```
[2, 4, 6]
```

### Explanation:

- The lambda function checks if a number is even ( $x \% 2 == 0$ ).
- `filter()` applies this condition to each element in `nums`.

### Using lambda with map()

The [map\(\) function](#) in Python takes in a function and a list as an argument. The function is called with a lambda function and a new list is returned which contains all the lambda-modified items returned by that function for each item.

### Example:

**# Example: Double each number in a list**

```
a = [1, 2, 3, 4]
b = map(lambda x: x * 2, a)
print(list(b))
```

### Output

```
[2, 4, 6, 8]
```

### Explanation:

- The lambda function doubles each number.
- map() iterates through a and applies the transformation.

### Using lambda with reduce()

The [reduce\(\)](#) function in Python takes in a function and a list as an argument. The function is called with a lambda function and an iterable and a new reduced result is returned. This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the [functools module](#).

### Example:

```
from functools import reduce
```

```
# Example: Find the product of all numbers in a list
```

```
a = [1, 2, 3, 4]
b = reduce(lambda x, y: x * y, a)
print(b)
```

### Output

```
24
```

## Python OOP Basics

Object Oriented Programming is a fundamental concept in Python, empowering developers to build modular, maintainable, and scalable applications. By understanding the core OOP principles (classes, objects, inheritance, encapsulation, polymorphism, and abstraction), programmers can leverage the full potential of Python OOP capabilities to design elegant and efficient solutions to complex problems.

OOPs is a way of organizing code that uses objects and classes to represent real-world entities and their behavior. In OOPs, object has attributes thing that has specific data and can perform certain actions using methods.

### OOPs Concepts in Python

- Class in Python
- Objects in Python
- Polymorphism in Python
- Encapsulation in Python
- Inheritance in Python
- Data Abstraction in Python



## Python OOPs Concepts

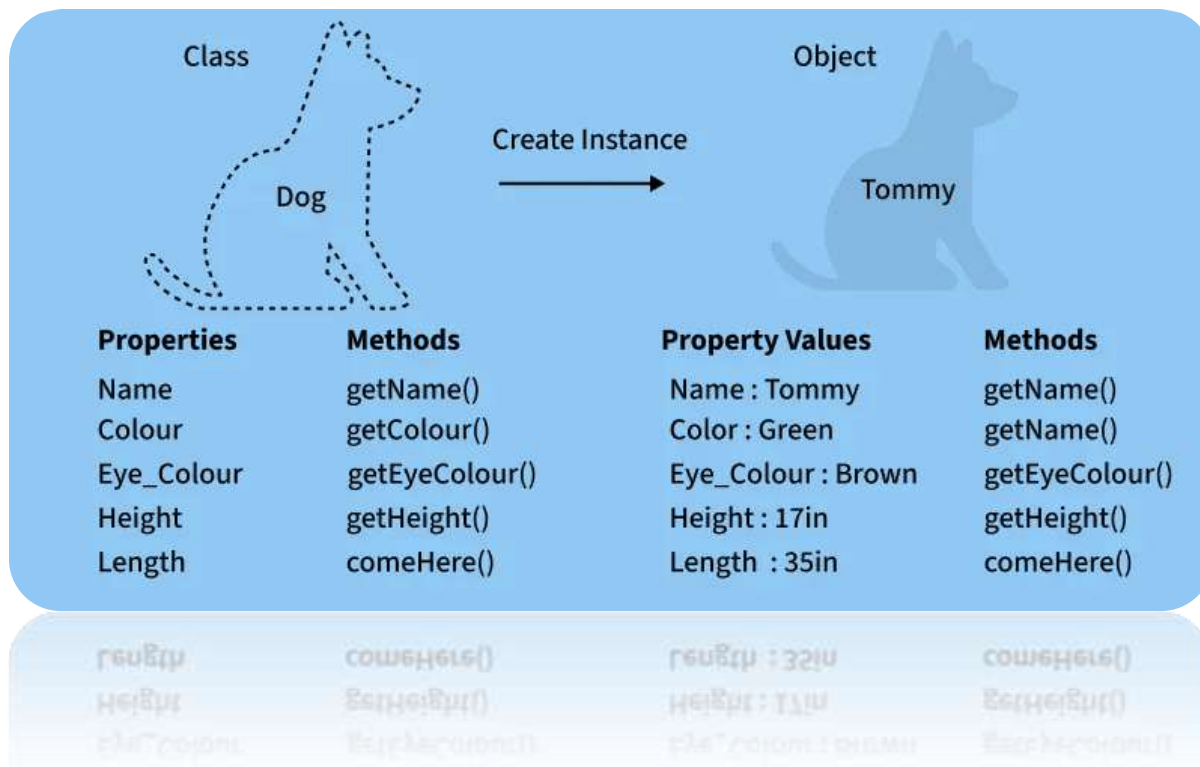
### Python Class

A class is a collection of objects. [Classes](#) are blueprints for creating objects. A class defines a set of attributes and methods that the created objects (instances) can have.

#### Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Example:  
Myclass.Myattribute

Classes and Objects (Here Dog is the Class and Bobby is Object)



## Creating a Class

Here, the class keyword indicates that we are creating a class followed by name of the class (Dog in this case).

```
class Dog:
    species = "Canine" # Class attribute

    def __init__(self, name, age):
        self.name = name # Instance attribute
        self.age = age # Instance attribute
```

### Explanation:

- **class Dog:** Defines a class named Dog.
- **species:** A class attribute shared by all instances of the class.
- **\_\_init\_\_ method:** Initializes the name and age attributes when a new object is created.

**Note:** For more information, refer to [python classes](#).

## Python Objects

An Object is an instance of a Class. It represents a specific implementation of the class and holds its own data.

An object consists of:

- **State:** It is represented by the attributes and reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object and reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

## Creating Object

Creating an object in Python involves instantiating a class to create a new instance of that class. This process is also referred to as object instantiation.

```
class Dog:
    species = "Canine" # Class attribute

    def __init__(self, name, age):
        self.name = name # Instance attribute
        self.age = age # Instance attribute

# Creating an object of the Dog class
dog1 = Dog("Buddy", 3)

print(dog1.name)
print(dog1.species)
```

### Output

```
Buddy
Canine
```

### Explanation:

- **dog1 = Dog("Buddy", 3):** Creates an object of the Dog class with name as "Buddy" and age as 3.
- **dog1.name:** Accesses the instance attribute name of the dog1 object.
- **dog1.species:** Accesses the class attribute species of the dog1 object.

*Note: For more information, refer to [python objects](#).*

### `__init__` Method

`__init__` method is the constructor in Python, automatically called when a new object is created. It initializes the attributes of the class.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

dog1 = Dog("Buddy", 3)
print(dog1.name)
```

### Output

```
Buddy
```

### Explanation:

- **`__init__`:** Special method used for initialization.
- **`self.name` and `self.age`:** Instance attributes initialized in the constructor.

## Class and Instance Variables

In Python, variables defined in a class can be either class variables or instance variables, and understanding the distinction between them is crucial for object-oriented programming.

### Class Variables

These are the variables that are shared across all instances of a class. It is defined at the class level, outside any methods. All objects of the class share the same value for a class variable unless explicitly overridden in an object.

### Instance Variables

Variables that are unique to each instance (object) of a class. These are defined within the `__init__` method or other instance methods. Each object maintains its own copy of instance variables, independent of other objects.

```
class Dog:
    # Class variable
    species = "Canine"

    def __init__(self, name, age):
        # Instance variables
        self.name = name
        self.age = age

# Create objects
dog1 = Dog("Buddy", 3)
dog2 = Dog("Charlie", 5)

# Access class and instance variables
print(dog1.species) # (Class variable)
print(dog1.name)    # (Instance variable)
print(dog2.name)    # (Instance variable)

# Modify instance variables
dog1.name = "Max"
print(dog1.name)    # (Updated instance variable)

# Modify class variable
Dog.species = "Feline"
print(dog1.species) # (Updated class variable)
print(dog2.species)
```

### Output

```
Canine
Buddy
Charlie
```

```
Max
Feline
Feline
```

### Explanation:

- **Class Variable (species):** Shared by all instances of the class. Changing Dog.species affects all objects, as it's a property of the class itself.
- **Instance Variables (name, age):** Defined in the \_\_init\_\_ method. Unique to each instance (e.g., dog1.name and dog2.name are different).
- **Accessing Variables:** Class variables can be accessed via the class name (Dog.species) or an object (dog1.species). Instance variables are accessed via the object (dog1.name).
- **Updating Variables:** Changing Dog.species affects all instances. Changing dog1.name only affects dog1 and does not impact dog2.

### Python String

A string is a sequence of characters. Python treats anything inside quotes as a string. This includes letters, numbers, and symbols. Python has no character data type so single character is a string of length 1.

```
s = "GfG"
```

```
print(s[1]) # access 2nd char
s1 = s + s[0] # update
print(s1) # print
```

### Output

```
f
GfGG
```

In this example, **s** holds the value "GfG" and is defined as a string.

### Creating a String

Strings can be created using either **single (')** or **double (")** quotes.

```
s1 = 'GfG'
s2 = "GfG"
print(s1)
print(s2)
```

### Output

```
GfG
GfG
```

### Multi-line Strings

If we need a string to span multiple lines then we can use **triple quotes** (''' or ''').

```
s = """I am Learning  
Python String on Tech"""  
print(s)
```

```
s = '''I'm a  
Geek'''  
print(s)
```

### Output

```
I am Learning  
Python String on Tech  
I'm a  
Geek
```

### Accessing characters in Python String

Strings in Python are sequences of characters, so we can access individual characters using **indexing**. Strings are indexed starting from **0** and **-1** from end. This allows us to retrieve specific characters from the string.

Python String syntax indexing

```
s = "Tech"
```

```
# Accesses first character: 'T'  
print(s[0])
```

```
# Accesses 5th character: 'h'  
print(s[4])
```

### Output

```
T  
h
```

**Note:** Accessing an index out of range will cause an **IndexError**. Only integers are allowed as indices and using a float or other types will result in a **TypeError**.

### Access string with [Negative Indexing](#)

Python allows negative address references to access characters from back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

```
s = "Tech"
```

```
# Accesses 3rd character: 'k'  
print(s[-10])
```

```
# Accesses 5th character from end: 'G'
```

```
print(s[-5])
```

## Output

```
k
```

```
G
```

## String Slicing

[Slicing](#) is a way to extract portion of a string by specifying the **start** and **end** indexes. The syntax for slicing is **string[start:end]**, where **start** starting index and **end** is stopping index (excluded).

## String Immutability

**Strings in Python are immutable.** This means that they cannot be changed after they are created. If we need to manipulate strings then we can use methods like **concatenation**, **slicing**, or **formatting** to create new strings based on the original.

```
s = "tech"
```

```
# Trying to change the first character raises an error
```

```
# s[0] = 'l' # Uncommenting this line will cause a TypeError
```

```
# Instead, create a new string
```

```
s = "G" + s[1:]
```

```
print(s)
```

## Output

```
Tech
```

## Deleting a String

In Python, it is not possible to delete individual characters from a string since strings are immutable. However, we can delete an entire string variable using the **del** keyword.

```
s = "tch"
```

```
# Deletes entire string
```

```
del s
```

**Note:** After deleting the string using **del** and if we try to access **s** then it will result in a **NameError** because the variable no longer exists.

## Updating a String

To update a part of a string we need to create a new string since strings are immutable.

```
s = "hello tech"
```

```
# Updating by creating a new string
```

```
s1 = "H" + s[1:]
```

```
# replacnig "tech" with "Tech"
```

```
s2 = s.replace("tech", "Tech")
```

```
print(s1)
```

```
print(s2)
```

### Output

```
Hello tech
```

```
hello Tech
```

### Explanation:

- **For s1**, The original string **s** is sliced from index 1 to end of string and then concatenate "H" to create a new string **s1**.
- **For s2**, we can created a new string s2 and used [replace\(\) method](#) to replace 'tech' with 'Tech'.

### Common String Methods

Python provides a various built-in methods to manipulate strings. Below are some of the most useful methods.

**len()**: The [len\(\)](#) function returns the total number of characters in a string.

```
s = "Tech"
```

```
print(len(s))
```

```
# output: 13
```

### Output

```
13
```

**upper() and lower()**: [upper\(\)](#) method converts all characters to uppercase. [lower\(\)](#) method converts all characters to lowercase.

```
s = "Hello World"
```

```
print(s.upper()) # output: HELLO WORLD
```

```
print(s.lower()) # output: hello world
```

### Output

```
HELLO WORLD
```

```
hello world
```

**strip() and replace():** [strip\(\)](#) removes leading and trailing whitespace from the string and [replace\(old, new\)](#) replaces all occurrences of a specified substring with another.

```
s = " Gfg "
```

```
# Removes spaces from both ends
```

```
print(s.strip())
```

```
s = "Python is fun"
```

```
# Replaces 'fun' with 'awesome'
```

```
print(s.replace("fun", "awesome"))
```

### Output

```
Gfg
Python is awesome
```

To learn more about string methods, please refer to [Python String Methods](#).

## Concatenating and Repeating Strings

We can concatenate strings using [+ operator](#) and repeat them using [\\* operator](#).

Strings can be combined by using **+ operator**.

```
s1 = "Hello"
s2 = "World"
s3 = s1 + " " + s2
print(s3)
```

### Output

```
Hello World
```

We can repeat a string multiple times using **\* operator**.

```
s = "Hello "
print(s * 3)
```

### Output

```
Hello Hello Hello
```

## Formatting Strings

Python provides several ways to include variables inside strings.

### Using f-strings

The simplest and most preferred way to format strings is by using [f-strings](#).

```
name = "Alice"
```

```
age = 22
print(f"Name: {name}, Age: {age}")
```

### Output

```
Name: Alice, Age: 22
```

### Using format()

Another way to format strings is by using [format\(\)](#) method.

```
s = "My name is {} and I am {} years old.".format("Alice", 22)
print(s)
```

### Output

```
My name is Alice and I am 22 years old.
```

### Using in for String Membership Testing

The [in keyword](#) checks if a particular substring is present in a string.

