Python Training Program

- Programming Basics
- Introduction to Python Programming
- Python Data Types and Operators
- Conditional Statements and Loops
- Python Functions
- Fundamentals of NumPy Library
- Working With Pandas
- Statistics Fundamentals
- Transition from Excel VBA to Python
- Overview of Python Libraries for Excel Automation(openpyxl, pandas)
- Creating Python-based Excel Plugins/Add-ins
- Advanced Excel Integration
- Creating Custom Excel Plugins to Execute Python Scripts:
- Integrating Dashboards and Advanced Visualizations with Python & Excel
- Automating Data Fetching with APIs
- Parsing and Processing JSON/XML Data:
- Integrating API Data with Excel
- Optical character recognition
- Data Extraction from PDFs
- Version Control with Git

Python Refresher - Day 1 & 2

File formats -

1. python file - filename.py (pycharm, VScode, spyder, notepad) and execute it on terminal
2. jupyter notebook - filename.ipynb (jupyter notebook, google colab)

Python Interface

A Python interface refers to the means through which you can interact with Python programs, libraries, or external systems. Here are a few different ways you can interface with Python:

1. Interactive Python Shell (REPL)

- **Description**: The Python Shell, also known as the Read-Eval-Print Loop (REPL), is an interactive command-line interface where you can type and execute Python commands

one at a time. This is useful for quick experiments and debugging.
- **Example**: You can start the Python Shell by simply typing `python` or `python3` in your terminal or command prompt.

2. Integrated Development Environments (IDEs)

- **Popular IDEs**: PyCharm, Visual Studio Code, Spyder, Jupyter Notebook.
- **Description**: IDEs provide a comprehensive environment for writing, testing, and debugging Python code. They come with features like syntax highlighting, code completion, version control integration, and more.
- **Example**: PyCharm offers an advanced interface with debugging tools, refactoring support, and integration with version control systems like Git.

3. Jupyter Notebooks

- **Description**: Jupyter Notebooks are an interactive web-based environment where you can combine code execution, text, and visualizations in a single document. This is particularly useful for data analysis, machine learning, and teaching.
- **Example**: You can run Jupyter Notebooks by installing the Jupyter package and starting a notebook server with the command `jupyter notebook`.

4. Command-Line Interface (CLI)

- **Description**: Python scripts can be executed directly from the command line. You can write Python programs that take command-line arguments and perform tasks based on those arguments.
- **Example**: A Python script named `myscript.py` can be run using `python myscript.py` in the terminal.

5. Google Colab Notebook

- Can be access using - https://colab.research.google.com/
- Requires google sign-in

Each of these interfaces serves different purposes and can be chosen based on the specific requirements of your project or task.

Features of Python Programming Language-

Python is a versatile and powerful programming language that is widely used in various fields. Here are some of its key features:

1. Easy