```
s = "Tech"
print("Tech" in s)
print("GfG" in s)
```

### Output

```
True
False
```

## Python Lists

In Python, a **list** is a built-in dynamic sized array (automatically grows and shrinks). We can store all types of items (including another list) in a list.

- Can contain duplicate items.
- Mutable : We can modify, replace or delete the items.
- Ordered : Maintains the order of elements based on how they are added.
- Items can be accessed directly using their position (index), starting from 0.
- May contain mixed type of items, this is possible because a list mainly stores references at contiguous locations and actual items maybe stored at different locations.

# Creating a Python list with different data types

```
a = [10, 20, "GfG", 40, True]
print(a)
```

# Accessing elements using indexing

```
print(a[0]) # 10
print(a[1]) # 20
print(a[2]) # "GfG"
print(a[3]) # 40
```

```
print(a[4]) # True
```

# Checking types of elements

```
print(type(a[2])) # str
print(type(a[4])) # bool
```

**Explanation:**

- The list contains a mix of integers (10, 20, 40), a string ("GfG") and a boolean (True).
- The list is printed and individual elements are accessed using their indexes (starting from 0).
- `type(a[2])` confirms "GfG" is a str.
- `type(a[4])` confirms True is a bool.

Python List

**Note: Lists Store References, Not Values**

*Each element in a list is not stored directly inside the list structure. Instead, the list stores references (pointers) to the actual objects in memory. **Example** (from the image representation).*

- *The list `a` itself is a container with references (addresses) to the actual values.*
- *Python internally creates separate objects for 10, 20, "GfG", 40 and True, then stores their memory addresses inside `a`.*
- *This means that modifying an element doesn't affect other elements but can affect the referenced object if it is mutable*

## Creating a List

Here are some common methods to create a list:

### Using Square Brackets

# List of integers

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

# List of strings

```
b = ['apple', 'banana', 'cherry']
```

# Mixed data types

```
c = [1, 'hello', 3.14, True]
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

### Output

```
[1, 2, 3, 4, 5]
['apple', 'banana', 'cherry']
[1, 'hello', 3.14, True]
```

## Using list() Constructor

We can also create a list by passing an **iterable** (like a **string**, **tuple** or another **list**) to [\*\*list\(\)\*\* function](#).

```
# From a tuple
```

```
a = list([1, 2, 3, 'apple', 4.5])  
print(a)
```

### Output

```
[1, 2, 3, 'apple', 4.5]
```

### Creating List with Repeated Elements

We can create a list with repeated elements using the multiplication operator.

```
# Create a list [2, 2, 2, 2, 2]
```

```
a = [2] * 5
```

```
# Create a list [0, 0, 0, 0, 0, 0, 0]
```

```
b = [0] * 7
```

```
print(a)
```

```
print(b)
```

### Output

```
[2, 2, 2, 2, 2]
```

```
[0, 0, 0, 0, 0, 0, 0]
```

### Accessing List Elements

Elements in a list can be accessed using **indexing**. Python indexes start at **0**, so **a[0]** will access the first element, while **negative indexing** allows us to access elements from the end of the list. Like index -1 represents the last elements of list.

```
a = [10, 20, 30, 40, 50]
```

```
# Access first element
```

```
print(a[0])
```

```
# Access last element
```

```
print(a[-1])
```

### Output

```
10
```

```
50
```

### Adding Elements into List

We can add elements to a list using the following methods:

- **append()**: Adds an element at the end of the list.
- **extend()**: Adds multiple elements to the end of the list.
- **insert()**: Adds an element at a specific position.

# Initialize an empty list

```
a = []
```

# Adding 10 to end of list

```
a.append(10)
```

```
print("After append(10):", a)
```

# Inserting 5 at index 0

```
a.insert(0, 5)
```

```
print("After insert(0, 5):", a)
```

# Adding multiple elements [15, 20, 25] at the end

```
a.extend([15, 20, 25])
```

```
print("After extend([15, 20, 25]):", a)
```

### Output

```
After append(10): [10]
```

```
After insert(0, 5): [5, 10]
```

```
After extend([15, 20, 25]): [5, 10, 15, 20, 25]
```

### Updating Elements into List

We can change the value of an element by accessing it using its index.

```
a = [10, 20, 30, 40, 50]
```

# Change the second element

```
a[1] = 25
```

```
print(a)
```

### Output

```
[10, 25, 30, 40, 50]
```

### Removing Elements from List

We can remove elements from a list using:

- **remove()**: Removes the first occurrence of an element.
- **pop()**: Removes the element at a specific index or the last element if no index is specified.
- **del statement**: Deletes an element at a specified index.

```
a = [10, 20, 30, 40, 50]
```

# Removes the first occurrence of 30

```
a.remove(30)
```

```
print("After remove(30):", a)

# Removes the element at index 1 (20)
popped_val = a.pop(1)
print("Popped element:", popped_val)
print("After pop(1):", a)

# Deletes the first element (10)
del a[0]
print("After del a[0]:", a)
```

### Output

```
After remove(30): [10, 20, 40, 50]
Popped element: 20
After pop(1): [10, 40, 50]
After del a[0]: [40, 50]
```

## Iterating Over Lists

We can iterate the Lists easily by using a **for loop** or other iteration methods. Iterating over lists is useful when we want to do some operation on each item or access specific items based on certain conditions. Let's take an example to iterate over the list using **for loop**.

### Using for Loop

```
a = ['apple', 'banana', 'cherry']

# Iterating over the list
for item in a:
    print(item)
```

### Output

```
apple
banana
cherry
```

*To learn various other methods, please refer to [iterating over lists](#).*

## Nested Lists in Python

A nested list is a list within another list, which is useful for representing matrices or tables. We can access nested elements by chaining indexes.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
```

```
]
```

```
# Access element at row 2, column 3  
print(matrix[1][2])
```

### Output

```
6
```

To learn more, please refer to [Multi-dimensional lists in Python](#)

## List Comprehension in Python

List comprehension is a concise way to create lists using a single line of code. It is useful for applying an operation or filter to items in an iterable, such as a list or range.

```
# Create a list of squares from 1 to 5  
squares = [x**2 for x in range(1, 6)]  
print(squares)
```

### Output

```
[1, 4, 9, 16, 25]
```

### Explanation:

- **for x in range(1, 6):** loops through each number from **1 to 5** (excluding 6).
- **x\*\*2:** squares each number x.
- **[ ]:** collects all the squared numbers into a new list.

## Dictionaries in Python

**Python dictionary** is a data structure that stores the value in **key: value** pairs. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be **immutable**.

**Example:** Here, The data is stored in **key:value** pairs in dictionaries, which makes it easier to find values.

```
d = {1: 'Tech', 2: 'For', 3: 'Tech'}  
print(d)
```

### Output

```
{1: 'Tech', 2: 'For', 3: 'Tech'}
```

## How to Create a Dictionary

Dictionary can be created by placing a sequence of elements within curly {} braces, separated by a 'comma'.

```
d1 = {1: 'Tech', 2: 'For', 3: 'Tech'}  
print(d1)
```

```
# create dictionary using dict() constructor
```

```
d2 = dict(a = "Tech", b = "for", c = "Tech")
print(d2)
```

### Output

```
{1: 'Tech', 2: 'For', 3: 'Tech'}
{'a': 'Tech', 'b': 'for', 'c': 'Tech'}
```

- **Dictionary keys are case sensitive:** the same name but different cases of Key will be treated distinctly.
- **Keys must be immutable:** This means keys can be strings, numbers or tuples but not lists.
- **Keys must be unique:** Duplicate keys are not allowed and any duplicate key will overwrite the previous value.
- Dictionary internally uses [Hashing](#). Hence, operations like search, insert, delete can be performed in **Constant Time**.

*From Python 3.7 Version onward, Python dictionary are Ordered.*

### Accessing Dictionary Items

We can access a value from a dictionary by using the **key** within square brackets or [get\(\)](#) method.

```
d = { "name": "Prajjwal", 1: "Python", (1, 2): [1,2,4] }
```

```
# Access using key
print(d["name"])
```

```
# Access using get()
print(d.get("name"))
```

### Output

```
Prajjwal
Prajjwal
```

### Adding and Updating Dictionary Items

We can add new key-value pairs or update existing keys by using assignment.

```
d = {1: 'Tech', 2: 'For', 3: 'Tech'}
```

```
# Adding a new key-value pair
d["age"] = 22
```

```
# Updating an existing value
d[1] = "Python dict"
```

```
print(d)
```

### Output

```
{1: 'Python dict', 2: 'For', 3: 'Tech', 'age': 22}
```

## Removing Dictionary Items

We can remove items from dictionary using the following methods:

- **del**: Removes an item by key.
- **pop()**: Removes an item by key and returns its value.
- **clear()**: Empties the dictionary.
- **popitem()**: Removes and returns the last key-value pair.

```
d = {1: 'Tech', 2: 'For', 3: 'Tech', 'age': 22}
```

# Using del to remove an item

```
del d["age"]  
print(d)
```

# Using pop() to remove an item and return the value

```
val = d.pop(1)  
print(val)
```

# Using popitem to removes and returns  
# the last key-value pair.

```
key, val = d.popitem()  
print(f"Key: {key}, Value: {val}")
```

# Clear all items from the dictionary

```
d.clear()  
print(d)
```

### Output

```
{1: 'Tech', 2: 'For', 3: 'Tech'}  
  
Tech  
  
Key: 3, Value: Tech  
  
{}
```

## Iterating Through a Dictionary

We can iterate over **keys** [using [keys\(\) method](#)] , **values** [using [values\(\) method](#)] or both [using [item\(\) method](#)] with a [for loop](#).

```
d = {1: 'Tech', 2: 'For', 'age': 22}
```

# Iterate over keys

```
for key in d:  
    print(key)
```

# Iterate over values

```
for value in d.values():  
    print(value)
```

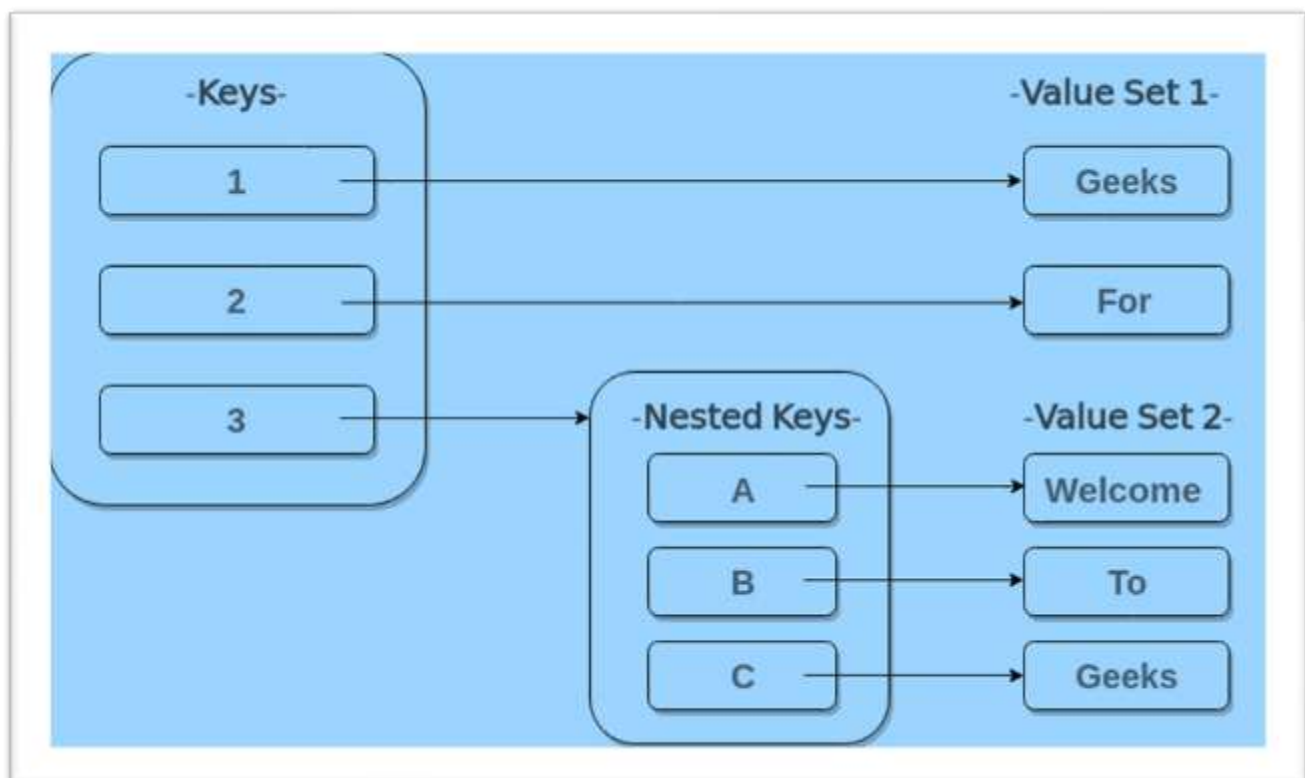
```
# Iterate over key-value pairs
for key, value in d.items():
    print(f'{key}: {value}')
```

### Output

```
1
2
age
Tech
For
22
1: Tech
2: For
age: 22
```

Read in detail: [Ways to Iterating Over a Dictionary](#)

### Nested Dictionaries



### Example of Nested Dictionary:

```
d = {1: 'Tech', 2: 'For',
      3: {'A': 'Welcome', 'B': 'To', 'C': 'Tech'}}
```

```
print(d)
```

## Output

```
{1: 'Tech', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Tech'}}
```

*Read in Detail: [Python Nested Dictionary](#)*

## Copying Dictionaries in Python

A copy of a dictionary can be created using either **shallow copy** or **deep copy** methods. These methods allow duplicating dictionary objects, but they behave differently when it comes to nested data. Let's discuss both in detail.

### 1. Shallow Copy

A **shallow copy** makes a new dictionary with same outer values as the original. But if the dictionary has nested data (like a list or another dictionary), both copies still share that inner data. So, changes to nested parts will affect other.

It is created using **copy.copy()** method from Python's copy module.

#### Example:

This code shows that in a shallow copy, changes to nested data affect both the original and the copy because the nested parts are shared.

```
import copy
original = {'name': 'Alice', 'marks': {'math': 90, 'science': 95}}

# Create a shallow copy
shallow = copy.copy(original)

# Modify nested value in the copy
shallow['marks']['math'] = 100

print("Original:", original)
print("Shallow Copy:", shallow)
```

## Output

```
Original: {'name': 'Alice', 'marks': {'math': 100, 'science': 95}}
```

```
Shallow Copy: {'name': 'Alice', 'marks': {'math': 100, 'science': 95}}
```

#### Explanation:

- **shallow = copy.copy(original):** creates a shallow copy, nested '**marks**' remains shared.
- **shallow['marks']['math'] = 100:** updates '**math**' in the shared nested dictionary.
- **print(original), print(shallow):** both show updated '**math**' value due to shared data.

### 2. Deep Copy

A **deep copy** makes a new dictionary and also creates separate copies of all nested data (like lists or other dictionaries). This means original and copy are completely independent, changes made to nested parts do not affect other.

It is created using **copy.deepcopy()** method from Python's copy module.

### Example:

This Example shows that a deep copy creates a fully independent copy of both the outer and nested data, so changes in the copy do not affect the original.

```
import copy
original = {'name': 'Alice', 'marks': {'math': 90, 'science': 95}}

# Create a deep copy
deep = copy.deepcopy(original)

# Modify nested value in the deep copy
deep['marks']['math'] = 100

print("Original:", original)
print("Deep Copy:", deep)
```

### Output

```
Original: {'name': 'Alice', 'marks': {'math': 90, 'science': 95}}
Deep Copy: {'name': 'Alice', 'marks': {'math': 100, 'science': 95}}
```

### Explanation:

- **deep = copy.deepcopy(original):** creates a deep copy, nested 'marks' is also copied separately.
- **deep['marks']['math'] = 100:** updates 'math' in the deep copy's nested dictionary only.
- **print(original), print(deep):** original remains unchanged, only deep copy shows the updated 'math' value.

## Python Tuples

A tuple in Python is an immutable ordered collection of elements.

- Tuples are similar to lists, but unlike lists, they cannot be changed after their creation (i.e., they are immutable).
- Tuples can hold elements of different data types.
- The main characteristics of tuples are being **ordered**, **heterogeneous** and **immutable**.

### Creating a Tuple

A tuple is created by placing all the items inside parentheses (), separated by commas. A tuple can have any number of items and they can be of different [data types](#).

### Example:

```
tup = ()
print(tup)

# Using String
tup = ('Tech', 'For')
print(tup)

# Using List
li = [1, 2, 4, 5, 6]
print(tuple(li))

# Using Built-in Function
tup = tuple('Tech')
print(tup)
```

### Output

```
()
('Tech', 'For')
(1, 2, 4, 5, 6)
('T', 'e', 'c', 'h')
```

Let's understand tuple in detail:

### Creating a Tuple with Mixed Datatypes.

Tuples can contain elements of various data types, including other tuples, [lists](#), [dictionaries](#) and even [functions](#).

### Example:

```
tup = (5, 'Welcome', 7, 'Tech')
print(tup)
```

### # Creating a Tuple with nested tuples

```
tup1 = (0, 1, 2, 3)
tup2 = ('python', 'geek')
tup3 = (tup1, tup2)
print(tup3)
```

### # Creating a Tuple with repetition

```
tup1 = ('Tech',) * 3
print(tup1)
```

### # Creating a Tuple with the use of loop

```
tup = ('Tech')
n = 5
for i in range(int(n)):
```

```
tup = (tup,)
print(tup)
```

## Output

```
(5, 'Welcome', 7, 'Tech')
((0, 1, 2, 3), ('python', 'geek'))
('Tech', 'Tech', 'Tech')
('Tech',)
(('Tech',),)
((( 'Tech',),),)
(((( 'Tech',),),),)
((((('Tech',),),),),)
```

## Python Tuple Basic Operations

Below are the Python tuple operations.

- Accessing of Python Tuples
- Concatenation of Tuples
- Slicing of Tuple
- Deleting a Tuple

### Accessing of Tuples

We can access the elements of a tuple by using indexing and [slicing](#), similar to how we access elements in a list. Indexing starts at 0 for the first element and goes up to n-1, where n is the number of elements in the tuple. Negative indexing starts from -1 for the last element and goes backward.

#### Example:

##### # Accessing Tuple with Indexing

```
tup = tuple("Tech")
print(tup[0])
```

##### # Accessing a range of elements using slicing

```
print(tup[1:4])
print(tup[:3])
```

##### # Tuple unpacking

```
tup = ("Tech", "For", "Tech")
```

##### # This line unpack values of Tuple1

```
a, b, c = tup
print(a)
print(b)
print(c)
```

## Output

```
T
('e', 'c', 'h')
('e', 'c', 'h')
Tech
For
Tech
```

## Concatenation of Tuples

Tuples can be concatenated using the + operator. This operation combines two or more tuples to create a new tuple.

**Note:** Only the same datatypes can be combined with concatenation, an error arises if a list and a tuple are combined.

```
tup1 = (0, 1, 2, 3)
tup2 = ('Tech', 'For', 'Tech')

tup3 = tup1 + tup2
print(tup3)
```

## Output

```
(0, 1, 2, 3, 'Tech', 'For', 'Tech')
```

## Slicing of Tuple

[Slicing a tuple](#) means creating a new tuple from a subset of elements of the original tuple. The slicing syntax is tuple[start:stop:step].

**Note-** Negative Increment values can also be used to reverse the sequence of Tuples.

```
tup = tuple('TECH')
```

```
# Removing First element
```

```
print(tup[1:])
```

```
# Reversing the Tuple
```

```
print(tup[::-1])
```

```
# Printing elements of a Range
```

```
print(tup[4:9])
```

## Deleting a Tuple

Since tuples are immutable, we cannot delete individual elements of a tuple. However, we can delete an entire tuple using [del statement](#).

**Note:** Printing of Tuple after deletion results in an Error.

```
tup = (0, 1, 2, 3, 4)
```

```
del tup
```

```
print(tup)
```

### Output

*ERROR!*

Traceback	(most	recent	call	last):
File	"<main.py>",	line	6,	in
<module>				
NameError: name 'tup' is not defined				

## Tuple Unpacking with Asterisk (\*)

In Python, the "\*" operator can be used in tuple unpacking to grab multiple items into a list. This is useful when you want to extract just a few specific elements and collect the rest together.

```
tup = (1, 2, 3, 4, 5)
```

```
a, *b, c = tup
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

### Output

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

### Explanation:

- **a** gets the first item.
- **c** gets the last item.
- **\*b** collects everything in between into a list.

## Sets in Python

A Set in Python is used to store a collection of items with the following properties.

- No duplicate elements. If try to insert the same item again, it overwrites previous one.
- An unordered collection. When we access all items, they are accessed without any specific order and we cannot access items using indexes as we do in lists.

- Internally use [hashing](#) that makes set efficient for search, insert and delete operations. It gives a major advantage over a [list](#) for problems with these operations.
- Mutable, meaning we can add or remove elements after their creation, the individual elements within the set cannot be changed directly.

## Example of Python Sets

```
s = {10, 50, 20}
print(s)
print(type(s))
```

### Output

```
{10, 50, 20}
<class 'set'>
```

**Note :** There is no specific order for set elements to be printed

## Type Casting with Python Set method

The Python set() method is used for type casting.

```
# typecasting list to set
s = set(["a", "b", "c"])
print(s)
```

```
# Adding element to the set
s.add("d")
print(s)
```

### Output

```
{'c', 'b', 'a'}
{'d', 'c', 'b', 'a'}
```

## Check unique and Immutable with Python Set

Python sets cannot have duplicate values. While you cannot modify the individual elements directly, you can still add or remove elements from the set.

```
# Python program to demonstrate that
# a set cannot have duplicate values
# and we cannot change its items
```

```
# a set cannot have duplicate values
s = {"Tech", "for", "Tech"}
print(s)
```

```
# values of a set cannot be changed
s[1] = "Hello"
```

```
print(s)
```

### Output:

The first code explains that the set cannot have a duplicate value. Every item in it is a unique value.

The second code generates an error because we cannot assign or change a value once the set is created. We can only add or delete items in the set.

```
{ 'Tech', 'for' }  
TypeError: 'set' object does not support item assignment
```

## Heterogeneous Element with Python Set

Python sets can store heterogeneous elements in it, i.e., a set can store a mixture of string, integer, boolean, etc datatypes.

```
# Python example demonstrate that a set  
# can store heterogeneous elements  
s = {"Tech", "for", 10, 52.7, True}  
print(s)
```

### Output

```
{True, 'for', 'Tech', 10, 52.7}
```

## Python Frozen Sets

**Frozen sets** in Python are immutable objects that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied. It can be done with [frozenset\(\)](#) [method](#) in Python.

While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.

If no parameters are passed, it returns an empty frozenset.

```
# Python program to demonstrate differences  
# between normal and frozen set
```

```
# Same as {"a", "b", "c"}  
s = set(["a", "b", "c"])
```

```
print("Normal Set")  
print(s)
```

```
# A frozen set  
fs = frozenset(["e", "f", "g"])
```

```
print("\nFrozen Set")  
print(fs)
```

```
# Uncommenting below line would cause error as
# we are trying to add element to a frozen set
# fs.add("h")
```

## Output

Normal Set

```
set(['a', 'c', 'b'])
```

Frozen Set

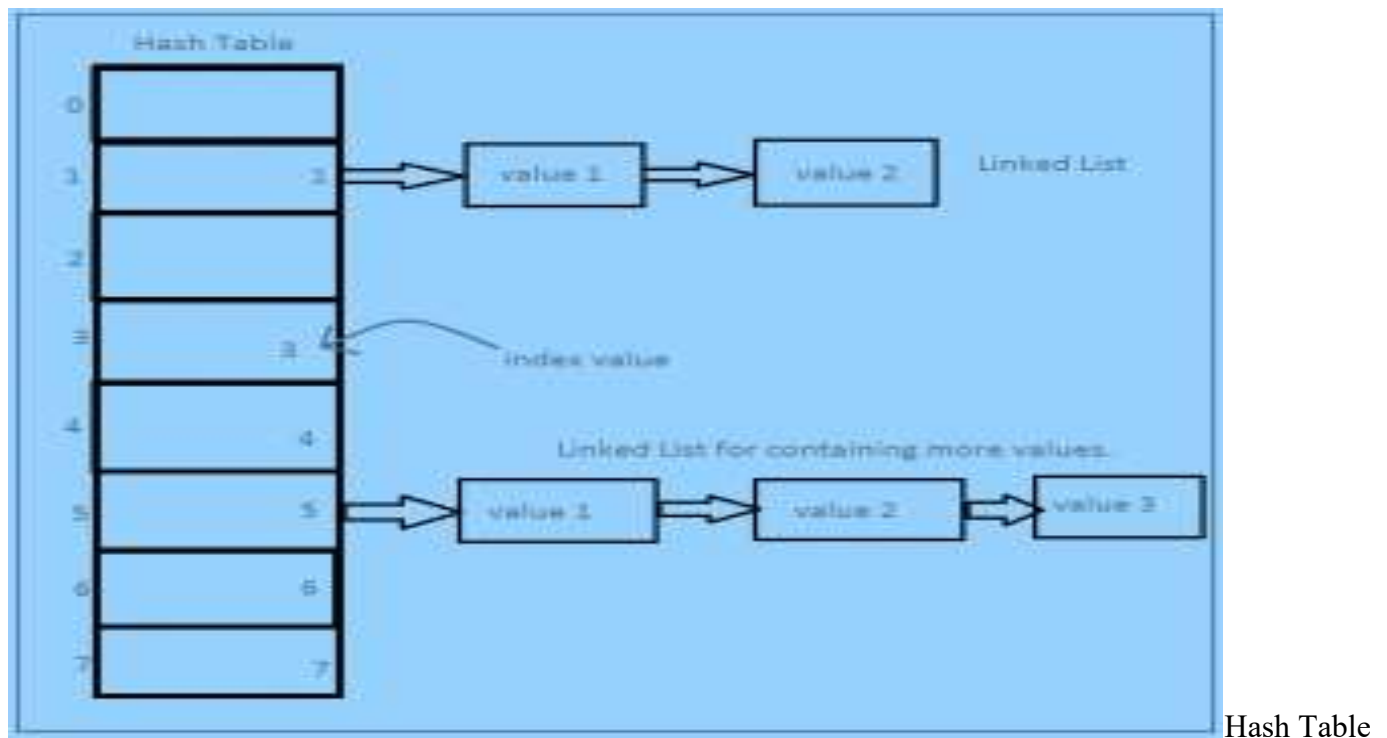
```
frozenset(['e', 'g', 'f'])
```

## Internal working of Set

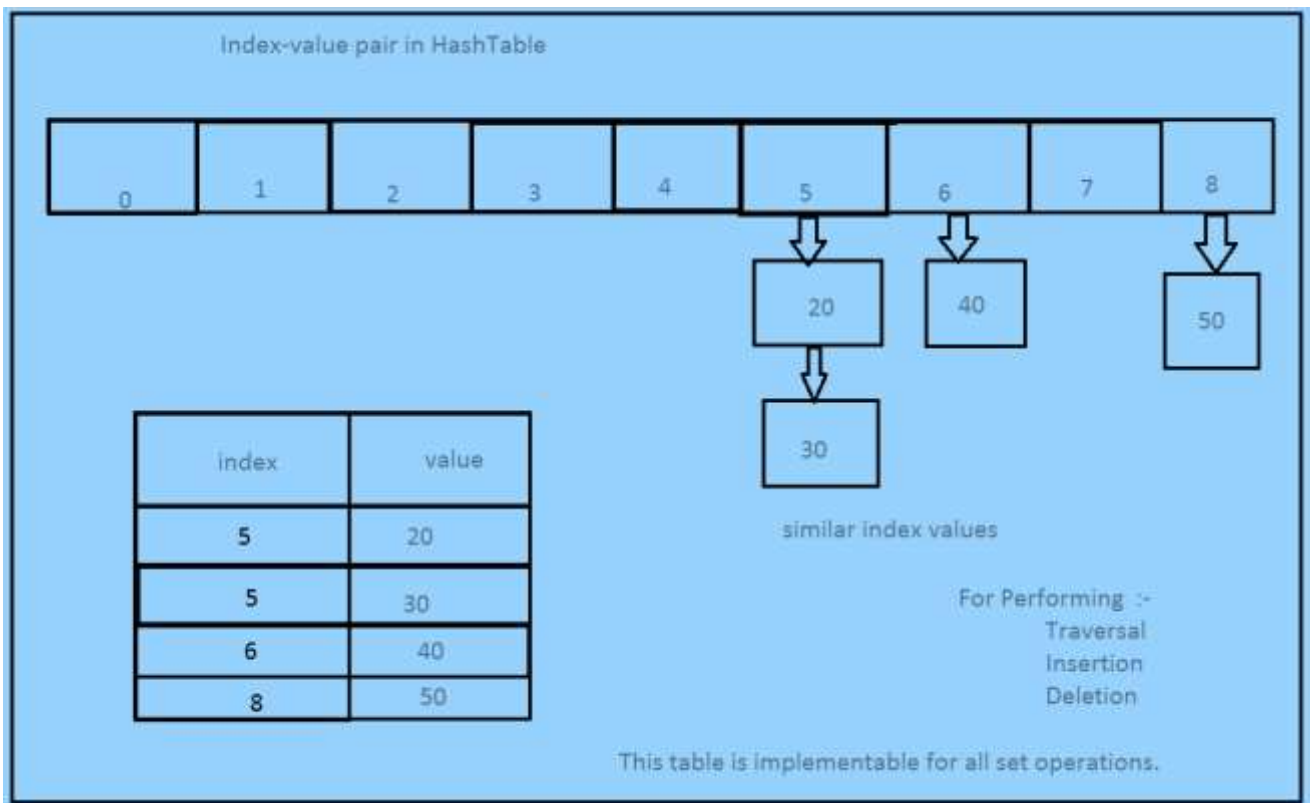
This is based on a data structure known as a hash table. If Multiple values are present at the same index position, then the value is appended to that index position, to form a Linked List.

In, Python Sets are implemented using a dictionary with dummy variables, where key beings the members set with greater optimizations to the time complexity.

## Set Implementation:



Sets with Numerous operations on a single HashTable:



## Hashing

### Methods for Sets

#### Adding elements to Python Sets

Insertion in the set is done through the [set.add\(\)](#) function, where an appropriate record value is created to store in the hash table. Same as checking for an item, i.e.,  $O(1)$  on average. However, in worst case it can become  $O(n)$ .

```
# A Python program to
# demonstrate adding elements
# in a set
```

```
# Creating a Set
people = {"Jay", "Idrish", "Archi"}
```

```
print("People:", end = " ")
print(people)
```

```
# This will add Daxit
# in the set
people.add("Daxit")
```

```
# Adding elements to the
# set using iterator
for i in range(1, 6):
    people.add(i)
```

```
print("\nSet after adding element:", end = " ")
print(people)
```

### Output

```
People: {'Idrish', 'Archi', 'Jay'}

Set after adding element: {1, 2, 3, 4, 5, 'Daxit', 'Archi', 'Jay', 'Idrish'}
```

### Union operation on Python Sets

Two sets can be merged using union() function or | operator. Both Hash Table values are accessed and traversed with merge operation perform on them to combine the elements, at the same time duplicates are removed. The Time Complexity of this is  **$O(\text{len}(s1) + \text{len}(s2))$**  where s1 and s2 are two sets whose union needs to be done.

```
# Python Program to
# demonstrate union of
# two sets
```

```
people = {"Jay", "Idrish", "Archil"}
vampires = {"Karan", "Arjun"}
dracula = {"Deepanshu", "Raju"}
```

```
# Union using union()
# function
population = people.union(vampires)
```

```
print("Union using union() function")
print(population)
```

```
# Union using "|"
# operator
population = people|dracula
```

```
print("\nUnion using '|' operator")
print(population)
```

### Output

```
Union using union() function
{'Idrish', 'Arjun', 'Jay', 'Karan', 'Archil'}

Union using '|' operator
{'Idrish', 'Deepanshu', 'Raju', 'Jay', 'Archil'}
```

## Intersection operation on Python Sets

This can be done through `intersection()` or `&` operator. Common Elements are selected. They are similar to iteration over the Hash lists and combining the same values on both the Table. Time Complexity of this is  $O(\min(\text{len}(s1), \text{len}(s2)))$  where `s1` and `s2` are two sets whose union needs to be done.

# Python program to

# demonstrate intersection

# of two sets

```
set1 = set()
```

```
set2 = set()
```

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

```
    set1.add(i)
```

```
for i in range(3,9):
```

```
    set2.add(i)
```

# Intersection using

# `intersection()` function

```
set3 = set1.intersection(set2)
```

```
print("Intersection using intersection() function")
```

```
print(set3)
```

# Intersection using

# `"&"` operator

```
set3 = set1 & set2
```

```
print("\nIntersection using '&' operator")
```

```
print(set3)
```

### Output

```
Intersection using intersection() function
```

```
{3, 4}
```

```
Intersection using '&' operator
```

```
{3, 4}
```

## Finding Differences of Sets in Python

To find differences between sets. Similar to finding differences in the linked list. This is done through `difference()` or `-` operator. Time complexity of finding difference `s1 - s2` is  $O(\text{len}(s1))$

# Python program to

```

# demonstrate difference
# of two sets

set1 = set()
set2 = set()

for i in range(5):
    set1.add(i)

for i in range(3,9):
    set2.add(i)

# Difference of two sets
# using difference() function
set3 = set1.difference(set2)

print(" Difference of two sets using difference() function")
print(set3)

# Difference of two sets
# using '-' operator
set3 = set1 - set2

print("\nDifference of two sets using '-' operator")
print(set3)

```

## Output

```

Difference of two sets using difference() function
{0, 1, 2}

Difference of two sets using '-' operator
{0, 1, 2}

```

## Clearing Python Sets

Set Clear() method empties the whole set inplace.

```

# Python program to
# demonstrate clearing
# of set

```

```

set1 = {1,2,3,4,5,6}

print("Initial set")
print(set1)

```

```
# This method will remove  
# all the elements of the set  
set1.clear()
```

```
print("\nSet after using clear() function")  
print(set1)
```

### Output

```
Initial set  
{1, 2, 3, 4, 5, 6}  
  
Set after using clear() function  
set()
```

**However, there are two major pitfalls in Python sets:**

1. The set doesn't maintain elements in any particular order.
2. Only instances of immutable types can be added to a Python set.

### Time complexity of Sets

Operation	Average case	Worst Case	notes
$x \text{ in } s$	$O(1)$	$O(n)$	
Union $s t$	$O(\text{len}(s)+\text{len}(t))$		
Intersection $s\&t$	$O(\min(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$	replace "min" with "max" if t is not a set
Multiple intersection $s1\&s2\&...\&sn$		$(n-1)*O(l)$ where l is $\max(\text{len}(s1),...,\text{len}(sn))$	
Difference $s-t$	$O(\text{len}(s))$		

### Operators for Sets

Sets and frozen sets support the following operators:

Operators	Notes
$\text{key in } s$	containment check
$\text{key not in } s$	non-containment check

Operators	Notes
$s1 == s2$	$s1$ is equivalent to $s2$
$s1 != s2$	$s1$ is not equivalent to $s2$
$s1 \leq s2$	$s1$ is subset of $s2$
$s1 < s2$	$s1$ is proper subset of $s2$
$s1 \geq s2$	$s1$ is superset of $s2$
$s1 > s2$	$s1$ is proper superset of $s2$
$s1   s2$	the union of $s1$ and $s2$
$s1 \& s2$	the intersection of $s1$ and $s2$
$s1 - s2$	the set of elements in $s1$ but not $s2$
$s1 \wedge s2$	the set of elements in precisely one of $s1$ or $s2$

### Problems based on Set

- [Distinct Elements in an Array](#)
- [Union of Two Arrays](#)
- [Intersection of Two Arrays](#)
- [Distinct Elements in an Array](#)
- [Union of Two Arrays](#)
- [Intersection of Two Arrays](#)
- [Repeating Elements](#)
- [Check if an Array is Subset of other](#)
- [Check Pair with Target Sum](#)
- [Check for Disjoint Arrays or Sets](#)
- [Duplicate within K Distance](#)
- [Longest Consecutive Sequence](#)
- [Check for Disjoint Arrays or Sets](#)
- [Duplicate within K Distance in an Array](#)
- [2 Sum - Check for Pair with target sum](#)
- [Longest Consecutive Sequence](#)
- [Only Repeating Element From 1 To n-](#)

### Python Collections Module

The collections module in Python provides specialized containers (different from general purpose built-in containers like dict, list, tuple and set). These specialized containers are designed to address specific programming needs efficiently and offer additional functionalities.

## Why do we need Collections Module?

1. Provides specialized container data types beyond built-in types like list, dict and tuple.
2. Includes efficient alternatives such as deque, Counter, OrderedDict, defaultdict and namedtuple.
3. Simplifies complex data structure handling with cleaner and faster implementations.
4. Helps in frequency counting, queue operations and structured records with minimal code.
5. Ideal for improving performance and code readability in data-heavy applications.

## Counters

A [counter](#) is a sub-class of the dictionary. It is used to keep the count of the elements in an iterable in the form of an unordered dictionary where the key represents element in the iterable and value represents the count of that element in the iterable.

**Note:** It is equivalent to bag or multiset of other languages.

### Syntax:

```
class collections.Counter([iterable-or-mapping])
```

## Initializing Counter Objects

The counter object can be initialized using counter() function and this function can be called in one of the following ways:

- With a sequence of items
- With a dictionary containing keys and counts
- With keyword arguments mapping string names to counts

**Example:** Program to demonstrate different ways to create a Counter

```
from collections import Counter
```

```
# Creating Counter from a list (sequence of items)
```

```
print(Counter(['B','B','A','B','C','A','B','B','A','C']))
```

```
# Creating Counter from a dictionary
```

```
print(Counter({'A':3, 'B':5, 'C':2}))
```

```
# Creating Counter using keyword arguments
```

```
print(Counter(A=3, B=5, C=2))
```

## Output

```
Counter({'B': 5, 'A': 3, 'C': 2})
```

```
Counter({'B': 5, 'A': 3, 'C': 2})
```

```
Counter({'B': 5, 'A': 3, 'C': 2})
```

## OrderedDict

An [OrderedDict](#) is a dictionary that preserves the order in which keys are inserted. While regular dictionaries do this from Python 3.7+, OrderedDict also offers extra features like moving re-inserted keys to the end making it useful for order-sensitive operations.

### Syntax:

```
class collections.OrderDict()
```

### Example:

```
from collections import OrderedDict
print("This is a Dict:\n")
d = {}
d['a'] = 1
d['b'] = 2
d['c'] = 3
d['d'] = 4
```

```
for key, value in d.items():
    print(key, value)
```

```
print("\nThis is an Ordered Dict:\n")
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4
```

```
for key, value in od.items():
    print(key, value)
```

### Output

```
This is a Dict:
```

```
a 1
b 2
c 3
d 4
```

```
This is an Ordered Dict:
```

```
a 1
b 2
c 3
```

While deleting and re-inserting the same key will push the key to the last to maintain the order of insertion of the key.

### Example (Deleting and reinserting a key):

```
from collections import OrderedDict
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4
```

```
print('Before Deleting')
for key, value in od.items():
    print(key, value)
```

```
# deleting element
od.pop('a')
```

```
# Re-inserting the same
od['a'] = 1
```

```
print('\nAfter re-inserting')
for key, value in od.items():
    print(key, value)
```

### Output

```
Before Deleting
```

```
a 1
```

```
b 2
```

```
c 3
```

```
d 4
```

```
After re-inserting
```

```
b 2
```

```
c 3
```

```
d 4
```

```
a 1
```

### DefaultDict

A [DefaultDict](#) is also a sub-class to dictionary. It is used to provide some default values for the key that does not exist and never raises a `KeyError`.

### Syntax:

```
class collections.defaultdict(default_factory)
```

`default_factory` is a function that provides the default value for the dictionary created. If this parameter is absent then the `KeyError` is raised.

### Initializing DefaultDict Objects

`DefaultDict` objects can be initialized using `DefaultDict()` method by passing the data type as an argument.

### Example:

```
from collections import defaultdict

# Creating a defaultdict with default value of 0 (int)
d = defaultdict(int)
L = [1, 2, 3, 4, 2, 4, 1, 2]

# Counting occurrences of each element in the list
for i in L:
    d[i] += 1 # No need to check key existence; default is 0

print(d)
```

### Output

```
defaultdict(<class 'int'>, {1: 2, 2: 3, 3: 1, 4: 2})
```

### Example 2:

```
from collections import defaultdict

# Defining a dict
d = defaultdict(list)

for i in range(5):
    d[i].append(i)

print("Dictionary with values as list:")
print(d)
```

### Output

```
Dictionary with values as list:
defaultdict(<class 'list'>, {0: [0], 1: [1], 2: [2], 3: [3], 4: [4]})
```

### ChainMap

A [ChainMap](#) encapsulates many dictionaries into a single unit and returns a list of dictionaries.

### Syntax:

```
class collections.ChainMap(dict1, dict2)
```

### Example:

```
from collections import ChainMap
```

```
d1 = {'a': 1, 'b': 2}
```

```
d2 = {'c': 3, 'd': 4}
```

```
d3 = {'e': 5, 'f': 6}
```

```
# Defining the chainmap
```

```
c = ChainMap(d1, d2, d3)
```

```
print(c)
```

### Output

```
ChainMap({'a': 1, 'b': 2}, {'c': 3, 'd': 4}, {'e': 5, 'f': 6})
```

### Accessing Keys and Values from ChainMap

Values from ChainMap can be accessed using the key name. They can also be accessed by using the `keys()` and `values()` method.

### Example:

```
from collections import ChainMap
```

```
d1 = {'a': 1, 'b': 2}
```

```
d2 = {'c': 3, 'd': 4}
```

```
d3 = {'e': 5, 'f': 6}
```

```
# Defining the chainmap
```

```
c = ChainMap(d1, d2, d3)
```

```
# Accessing Values using key name
```

```
print(c['a'])
```

```
# Accessing values using values()
```

```
print(c.values())
```

```
# Accessing keys using keys()
```

```
print(c.keys())
```

### Output

```
1
ValuesView(ChainMap({'a': 1, 'b': 2}, {'c': 3, 'd': 4}, {'e': 5, 'f': 6}))
KeysView(ChainMap({'a': 1, 'b': 2}, {'c': 3, 'd': 4}, {'e': 5, 'f': 6}))
```

## Adding new dictionary

A new dictionary can be added by using the **new\_child()** method. The newly added dictionary is added at the beginning of the ChainMap.

### Example:

```
import collections

# initializing dictionaries
dic1 = { 'a' : 1, 'b' : 2 }
dic2 = { 'b' : 3, 'c' : 4 }
dic3 = { 'f' : 5 }

# initializing ChainMap
chain = collections.ChainMap(dic1, dic2)

# printing chainMap
print ("All the ChainMap contents are :")
print (chain)

# using new_child() to add new dictionary
chain1 = chain.new_child(dic3)

# printing chainMap
print ("Displaying new ChainMap :")
print (chain1)
```

### Output

```
All the ChainMap contents are:
ChainMap({'a': 1, 'b': 2}, {'b': 3, 'c': 4})
Displaying new ChainMap :
ChainMap({'f': 5}, {'a': 1, 'b': 2}, {'b': 3, 'c': 4})
```

## NamedTuple

A [NamedTuple](#) is like a regular tuple but with named fields, making data more readable and accessible. Instead of using indexes, you can access elements by name (e.g., student.fname), which improves code clarity and ease of use.

### Syntax:

```
class collections.namedtuple(typename, field_names)
```

### Example:

```
from collections import namedtuple

# Declaring namedtuple()
```

```
Student = namedtuple('Student',['name','age','DOB'])
```

```
# Adding values
```

```
S = Student('Nandini','19','2541997')
```

```
# Access using index
```

```
print ("The Student age using index is : ",end = "")
```

```
print (S[1])
```

```
# Access using name
```

```
print ("The Student name using keyname is : ",end = "")
```

```
print (S.name)
```

## Output

```
The Student age using index is : 19
```

```
The Student name using keyname is : Nandini
```

## Conversion Operations

1. **\_make()**: This function is used to return a namedtuple() from the iterable passed as argument.
2. **\_asdict()**: This function returns the [OrderedDict\(\)](#) as constructed from the mapped values of namedtuple().

### Example:

```
from collections import namedtuple
```

```
# Declaring namedtuple()
```

```
Student = namedtuple('Student',['name','age','DOB'])
```

```
# Adding values
```

```
S = Student('Nandini','19','2541997')
```

```
# initializing iterable
```

```
li = ['Manjeet', '19', '411997' ]
```

```
# initializing dict
```

```
di = { 'name' : "Nikhil", 'age' : 19 , 'DOB' : '1391997' }
```

```
# using _make() to return namedtuple()
```

```
print ("The namedtuple instance using iterable is : ")
```

```
print (Student._make(li))
```

```
# using _asdict() to return an OrderedDict()
```

```
print ("The OrderedDict instance using namedtuple is : ")
```

```
print (S._asdict())
```

## Output

```
The namedtuple instance using iterable is :  
Student(name='Manjeet', age='19', DOB='411997')  
The OrderedDict instance using namedtuple is :  
{'name': 'Nandini', 'age': '19', 'DOB': '2541997'}
```

## Deque

[Deque \(Doubly Ended Queue\)](#) is the optimized list for quicker append and pop operations from both sides of the container. It provides  $O(1)$  time complexity for append and pop operations as compared to list with  $O(n)$  time complexity.

### Syntax:

```
class collections.deque(list)
```

This function takes the list as an argument.

### Example:

```
from collections import deque
```

```
# Declaring deque
```

```
queue = deque(['name','age','DOB'])
```

```
print(queue)
```

## Output

```
deque(['name', 'age', 'DOB'])
```

## Inserting Elements

Elements in deque can be inserted from both ends. To insert the elements from right `append()` method is used and to insert the elements from the left `appendleft()` method is used.

### Example:

```
from collections import deque
```

```
# Initializing deque with initial values
```

```
de = deque([1, 2, 3])
```

```
# Append 4 to the right end of deque
```

```
de.append(4)
```

```
# Print deque after appending to the right
```

```
print("The deque after appending at right is :")
```

```
print(de)
```

```
# Append 6 to the left end of deque
```

```
de.appendleft(6)
```

```
# Print deque after appending to the left
```

```
print("The deque after appending at left is :")
```

```
print(de)
```

### Output

```
The deque after appending at right is :
```

```
deque([1, 2, 3, 4])
```

```
The deque after appending at left is :
```

```
deque([6, 1, 2, 3, 4])
```

### Removing Elements

Elements can also be removed from the deque from both the ends. To remove elements from right use `pop()` method and to remove elements from the left use `popleft()` method.

#### Example:

```
from collections import deque
```

```
# Initialize deque with initial values
```

```
de = deque([6, 1, 2, 3, 4])
```

```
# Delete element from the right end (removes 4)
```

```
de.pop()
```

```
# Print deque after deletion from the right
```

```
print("The deque after deleting from right is :")
```

```
print(de)
```

```
# Delete element from the left end (removes 6)
```

```
de.popleft()
```

```
# Print deque after deletion from the left
```

```
print("The deque after deleting from left is :")
```

```
print(de)
```

### Output

```
The deque after deleting from right is :
```

```
deque([6, 1, 2, 3])
```

```
The deque after deleting from left is :
```

```
deque([1, 2, 3])
```

### UserDict

[UserDict](#) is a dictionary-like container that acts as a wrapper around the dictionary objects. This container is used when someone wants to create their own dictionary with some modified or new functionality.

### Syntax:

```
class collections.UserDict([initialdata])
```

### Example:

```
from collections import UserDict

# Creating a dictionary where deletion is not allowed
class MyDict(UserDict):

    # Prevents using 'del' on dictionary
    def __del__(self):
        raise RuntimeError("Deletion not allowed")

    # Prevents using pop() on dictionary
    def pop(self, s=None):
        raise RuntimeError("Deletion not allowed")

    # Prevents using popitem() on dictionary
    def popitem(self, s=None):
        raise RuntimeError("Deletion not allowed")

# Create an instance of MyDict
d = MyDict({'a': 1, 'b': 2, 'c': 3})

d.pop(1)
```

### Output:

```
Traceback (most recent call last):
File "/home/f8db849e4cf1e58177983b2b6023c1a3.py", line 32, in <module>
d.pop(1)
File "/home/f8db849e4cf1e58177983b2b6023c1a3.py", line 20, in pop
raise RuntimeError("Deletion not allowed")
RuntimeError: Deletion not allowed
Exception ignored in: <bound method MyDict.__del__ of {'a': 1, 'b': 2, 'c': 3}>
Traceback (most recent call last):
```

```
File      "/home/f8db849e4cf1e58177983b2b6023c1a3.py",    line    15,    in    __del__
RuntimeError: Deletion not allowed
```

## UserList

[UserList](#) is a list like container that acts as a wrapper around the list objects. This is useful when someone wants to create their own list with some modified or additional functionality.

### Syntax:

```
class collections.UserList([list])
```

### Example:

```
from collections import UserList

# Creating a list where deletion is not allowed
class MyList(UserList):

    # Prevents using remove() on list
    def remove(self, s=None):

        raise RuntimeError("Deletion not allowed")

    # Prevents using pop() on list
    def pop(self, s=None):

        raise RuntimeError("Deletion not allowed")

# Create an instance of MyList
L = MyList([1, 2, 3, 4])
print("Original List")

# Append 5 to the list
L.append(5)
print("After Insertion")
print(L)

# Attempt to remove an item (will raise error)
L.remove()
```

### Output:

<i>Original</i>					<i>List</i>
<i>After</i>					<i>Insertion</i>
<i>[1, 2, 3, 4, 5]</i>					
<i>Traceback</i>	<i>(most</i>	<i>recent</i>	<i>call</i>		<i>last):</i>
<i>File</i>	<i>"/home/c90487eefa7474c0566435269f50a52a.py",</i>	<i>line</i>	<i>33,</i>	<i>in</i>	<i>&lt;module&gt;</i>
<i>L.remove()</i>					
<i>File</i>	<i>"/home/c90487eefa7474c0566435269f50a52a.py",</i>	<i>line</i>	<i>15,</i>	<i>in</i>	<i>remove</i>
<i>raise</i>	<i>RuntimeError("Deletion</i>	<i>not</i>			<i>allowed")</i>
<i>RuntimeError: Deletion not allowed</i>					

## UserString

[UserString](#) is a string like container and just like UserDict and UserList it acts as a wrapper around string objects. It is used when someone wants to create their own strings with some modified or additional functionality.

### Syntax:

```
class collections.UserString(seq)
```

### Example:

```
from collections import UserString
```

```
# Creating a Mutable String
```

```
class Mystring(UserString):
```

```
    # Function to append to string
```

```
    def append(self, s):
```

```
        self.data += s
```

```
    # Function to remove from string
```

```
    def remove(self, s):
```

```
        self.data = self.data.replace(s, "")
```

```
# Driver's code
```

```
s1 = Mystring("Tech")
```

```
print("Original String:", s1.data)
```

```
# Appending to string
```

```
s1.append("s")
```

```
print("String After Appending:", s1.data)
```

```
# Removing from string
```

```
s1.remove("e")
```

```
print("String after Removing:", s1.data)
```

### Output

```
Original String: Tech
```

```
String After Appending: Techs
```

```
String after Removing: Gkss
```

## Comprehensions in Python

Comprehensions in Python provide a concise and efficient way to create new sequences from existing ones. They enhance code readability and reduce the need for lengthy loops.

### Why do we need Comprehensions

- **Encourages Modular Thinking:** Promotes writing logic in smaller, reusable expressions.
- **Widely Used in Real-World Code:** Found in data science, web development and automation scripts.
- **Easier Debugging & Testing:** Fewer lines reduce the surface area for bugs.
- **Seamless with Other Python Features:** Integrates well with functions like `zip()`, `enumerate()` and `lambda`.

### Types of Comprehensions in Python

Python offers different types of comprehensions to simplify creation of data structures in a clean, readable way. Let's explore each type with simple examples.

#### 1. List Comprehensions

[List comprehensions](#) allow for the creation of lists in a single line, improving efficiency and readability. They follow a specific pattern to transform or filter data from an existing iterable.

##### Syntax:

```
[expression for item in iterable if condition]
```

##### Where:

- **expression:** Operation applied to each item.
- **item:** Variable representing the element from the iterable.
- **iterable:** The source collection.
- **condition (optional):** A filter to include only specific items.

**Example 1:** Generating a list of even numbers

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
res = [num for num in a if num % 2 == 0]
print(res)
```

##### Output

```
[2, 4, 6, 8]
```

**Explanation:** This creates a list of even numbers by filtering elements from **a** that are divisible by 2.

**Example 2:** Creating a list of squares

```
res = [num**2 for num in range(1, 6)]  
print(res)
```

### Output

```
[1, 4, 9, 16, 25]
```

**Explanation:** This generates a list of squares for numbers from 1 to 5.

## 2. Dictionary comprehension

[Dictionary Comprehensions](#) are used to construct dictionaries in a compact form, making it easy to generate key-value pairs dynamically based on an iterable.

### Syntax:

```
{key_expression: value_expression for item in iterable if condition}
```

### Where:

- **key\_expression:** Determines the dictionary key.
- **value\_expression:** Computes the value.
- **iterable:** The source collection.
- **condition (optional):** Filters elements before adding them.

**Example 1:** Creating a dictionary of numbers and their cubes

```
res = {num: num**3 for num in range(1, 6)}  
print(res)
```

### Output

```
{1: 1, 2: 8, 3: 27, 4: 64, 5: 125}
```

**Explanation:** This creates a dictionary where keys are numbers from 1 to 5 and values are their cubes.

**Example 2:** Mapping states to capitals

```
a = ["Texas", "California", "Florida"] # states  
b = ["Austin", "Sacramento", "Tallahassee"] # capital
```

```
res = {state: capital for state, capital in zip(a, b)}  
print(res)
```

### Output

```
{'Texas': 'Austin', 'California': 'Sacramento', 'Florida': 'Tallahassee'}
```

**Explanation:** [zip\(\) function](#) pairs each state with its corresponding capital, creating a dictionary.

## 3. Set comprehensions

[Set](#) Comprehensions are similar to list comprehensions but result in sets, automatically eliminating duplicate values while maintaining a concise syntax.

### Syntax:

```
{expression for item in iterable if condition}
```

**Where:**

- **expression:** The operation applied to each item.
- **iterable:** The source collection.
- **condition (optional):** Filters elements before adding them.

**Example 1:** Extracting unique even numbers

```
a = [1, 2, 2, 3, 4, 4, 5, 6, 6, 7]
```

```
res = {num for num in a if num % 2 == 0}  
print(res)
```

**Output**

```
{2, 4, 6}
```

**Explanation:** This creates a set of even numbers from **a**, automatically removing duplicates.

**Example 2:** Creating a set of squared values

```
res = {num**2 for num in range(1, 6)}  
print(res)
```

**Output**

```
{1, 4, 9, 16, 25}
```

**Explanation:** This generates a set of squares, ensuring each value appears only once.

### 3. Generator comprehensions

[Generator](#) Comprehensions create iterators that generate values lazily, making them memory-efficient as elements are computed only when accessed.

**Syntax:**

```
(expression for item in iterable if condition)
```

**Where:**

- **expression:** Operation applied to each item.
- **iterable:** The source collection.
- **condition (optional):** Filters elements before including them.

**Example 1:** Generating even numbers using a generator

```
res = (num for num in range(10) if num % 2 == 0)  
print(list(res))
```

**Output**

```
[0, 2, 4, 6, 8]
```

**Explanation:** This generator produces even numbers from 0 to 9, but values are only computed when accessed.

## Example 2: Generating squares using a generator

```
res = (num**2 for num in range(1, 6))  
print(tuple(res))
```

### Output

```
(1, 4, 9, 16, 25)
```

**Explanation:** The generator creates squared values on demand and returns them as a tuple when converted.

## Python Exception Handling

Python Exception Handling handles errors that occur during the execution of a program. Exception handling allows to respond to the error, instead of crashing the running program. It enables you to catch and manage errors, making your code more robust and user-friendly. Let's look at an example:

### Handling a Simple Exception in Python

Exception handling helps in preventing crashes due to errors. Here's a basic example demonstrating how to catch an exception and handle it gracefully:

```
# Simple Exception Handling Example  
n = 10  
try:  
    res = n / 0 # This will raise a ZeroDivisionError  
  
except ZeroDivisionError:  
    print("Can't be divided by zero!")
```

### Output

```
Can't be divided by zero!
```

**Explanation:** In this example, dividing number by 0 raises a **ZeroDivisionError**. The try block contains the code that might cause an exception and the except block handles the exception, printing an error message instead of stopping the program.

## Difference Between Exception and Error

- **Error:** Errors are serious issues that a program should not try to handle. They are usually problems in the code's logic or configuration and need to be fixed by the programmer. Examples include syntax errors and memory errors.
- **Exception:** Exceptions are less severe than errors and can be handled by the program. They occur due to situations like invalid input, missing files or network issues.

**Example:**

### # Syntax Error (Error)

```
print("Hello world" # Missing closing parenthesis
```

### # ZeroDivisionError (Exception)

```
n = 10  
res = n / 0
```

**Explanation:** A syntax error is a coding mistake that prevents the code from running. In contrast, an exception like ZeroDivisionError can be managed during the program's execution using exception handling.

## Syntax and Usage

Exception handling in Python is done using the try, except, else and finally blocks.

```
try:  
#           Code           that           might           raise           an           exception  
except  
#           Code           to           handle           the           SomeException:  
#           Code           to           run           if           no           exception           occurs  
else:  
#           Code           to           run           if           no           exception           occurs  
finally:  
# Code to run regardless of whether an exception occurs
```

## try, except, else and finally Blocks

- **try Block:** [try block](#) lets us test a block of code for errors. Python will "try" to execute the code in this block. If an exception occurs, execution will immediately jump to the except block.
- **except Block:** [except block](#) enables us to handle the error or exception. If the code inside the try block throws an error, Python jumps to the except block and executes it. We can handle specific exceptions or use a general except to catch all exceptions.
- **else Block:** [else block](#) is optional and if included, must follow all except blocks. The else block runs only if no exceptions are raised in the try block. This is useful for code that should execute if the try block succeeds.
- **finally Block:** [finally block](#) always runs, regardless of whether an exception occurred or not. It is typically used for cleanup operations (closing files, releasing resources).

### Example:

```
try:  
    n = 0  
    res = 100 / n  
  
except ZeroDivisionError:  
    print("You can't divide by zero!")  
  
except ValueError:  
    print("Enter a valid number!")  
  
else:
```

```
print("Result is", res)
```

finally:

```
print("Execution complete.")
```

### Output

```
You can't divide by zero!  
Execution complete.
```

### Explanation:

- **try block** asks for user input and tries to divide 100 by the input number.
- **except blocks** handle `ZeroDivisionError` and `ValueError`.
- **else block** runs if no exception occurs, displaying the result.
- **finally block** runs regardless of the outcome, indicating the completion of execution.

Please refer [Python Built-in Exceptions](#) for some common exceptions.

## Python Catching Exceptions

When working with exceptions in Python, we can handle errors more efficiently by specifying the types of exceptions we expect. This can make code both safer and easier to debug.

### Catching Specific Exceptions

Catching specific exceptions makes code to respond to different exception types differently.

### Example:

try:

```
x = int("str") # This will cause ValueError
```

```
#inverse
```

```
inv = 1 / x
```

except `ValueError`:

```
print("Not Valid!")
```

except `ZeroDivisionError`:

```
print("Zero has no inverse!")
```

### Output

```
Not Valid!
```

### Explanation:

- The `ValueError` is caught because the string "str" cannot be converted to an integer.
- If x were 0 and conversion successful, the `ZeroDivisionError` would be caught when attempting to calculate its inverse.

## Catching Multiple Exceptions

We can catch multiple exceptions in a single block if we need to handle them in the same way or we can separate them if different types of exceptions require different handling.

### Example:

```
a = ["10", "twenty", 30] # Mixed list of integers and strings
try:
    total = int(a[0]) + int(a[1]) # 'twenty' cannot be converted to int

except (ValueError, TypeError) as e:
    print("Error", e)

except IndexError:
    print("Index out of range.")
```

### Output

```
Error invalid literal for int() with base 10: 'twenty'
```

### Explanation:

- The ValueError is caught when trying to convert "twenty" to an integer.
- TypeError might occur if the operation was incorrectly applied to non-integer types, but it's not triggered in this specific setup.
- IndexError would be caught if an index outside the range of the list was accessed, but in this scenario, it's under control.

### Catch-All Handlers and Their Risks

Here's a simple calculation that may fail due to various reasons.

### Example:

```
try:
    # Simulate risky calculation: incorrect type operation
    res = "100" / 20

except ArithmeticError:
    print("Arithmetic problem.")

except:
    print("Something went wrong!")
```

### Output

```
Something went wrong!
```

### Explanation:

- An ArithmeticError (more specific like ZeroDivisionError) might be caught if this were a number-to-number division error. However, TypeError is actually triggered here due to attempting to divide a string by a number.

- **catch-all except:** is used to catch the `TypeError`, demonstrating the risk that the programmer might not realize the actual cause of the error (type mismatch) without more detailed error logging.

## Raise an Exception

We [raise](#) an exception in Python using the `raise` keyword followed by an instance of the exception class that we want to trigger. We can choose from built-in exceptions or define our own custom exceptions by inheriting from Python's built-in `Exception` class.

### Basic Syntax:

```
raise ExceptionType("Error message")
```

### Example:

```
def set(age):  
    if age < 0:  
        raise ValueError("Age cannot be negative.")  
    print(f"Age set to {age}")  
  
try:  
    set(-5)  
except ValueError as e:  
    print(e)
```

### Output

```
Age cannot be negative.
```

### Explanation:

- The function `set` checks if the age is negative. If so, it raises a `ValueError` with a message explaining the issue.
- This ensures that the age attribute cannot be set to an invalid state, thus maintaining the integrity of the data.

### Advantages of Exception Handling:

- **Improved program reliability:** By handling exceptions properly, you can prevent your program from crashing or producing incorrect results due to unexpected errors or input.
- **Simplified error handling:** Exception handling allows you to separate error handling code from the main program logic, making it easier to read and maintain your code.
- **Cleaner code:** With exception handling, you can avoid using complex conditional statements to check for errors, leading to cleaner and more readable code.
- **Easier debugging:** When an exception is raised, the Python interpreter prints a traceback that shows the exact location where the exception occurred, making it easier to debug your code.

### Disadvantages of Exception Handling:

- **Performance overhead:** Exception handling can be slower than using conditional statements to check for errors, as the interpreter has to perform additional work to catch and handle the exception.
- **Increased code complexity:** Exception handling can make your code more complex, especially if you have to handle multiple types of exceptions or implement complex error handling logic.
- **Possible security risks:** Improperly handled exceptions can potentially reveal sensitive information or create security vulnerabilities in your code, so it's important to handle exceptions carefully and avoid exposing too much information about your program.

## Common Exceptions in Python

Python has many built-in exceptions, each representing a specific error condition. Please refer to [Python Built-in Exceptions](#) article for more detail.

## File Handling in Python

File handling refers to the process of performing operations on a file, such as creating, opening, reading, writing and closing it through a programming interface. It involves managing the data flow between the program and the file system on the storage device, ensuring that data is handled safely and efficiently.

### Why do we need File Handling

- To store data permanently, even after the program ends.
- To access external files like .txt, .csv, .json, etc.
- To process large files efficiently without using much memory.
- To automate tasks like reading configs or saving outputs.
- To handle input/output in real-world applications and tools.

### Opening a File in Python

To open a file, we can use `open()` function, which requires file-path and mode as arguments:

#### Syntax:

```
file = open('filename.txt', 'mode')
```

- **filename.txt:** name (or path) of the file to be opened.
- **mode:** mode in which you want to open the file (read, write, append, etc.).

**Note:** If you don't specify the mode, Python uses 'r' (read mode) by default.

### Basic Example: Opening a File

```
f = open("geek.txt", "r")  
print(f)
```

This code opens the file **demo.txt** in read mode. If the file exists, it connects successfully, otherwise, it throws an error.

### Closing a File

It's important to close the file after you're done using it. `file.close()` method closes the file and releases the system resources ensuring that changes are saved properly (in case of writing)

```
file = open("tech.txt", "r")
# Perform file operations
file.close()
```

## Checking File Properties

Once the file is open, we can check some of its properties:

```
f = open("tech.txt", "r")

print("Filename:", f.name)
print("Mode:", f.mode)
print("Is Closed?", f.closed)

f.close()
print("Is Closed?", f.closed)
```

**Output:**

Filename:		demo.txt
Mode:		r
Is	Closed?	False
Is Closed? True		

**Explanation:**

- **f.name:** Returns the name of the file that was opened (in this case, "demo.txt").
- **f.mode:** Tells us the mode in which the file was opened. Here, it's 'r' which means read mode.
- **f.closed:** Returns a boolean value- False when file is currently open otherwise True.

## Reading a File

Reading a file can be achieved by `file.read()` which reads the entire content of the file. After reading the file we can close the file using `file.close()` which closes the file after reading it, which is necessary to free up system resources.

**Example:** Reading a File in Read Mode (r)

```
file = open("tech.txt", "r")
content = file.read()
print(content)
file.close()
Output:
```

Hello	world
Tech	
123 456	

## Using with Statement

Instead of manually opening and closing the file, you can use the [with statement](#), which automatically handles closing. This reduces the risk of file corruption and resource leakage.

**Example:** Let's assume we have a file named **tech.txt** that contains text **"Hello, World!"**.

```
with open("tech.txt", "r") as file:  
    content = file.read()  
    print(content)
```

**Output:**

```
Hello, World!
```

## Handling Exceptions When Closing a File

It's important to [handle exceptions](#) to ensure that files are closed properly, even if an error occurs during file operations.

```
try:  
    file = open("tech.txt", "r")  
    content = file.read()  
    print(content)  
finally:  
    file.close()
```

**Output:**

```
Hello, World!
```

**Explanation:**

- **try:** Starts the block to handle code that might raise an error.
- **open():** Opens the file in read mode.
- **read():** Reads the content of the file.
- **finally:** Ensures the code inside it runs no matter what.
- **close():** Safely closes the file to free resources.

## Memory Management in Python

Memory management refers to process of allocating and deallocating memory to a program while it runs. Python handles memory management automatically using mechanisms like reference counting and garbage collection, which means programmers do not have to manually manage memory.

Let's explore how Python automatically manages memory using garbage collection and reference counting.

### Garbage Collection

It is a process in which Python automatically frees memory occupied by objects that are no longer in use.

- If an object has no references pointing to it (i.e., nothing is using it), garbage collector removes it from memory.

- This ensures that unused memory can be reused for new objects.

For more information, refer to [Garbage Collection in Python](#)

## Reference Counting

It is one of the primary memory management techniques used in Python, where:

- Every object keeps a reference counter, which tells how many variables (or references) are currently pointing to that object.
- When a new reference to the object is created, counter increases.
- When a reference is deleted or goes out of scope, counter decreases.
- If the counter reaches zero, it means no variable is using the object anymore, so Python automatically deallocates (frees) that memory.

### Example:

```
a = [1, 2, 3]
```

```
b = a
```

```
print(id(a), id(b)) # Same ID → both point to same list
```

```
b.append(4)
```

```
print(a)
```

### Output

```
140645192555456 140645192555456
```

```
[1, 2, 3, 4]
```

### Explanation:

- **a** and **b** both refer to same list in memory.
- Changing **b** also changes **a**, because both share same reference.

Now that we know how references work, let's see how Python organizes memory.

## Memory Allocation in Python

It is the process of reserving space in a computer's memory so that a program can store its data and variables while it runs. In Python, this process is handled automatically by interpreter, but the way objects are stored and reused can make a big difference in performance.

Let's see an example to understand it better.

### Example: Memory Optimization with Small Integers

Python applies an internal optimization called **object interning** for small immutable objects (like integers from -5 to 256 and some strings). Instead of creating a new object every time, Python reuses same object to save memory.

Suppose:

```
x = 10
y = 10
```

Here, Python does not create two separate objects for 10. Instead, both **x** and **y** point to the same memory location. Let's verify if it's true:

```
x = 10
y = x
```

```
if id(x) == id(y):
    print("x and y refer to same object")
```

### Output

```
x and y refer to same object
```

Now, if we change **x** to a different integer:

```
x = 10
y = x
x += 1
```

```
if id(x) != id(y):
    print("x and y do not refer to the same object")
```

### Output

```
x and y do not refer to the same object
```

When **x** is changed, Python creates a new object (11) for it. The old link with **10** breaks, but **y** still refers to **10**.

In Python, memory is divided mainly into two parts:

- Stack Memory
- Heap Memory

Both play different roles in how variables and objects are stored and accessed.

### Stack Memory

Stack memory is where method/function calls and reference variables are stored.

- Whenever a function is called, Python adds it to the call stack.
- Inside this function, all variables declared (like numbers, strings or temporary references) are stored in stack memory.
- Once the function finishes executing, stack memory used by it is automatically freed.

**In simple terms:** Stack memory is temporary and is only alive until the function or method call is running.

### How it Works:

- Allocation happens in a contiguous (continuous) block of memory.
- Python's compiler handles this automatically, so developers don't need to manage it.
- It is fast, but it is limited in size and scope (only works within a function call).

#### Example:

```
def func():
```

```
    # These variables are created in stack memory
```

```
    a = 20
```

```
    b = []
```

```
    c = ""
```

Here **a**, **b** and **c** are stored in stack memory when function **func()** is called. As soon as function ends, this memory is released automatically.

## Heap Memory

Heap memory is where actual objects and values are stored.

- When a variable is created, Python allocates its object/value in heap memory.
- Stack memory stores only the reference (pointer) to this object.
- Objects in heap memory can be shared among multiple functions or exist even after a function has finished executing.

**In simple terms:** Heap memory is like a storage area where all values/objects live and stack memory just keeps directions (references) to them.

#### How it Works:

- Heap memory allocation happens at runtime.
- Unlike stack, it does not follow a strict order it's more flexible.
- This is where large data structures (lists, dictionaries, objects) are stored.
- Garbage collection is responsible for cleaning up unused objects from heap memory.

#### Example:

```
# This list of 10 integers is allocated in heap memory
```

```
a = [0] * 10
```

## Python RegEx

A Regular Expression or RegEx is a special sequence of characters that uses a search pattern to find a string or set of strings.

It can detect the presence or absence of a text by matching it with a particular pattern and also can split a pattern into one or more sub-patterns.

## Regex Module in Python

Python has a built-in module named "**re**" that is used for regular expressions in Python. We can import this module by using [import statement](#).

Importing re module in Python using following command:

```
import re
```

## How to Use RegEx in Python?

You can use RegEx in Python after importing re module.

### Example:

This Python code uses regular expressions to search for the word "**portal**" in the given string and then prints the start and end indices of the matched word within the string.

```
import re
```

```
s = 'Tech: A computer science portal for tech'  
match = re.search(r'portal', s)
```

```
print('Start Index:', match.start())  
print('End Index:', match.end())
```

### Output

```
Start Index: 34
```

```
End Index: 40
```

**Note:** Here *r* character (*r'portal'*) stands for raw, not regex. The raw string is slightly different from a regular string, it won't interpret the *\* character as an escape character. This is because the regular expression engine uses *\* character for its own escaping purpose.

Before starting with the Python regex module let's see how to actually write regex using metacharacters or special sequences.

## RegEx Functions

The re module in Python provides various functions that help search, match, and manipulate strings using regular expressions.

Below are main functions available in the re module:

Function	Description
re.findall()	finds and returns all matching occurrences in a list
re.compile()	Regular expressions are compiled into pattern objects
re.split()	Split string by the occurrences of a character or a pattern.
re.sub()	Replaces all occurrences of a character or patter with a replacement string.

Function	Description
resubn	It's similar to re.sub() method but it returns a tuple: (new_string, number_of_substitutions)
re.escape()	Escapes special character
re.search()	Searches for first occurrence of character or pattern

Let's see the working of these RegEx functions with definition and examples:

## 1. re.findall()

Returns all non-overlapping matches of a pattern in the string as a list. It scans the string from left to right.

**Example:** This code uses regular expression `\d+` to find all sequences of one or more digits in the given string.

```
import re
string = """Hello my Number is 123456789 and
          my friend's number is 987654321"""
```

```
regex = '\d+'
match = re.findall(regex, string)
print(match)
```

### Output

```
['123456789', '987654321']
```

## 2. re.compile()

Compiles a regex into a pattern object, which can be reused for matching or substitutions.

**Example 1:** This pattern `[a-e]` matches all lowercase letters between 'a' and 'e', in the input string **"Aye, said Mr. Gibenson Stark"**. The output should be `['e', 'a', 'd', 'b', 'e']`, which are matching characters.

```
import re
p = re.compile('[a-e]')
print(p.findall("Aye, said Mr. Gibenson Stark"))
```

### Output

```
['e', 'a', 'd', 'b', 'e', 'a']
```

### Explanation:

- First occurrence is 'e' in "Aye" and not 'A', as it is Case Sensitive.
- Next Occurrence is 'a' in "said", then 'd' in "said", followed by 'b' and 'e' in "Gibenson", the Last 'a' matches with "Stark".

- Metacharacter backslash '\' has a very important role as it signals various sequences. If the backslash is to be used without its special meaning as metacharacter, use '\\'.

**Example 2:** The code uses regular expressions to find and list all single digits and sequences of digits in the given input strings. It finds single digits with `\d` and sequences of digits with `\d+`.

```
import re
p = re.compile('\d')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))

p = re.compile('\d+')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))
```

### Output

```
['1', '1', '4', '1', '8', '8', '6']
['11', '4', '1886']
```

**Example 3:** Word and non-word characters

- `\w` matches a single word character.
- `\w+` matches a group of word characters.
- `\W` matches non-word characters.

```
import re

p = re.compile('\w')
print(p.findall("He said * in some_lang."))

p = re.compile('\w+')
print(p.findall("I went to him at 11 A.M., he \
said *** in some_language."))

p = re.compile('\W')
print(p.findall("he said *** in some_language."))
```

### Output

```
['H', 'e', 's', 'a', 'i', 'd', 'i', 'n', 's', 'o', 'm', 'e', '_', 'l', 'a', 'n', 'g']
['I', 'went', 'to', 'him', 'at', '11', 'A', 'M', 'he', 'said', 'in', 'some_language']
[' ', ' ', ' ', '*', '*', '*', ' ', ' ', '.']
```

**Example 4:** The regular expression pattern `'ab*'` to find and list all occurrences of `'ab'` followed by zero or more `'b'` characters. In the input string `"ababbaabbb"`. It returns the following list of matches: `['ab', 'abb', 'abbb']`.

```
import re
p = re.compile('ab*')
print(p.findall("ababbaabbb"))
```

## Output

```
['ab', 'abb', 'a', 'abbb']
```

## Explanation:

- Output 'ab', is valid because of single 'a' accompanied by single 'b'.
- Output 'abb', is valid because of single 'a' accompanied by 2 'b'.
- Output 'a', is valid because of single 'a' accompanied by 0 'b'.
- Output 'abbb', is valid because of single 'a' accompanied by 3 'b'.

## 3. re.split()

Splits a string wherever the pattern matches. The remaining characters are returned as list elements.

## Syntax:

```
re.split(pattern, string, maxsplit=0, flags=0)
```

- **pattern:** Regular expression to match split points.
- **string:** The input string to split.
- **maxsplit (optional):** Limits the number of splits. Default is 0 (no limit).
- **flags (optional):** Apply regex flags like re.IGNORECASE.

## Example 1: Splitting by non-word characters or digits

This example demonstrates how to split a string using different patterns like non-word characters (`\W+`), apostrophes, and digits (`\d+`).

```
from re import split
```

```
print(split('\W+', 'Words, words , Words'))
print(split('\W+', "Word's words Words"))
print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))
print(split('\d+', 'On 12th Jan 2016, at 11:02 AM'))
```

## Output

```
['Words', 'words', 'Words']
['Word', 's', 'words', 'Words']
['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']
['On ', 'th Jan ', ', at ', ':', ' AM']
```

## Example 2: Using maxsplit and flags

This example shows how to limit the number of splits using maxsplit, and how flags can control case sensitivity.

```
import re
print(re.split('\d+', 'On 12th Jan 2016, at 11:02 AM', 1))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here', flags=re.IGNORECASE))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here'))
```

## Output

```
['On ', 'th Jan 2016, at 11:02 AM']  
['', 'y, ', 'oy oh ', 'oy, ', 'om', ' h', 'r', '']  
['A', 'y, Boy oh ', 'oy, ', 'om', ' h', 'r', '']
```

**Note:** In the second and third cases of the above, `[a-f]+` splits the string using any combination of lowercase letters from 'a' to 'f'. The `re.IGNORECASE` flag includes uppercase letters in the match.

## 4. re.sub()

The `re.sub()` function replaces all occurrences of a pattern in a string with a replacement string.

### Syntax:

```
re.sub(pattern, repl, string, count=0, flags=0)
```

- **pattern:** The regex pattern to search for.
- **repl:** The string to replace matches with.
- **string:** The input string to process.
- **count (optional):** Maximum number of substitutions (default is 0, which means replace all).
- **flags (optional):** Regex flags like `re.IGNORECASE`.

**Example 1:** The following examples show different ways to replace the pattern `'ub'` with `'~*'`, using various flags and count values.

```
import re
```

```
# Case-insensitive replacement of all 'ub'
```

```
print(re.sub('ub', '~*', 'Subject has Uber booked already', flags=re.IGNORECASE))
```

```
# Case-sensitive replacement of all 'ub'
```

```
print(re.sub('ub', '~*', 'Subject has Uber booked already'))
```

```
# Replace only the first 'ub', case-insensitive
```

```
print(re.sub('ub', '~*', 'Subject has Uber booked already', count=1, flags=re.IGNORECASE))
```

```
# Replace "AND" with "&", ignoring case
```

```
print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE))
```

## Output

```
S~*ject has ~*er booked already  
S~*ject has Uber booked already  
S~*ject has Uber booked already  
Baked Beans & Spam
```

## 5. re.subn()

`re.subn()` function works just like **`re.sub()`**, but instead of returning only the modified string, it returns a tuple: **(new\_string, number\_of\_substitutions)**

### Syntax:

```
re.subn(pattern, repl, string, count=0, flags=0)
```

### Example: Substitution with count

This example shows how `re.subn()` gives both the replaced string and the number of times replacements were made.

```
import re
```

```
# Case-sensitive replacement
```

```
print(re.subn('ub', '~*', 'Subject has Uber booked already'))
```

```
# Case-insensitive replacement
```

```
t = re.subn('ub', '~*', 'Subject has Uber booked already', flags=re.IGNORECASE)
```

```
print(t)
```

```
print(len(t))    # tuple length
```

```
print(t[0])      # modified string
```

### Output

```
('S~*ject has Uber booked already', 1)
('S~*ject has ~*er booked already', 2)
2
S~*ject has ~*er booked already
```

## 6. re.escape()

`re.escape()` function adds a backslash (`\`) before all special characters in a string. This is useful when you want to match a string literally, including any characters that have special meaning in regex (like `.`, `*`, `[`, `]`, etc.).

### Syntax:

```
re.escape(string)
```

### Example: Escaping special characters

This example shows how `re.escape()` treats spaces, brackets, dashes, and tabs as literal characters.

```
import re
```

```
print(re.escape("This is Awesome even 1 AM"))
```

```
print(re.escape("I Asked what is this [a-9], he said \t ^WoW"))
```

### Output

```
This\ is\ Awesome\ even\ 1\ AM
I\ Asked\ what\ is\ this\ \[, \ he\ said\ \ \ \ \ ^WoW
```

## 7. re.search()

The `re.search()` function searches for the first occurrence of a pattern in a string. It returns a **match object** if found, otherwise **None**.

**Note:** Use it when you want to check if a pattern exists or extract the first match.

### Example: Search and extract values

This example searches for a date pattern with a month name (letters) followed by a day (digits) in a sentence.

```
import re

regex = r"([a-zA-Z]+) (\d+)"
match = re.search(regex, "I was born on June 24")

if match:
    print("Match at index %s, %s" % (match.start(), match.end()))
    print("Full match:", match.group(0))
    print("Month:", match.group(1))
    print("Day:", match.group(2))
else:
    print("The regex pattern does not match.")
```

### Output

```
Match at index 14, 21
Full match: June 24
Month: June
Day: 24
```

## Meta-characters

Metacharacters are special characters in regular expressions used to define search patterns. The `re` module in Python supports several metacharacters that help you perform powerful pattern matching.

Below is a quick reference table:

MetaCharacters	Description
\	Used to drop the special meaning of character following it
[]	Represent a character class
^	Matches the beginning
\$	Matches the end

MetaCharacters	Description
.	Matches any character except newline
	Means OR (Matches with any of the characters separated by it.
?	Matches zero or one occurrence
*	Any number of occurrences (including 0 occurrences)
+	One or more occurrences
{ }	Indicate the number of occurrences of a preceding regex to match.
()	Enclose a group of Regex

Let's discuss each of these metacharacters in detail:

## 1. \ - Backslash

The backslash (\) makes sure that the character is not treated in a special way. This can be considered a way of escaping metacharacters.

For example, if you want to search for the dot(.) in the string then you will find that dot(.) will be treated as a special character as is one of the metacharacters (as shown in the above table). So for this case, we will use the backslash(\) just before the dot(.) so that it will lose its specialty. See the below example for a better understanding.

**Example:** The first search (**re.search(r'.', s)**) matches any character, not just the period, while the second search (**re.search(r'\.', s)**) specifically looks for and matches the period character.

```
import re
```

```
s = 'tech.fortech'
```

```
# without using \
```

```
match = re.search(r'.', s)
```

```
print(match)
```

```
# using \
```

```
match = re.search(r'\.', s)
```

```
print(match)
```

## Output

```
<re.Match object; span=(0, 1), match='g'>
```

```
<re.Match object; span=(5, 6), match='.'>
```

## 2. [] - Square Brackets

Square Brackets ([]) represent a character class consisting of a set of characters that we wish to match. For example, the character class [abc] will match any single a, b, or c.

We can also specify a range of characters using - inside the square brackets. For example,

- [0, 3] is same as [0123]
- [a-c] is same as [abc]

We can also invert the character class using the caret(^) symbol. For example,

- [^0-3] means any character except 0, 1, 2, or 3
- [^a-c] means any character except a, b, or c

**Example:** In this code, you're using regular expressions to find all the characters in the string that fall within the range of 'a' to 'm'. The **re.findall()** function returns a list of all such characters. In the given string, the characters that match this pattern are: 'c', 'k', 'b', 'f', 'j', 'e', 'h', 'l', 'd', 'g'.

```
import re
```

```
string = "The quick brown fox jumps over the lazy dog"
```

```
pattern = "[a-m]"
```

```
result = re.findall(pattern, string)
```

```
print(result)
```

### Output

```
['h', 'e', 'i', 'c', 'k', 'b', 'f', 'j', 'm', 'e', 'h', 'e', 'l', 'a',  
'd', 'g']
```

## 3. ^ - Caret

Caret (^) symbol matches the beginning of the string i.e. checks whether the string starts with the given character(s) or not. For example -

- ^g will check if the string starts with g such as tech, globe, girl, g, etc.
- ^ge will check if the string starts with ge such as tech, tech, etc.

**Example:** This code uses regular expressions to check if a list of strings starts with "The". If a string begins with "The," it's marked as "Matched" otherwise, it's labeled as "Not matched".

```
import re
```

```
regex = r'^The'
```

```
strings = ['The quick brown fox', 'The lazy dog', 'A quick brown fox']
```

```
for string in strings:
```

```
    if re.match(regex, string):
```

```
        print(f'Matched: {string}')
```

```
    else:
```

```
        print(f'Not matched: {string}')
```

## Output

```
Matched: The quick brown fox
Matched: The lazy dog
Not matched: A quick brown fox
```

## 4. \$ - Dollar

Dollar(\$) symbol matches the end of the string i.e checks whether the string ends with the given character(s) or not. For example-

- s\$ will check for the string that ends with a such as tech, ends, s, etc.
- ks\$ will check for the string that ends with ks such as tech, etc.

**Example:** This code uses a regular expression to check if the string ends with **"World!"**. If a match is found, it prints **"Match found!"** otherwise, it prints **"Match not found"**.

```
import re
```

```
string = "Hello World!"
pattern = r"World!"
```

```
match = re.search(pattern, string)
if match:
    print("Match found!")
else:
    print("Match not found.")
```

## Output

```
Match found!
```

## 5. . - Dot

Dot(.) symbol matches only a single character except for the newline character (\n). For example -

- a.b will check for the string that contains any character at the place of the dot such as acb, acbd, abbb, etc
- .. will check if the string contains at least 2 characters

**Example:** This code uses a regular expression to search for the pattern **"brown.fox"** within the string. The dot (.) in the pattern represents any character. If a match is found, it prints **"Match found!"** otherwise, it prints **"Match not found"**.

```
import re
```

```
string = "The quick brown fox jumps over the lazy dog."
pattern = r"brown.fox"
```

```
match = re.search(pattern, string)
if match:
```

```
print("Match found!")
else:
    print("Match not found.")
```

## Output

```
Match found!
```

## 6. | - Or

The | operator means either pattern on its left or right can match. a|b will match any string that contains a or b such as acd, bcd, abcd, etc.

## 7. ? - Question Mark

The question mark (?) indicates that the preceding element should be matched zero or one time. It allows you to specify that the element is optional, meaning it may occur once or not at all.

**For example**, ab?c will be matched for the string ac, acb, dabc but will not be matched for abbc because there are two b. Similarly, it will not be matched for abdc because b is not followed by c.

## 8. \* - Star

Star (\*) symbol matches zero or more occurrences of the regex preceding the \* symbol.

**For example**, ab\*c will be matched for the string ac, abc, abbbc, dabc, etc. but will not be matched for abdc because b is not followed by c.

## 9. + - Plus

Plus (+) symbol matches one or more occurrences of the regex preceding the + symbol.

**For example**, ab+c will be matched for the string abc, abbc, dabc, but will not be matched for ac, abdc, because there is no b in ac and b, is not followed by c in abdc.

## 10. {m, n} - Braces

Braces match any repetitions preceding regex from m to n both inclusive.

**For example**, a{2, 4} will be matched for the string aaab, baaaac, gaad, but will not be matched for strings like abc, bc because there is only one a or no a in both the cases.

## 11. (<regex>) - Group

Group symbol is used to group sub-patterns.

**For example**, (a|b)cd will match for strings like acd, abcd, gacd, etc.

## Special Sequences

Special sequences do not match for the actual character in the string instead it tells the specific location in the search string where the match must occur. It makes it easier to write commonly used patterns.

### List of special sequences

Special Sequence	Description	Examples	
\A	Matches if the string begins with the given character	\Afor	for tech
			for the world
\b	Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word.	\bge	techs
			get
\B	It is the opposite of the \b i.e. the string should not start or end with the given regex.	\Bge	together
			forge
\d	Matches any decimal digit, this is equivalent to the set class [0-9]	\d	123
			gee1
\D	Matches any non-digit character, this is equivalent to the set class [^0-9]	\D	tech
			geek1

Special Sequence	Description	Examples	
\s	Matches any whitespace character.	\s	TEC ks
			a bc a
\S	Matches any non-whitespace character	\S	a bd
			abcd
\w	Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].	\w	123
			tech4
\W	Matches any non-alphanumeric character.	\W	>\$
			gee<>
\Z	Matches if the string ends with the given regex	ab\Z	abcdab
			abababab

## Sets for character matching

A **Set** is a set of characters enclosed in '[]' brackets. Sets are used to match a single character in the set of characters specified between brackets. Below is the list of Sets:

Set	Description
\{n,\}	Quantifies the preceding character or group and matches at least n occurrences.
*	Quantifies the preceding character or group and matches zero or more occurrences.
[0123]	Matches the specified digits (0, 1, 2, or 3)
[^arn]	matches for any character EXCEPT a, r, and n
\d	Matches any digit (0-9).
[0-5][0-9]	matches for any two-digit numbers from 00 and 59

Set	Description
\w	Matches any alphanumeric character (a-z, A-Z, 0-9, or _).
[a-n]	Matches any lower case alphabet between a and n.
\D	Matches any non-digit character.
[arn]	matches where one of the specified characters (a, r, or n) are present
[a-zA-Z]	matches any character between a and z, lower case OR upper case
[0-9]	matches any digit between 0 and 9

## Match Object

A Match object contains all the information about the search and the result and if there is no match found then None will be returned. Let's see some of the commonly used methods and attributes of the match object.

### 1. Getting the string and the regex

match.re attribute returns the regular expression passed and match.string attribute returns the string passed.

#### Example:

The code searches for the letter "G" at a word boundary in the string "Welcome to TECH" and prints the regular expression pattern (**res.re**) and the original string (**res.string**).

```
import re
s = "Welcome to TECH"
res = re.search(r"\bG", s)
```

```
print(res.re)
print(res.string)
```

#### Output

```
re.compile('\bG')
Welcome to TECH
```

### 2. Getting index of matched object

- **start()** method returns the starting index of the matched substring
- **end()** method returns the ending index of the matched substring
- **span()** method returns a tuple containing the starting and the ending index of the matched substring

#### Example: Getting index of matched object

The code searches for substring "Gee" at a word boundary in string "Welcome to TECH" and prints start index of the match (**res.start()**), end index of the match (**res.end()**) and span of the match (**res.span()**).

```
import re
```

```
s = "Welcome to TECH"
res = re.search(r"\bGee", s)
```

```
print(res.start())
print(res.end())
print(res.span())
```

#### Output

```
11
14
(11, 14)
```

### 3. Getting matched substring

group() method returns the part of the string for which the patterns match. See the below example for a better understanding.

#### Example: Getting matched substring

The code searches for a sequence of two non-digit characters followed by a space and the letter 't' in the string "Welcome TECH" and prints the matched text using **res.group()**.

```
import re
s = "Welcome to TECH"
res = re.search(r"\D{2} t", s)
print(res.group())
```

#### Output

```
me t
```

In the above example, our pattern specifies for the string that contains at least 2 characters which are followed by a space, and that space is followed by a t.

### Basic RegEx Patterns

Let's understand some of the basic regular expressions. They are as follows:

#### 1. Character Classes

Character classes allow matching any one character from a specified set. They are

## 2. Ranges

In RegEx, a range allows matching characters or digits within a span using - inside []. For example, [0-9] matches digits, [A-Z] matches uppercase letters.

```
import re
print('Range',re.search(r'[a-zA-Z]', 'x'))
```

### Output

```
Range <re.Match object; span=(0, 1), match='x'>
```

## 3. Negation

Negation in a character class is specified by placing a ^ at the beginning of the brackets, meaning match anything except those characters.

### Syntax:

```
[^a-z]
```

### Example:

```
import re

print(re.search(r'[^a-z]', 'c'))
print(re.search(r'G[^e]', 'GETS'))
```

### Output

```
None
```

```
None
```

## 3. Shortcuts

Shortcuts are shorthand representations for common character classes. Let's discuss some of the shortcuts provided by the regular expression engine.

- \w - matches a word character
- \d - matches digit character
- \s - matches whitespace character (space, tab, newline, etc.)
- \b - matches a zero-length character

## 4. Beginning and End of String

The ^ character chooses the beginning of a string and the \$ character chooses the end of a string.

```
import re
```

### # Beginning of String

```
match = re.search(r'^TECH', 'Campus TECH of the month')
print('Beg. of String:', match)
```

```
match = re.search(r'^TECH', 'TECHof the month')
print('Beg. of String:', match)
```

### # End of String

```
match = re.search(r'TECHs$', 'Compute science portal-Tech')
print('End of String:', match)
```

### Output

```
Beg. of String: None
Beg. of String: <_sre.SRE_Match object; span=(0, 4), match='TECH'>
End of String: <_sre.SRE_Match object; span=(31, 36), match='Tech'>
```

## 5. Any Character

The . character represents any single character outside a bracketed character class.

```
import re
print('Any Character', re.search(r'p.th.n', 'python 3'))
```

### Output

```
Any Character <_sre.SRE_Match object; span=(0, 6), match='python'>
```

## 6. Optional Characters

Regular expression engine allows you to specify optional characters using the ? character. It allows a character or character class either to present once or else not to occur. Let's consider the example of a word with an alternative spelling - color or colour.

```
import re

print('Color', re.search(r'colou?r', 'color'))
print('Colour', re.search(r'colou?r', 'colour'))
```

### Output

```
Color <_sre.SRE_Match object; span=(0, 5), match='color'>
Colour <_sre.SRE_Match object; span=(0, 6), match='colour'>
```

## 7. Repetition

Repetition enables you to repeat the same character or character class. Consider an example of a date that consists of day, month, and year. Let's use a regular expression to identify the date (mm-dd-yyyy).

```
import re
print('Date{mm-dd-yyyy}:', re.search(r'[\d]{2}-[\d]{2}-[\d]{4}', '18-08-2020'))
```

## Output

```
Date{mm-dd-yyyy}: <_sre.SRE_Match object; span=(0, 10), match='18-08-2020'>
```

Here, the regular expression engine checks for two consecutive digits. Upon finding the match, it moves to the hyphen character. After then, it checks the next two consecutive digits and the process is repeated.

Let's discuss three other regular expressions under repetition.

### 7.1 Repetition ranges

The repetition range is useful when you have to accept one or more formats. Consider a scenario where both three digits, as well as four digits, are accepted. Let's have a look at the regular expression.

```
import re
```

```
print('Three Digit:', re.search(r'[\d]{3,4}', '189'))  
print('Four Digit:', re.search(r'[\d]{3,4}', '2145'))
```

## Output

```
Three Digit: <_sre.SRE_Match object; span=(0, 3), match='189'>  
Four Digit: <_sre.SRE_Match object; span=(0, 4), match='2145'>
```

### 7.2 Open-Ended Ranges

There are scenarios where there is no limit for a character repetition. In such scenarios, you can set the upper limit as infinitive. A common example is matching street addresses. Let's have a look

```
import re
```

```
print(re.search(r'[\d]{1,}', '5th Floor, A-118,\nSector-136, Noida, Uttar Pradesh - 201305'))
```

## Output

```
<_sre.SRE_Match object; span=(0, 1), match='5'>
```

### 7.3 Shorthand

Shorthand characters allow you to use + character to specify one or more ({1,}) and \* character to specify zero or more ({0,}).

```
import re
```

```
print(re.search(r'[\d]+', '5th Floor, A-118,\nSector-136, Noida, Uttar Pradesh - 201305'))
```

## Output

```
<_sre.SRE_Match object; span=(0, 1), match='5'>
```

## 8. Grouping

Grouping is the process of separating an expression into groups by using parentheses, and it allows you to fetch each individual matching group.

```
import re
grp = re.search(r'([\d]{2})-([\d]{2})-([\d]{4})', '26-08-2020')
print(grp)
```

### Output

```
<_sre.SRE_Match object; span=(0, 10), match='26-08-2020'>
```

Let's see some of its functionality.

### 8.1 Return the entire match

The re module allows you to return the entire match using the *group()* method

```
import re
grp = re.search(r'([\d]{2})-([\d]{2})-([\d]{4})', '26-08-2020')
print(grp.group())
```

### Output

```
26-08-2020
```

### 8.2 Return a tuple of matched groups

You can use *groups()* method to return a tuple that holds individual matched groups

```
import re
grp = re.search(r'([\d]{2})-([\d]{2})-([\d]{4})', '26-08-2020')
print(grp.groups())
```

### Output

```
('26', '08', '2020')
```

### 8.3 Retrieve a single group

Upon passing the index to a group method, you can retrieve just a single group.

```
import re
grp = re.search(r'([\d]{2})-([\d]{2})-([\d]{4})', '26-08-2020')
print(grp.group(3))
```

### Output

```
2020
```

### 8.4 Name your groups

The re module allows you to name your groups. Let's look into the syntax.

```
import re
match = re.search(r'(?P<dd>[\d]{2})-(?P<mm>[\d]{2})-(?P<yyyy>[\d]{4})',
                  '26-08-2020')
print(match.group('mm'))
```

### Output

```
08
```

## 8.5 Individual match as a dictionary

We have seen how regular expression provides a tuple of individual groups. Not only tuple, but it can also provide individual match as a dictionary in which the name of each group acts as the dictionary key.

```
import re
match = re.search(r'(?P<dd>[\d]{2})-(?P<mm>[\d]{2})-(?P<yyyy>[\d]{4})',
                  '26-08-2020')
print(match.groupdict())
```

### Output

```
{'dd': '26', 'mm': '08', 'yyyy': '2020'}
```

## 9. Lookahead

In the case of a negated character class, it won't match if a character is not present to check against the negated character. We can overcome this case by using lookahead; it accepts or rejects a match based on the presence or absence of content.

```
import re
print('negation:', re.search(r'n[^e]', 'Python'))
print('lookahead:', re.search(r'n(?!e)', 'Python'))
```

### Output

```
negation: None
```

```
lookahead: <_sre.SRE_Match object; span=(5, 6), match='n'>
```

Lookahead can also disqualify the match if it is not followed by a particular character. This process is called a positive lookahead, and can be achieved by simply replacing ! character **with = character**.

```
import re
print('positive lookahead', re.search(r'n(?:=e)', 'jasmine'))
```

### Output

```
positive lookahead <_sre.SRE_Match object; span=(5, 6), match='n'>
```

## 10. Substitution

The regular expression can replace the string and returns the replaced one using the `re.sub` method. It is useful when you want to avoid characters such as `/`, `-`, `.`, etc. before storing it to a database. It takes three arguments:

- the regular expression
- the replacement string
- the source string being searched

Let's have a look at the below code that replaces `- character` from a credit card number.

```
import re
print(re.sub(r'([\d]{4})-([\d]{4})-([\d]{4})-([\d]{4})',r'\1\2\3\4',
            '1111-2222-3333-4444'))
```

### Output

```
1111222233334444
```

## Iterators in Python

An iterator in Python is an object used to traverse through all the elements of a collection (like lists, tuples, or dictionaries) one element at a time. It follows the iterator protocol, which involves two key methods:

- `__iter__()`: Returns the iterator object itself.
- `__next__()`: Returns the next value from the sequence. Raises `StopIteration` when the sequence ends.

### Why do we need iterators in Python

- **Lazy Evaluation** : Processes items only when needed, saving memory.
- **Generator Integration** : Pairs well with generators and functional tools.
- **Stateful Traversal** : Remembers position between calls.
- **Uniform Looping** : Works across data types with the same syntax.
- **Composable Logic** : Easily build complex pipelines using tools like `itertools`.

### Built-in Iterator Example

```
s = "TCH"
it = iter(s)
```

```
print(next(it))
print(next(it))
print(next(it))
```

### Output

```
T
C
H
```

## Creating an Custom Iterator

Creating a custom iterator in Python involves defining a class that implements the `__iter__()` and `__next__()` methods according to the Python iterator protocol.

### Steps to follow:

- **Define the Class:** Start by defining a class that will act as the iterator.
- **Initialize Attributes:** In the `__init__()` method of the class, initialize any required attributes that will be used throughout the iteration process.
- **Implement `__iter__()`:** This method should return the iterator object itself. This is usually as simple as returning `self`.
- **Implement `__next__()`:** This method should provide the next item in the sequence each time it's called.

Below is an example of a custom class called `EvenNumbers`, which iterates through even numbers starting from 2:

```
class EvenNumbers:
    def __iter__(self):
        self.n = 2 # Start from the first even number
        return self

    def __next__(self):
        x = self.n
        self.n += 2 # Increment by 2 to get the next even number
        return x

# Create an instance of EvenNumbers
even = EvenNumbers()
it = iter(even)

# Print the first five even numbers
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

### Output

```
2
4
6
8
10
```

### Explanation:

- **Initialization:** The `__iter__()` method initializes the iterator at 2, the first even number.
- **Iteration:** The `__next__()` method retrieves the current number and then increases it by 2, ensuring the next call returns the subsequent even number.
- **Usage:** We create an instance of `EvenNumbers`, turn it into an iterator and then use the `next()` function to fetch even numbers one at a time.

## StopIteration Exception

The `StopIteration` exception is integrated with Python's iterator protocol. It signals that the iterator has no more items to return. Once this exception is raised, further calls to `next()` on the same iterator will continue raising `StopIteration`.

### Example:

```
li = [100, 200, 300]
it = iter(li)
```

```
# Iterate until StopIteration is raised
while True:
    try:
        print(next(it))
    except StopIteration:
        print("End of iteration")
        break
```

### Output

```
100
200
300
End of iteration
```

In this example, the `StopIteration` exception is manually handled in the `while` loop, allowing for custom handling when the iterator is exhausted.

## Difference between Iterator and Iterable

Feature	Iterable	Iterator
Definition	Any object that can return an iterator	Object with a state for iteration
Key Method	Implements <code>__iter__()</code>	Implements both <code>__iter__()</code> and <code>__next__()</code>
Examples	List, Tuple, String, Dictionary, Set	Objects returned by <code>iter()</code>

- An iterable is an object that can be looped over (e.g. in a `for` loop).
- An iterator is the object returned by calling `iter()` on an iterable.

## Generators in Python

A **generator function** is a special type of function that returns an iterator object. Instead of using return to send back a single value, generator functions use yield to produce a series of results over time. This allows the function to generate values and pause its execution after each yield, maintaining its state between iterations.

### Example:

```
def fun(max):  
    cnt = 1  
    while cnt <= max:  
        yield cnt  
        cnt += 1
```

```
ctr = fun(5)  
for n in ctr:  
    print(n)
```

### Output

```
1  
2  
3  
4  
5
```

**Explanation:** This generator function fun yields numbers from 1 up to a specified max. Each call to **next()** on the generator object resumes execution right after the yield statement, where it last left off.

### Why Do We Need Generators?

- **Memory Efficient** : Handle large or infinite data without loading everything into memory.
- **No List Overhead** : Yield items one by one, avoiding full list creation.
- **Lazy Evaluation** : Compute values only when needed, improving performance.
- **Support Infinite Sequences** : Ideal for generating unbounded data like Fibonacci series.
- **Pipeline Processing** : Chain generators to process data in stages efficiently.

Let's take a deep dive in python generators:

### Creating Generators

Creating a generator in Python is as simple as defining a function with at least one yield statement. When called, this function doesn't return a single value; instead, it returns a generator object that supports the iterator protocol. The generator has the following syntax in Python:

<i>def</i>		<i>generator_function_name(parameters):</i>
<i>#</i>	<i>Your</i>	<i>code</i>
<i>yield</i>		<i>here</i>
<i># Additional code can follow</i>		<i>expression</i>

**Example:** we will create a simple generator that will yield three integers. Then we will print these integers by using Python for loop.

```
def fun():  
    yield 1  
    yield 2  
    yield 3
```

**# Driver code to check above generator function**

```
for val in fun():  
    print(val)
```

### Output

```
1  
2  
3
```

### Yield vs Return

- **Yield:** is used in generator functions to provide a sequence of values over time. When yield is executed, it pauses the function, returns the current value and retains the state of the function. This allows the function to continue from same point when called again, making it ideal for generating large or complex sequences efficiently.
- **Return:** is used to exit a function and return a final value. Once return is executed, function is terminated immediately and no state is retained. This is suitable for cases where a single result is needed from a function.

### Example with return:

```
def fun():  
    return 1 + 2 + 3
```

```
res = fun()  
print(res)
```

### Output

```
6
```

### Generator Expression

**Generator expressions** are a concise way to create generators. They are similar to list comprehensions but use parentheses instead of square brackets and are more memory efficient.

### Syntax:

*(expression for item in iterable)*

**Example:** We will create a generator object that will print the squares of integers between the range of 1 to 6 (exclusive).

```
sq = (x*x for x in range(1, 6))  
for i in sq:  
    print(i)
```

### Output

```
1  
4  
9  
16  
25
```

### Applications of Generators in Python

Suppose we need to create a stream of Fibonacci numbers. Using a generator makes this easy, you just call `next()` to get the next number without worrying about the stream ending.

- Generators are especially useful for processing large data files, like logs, because:
- They handle data in small parts, saving memory
- They don't load the entire file at once
- While iterators can do similar tasks, generators are quicker to write since you don't need to define `__next__` and `__iter__` methods manually.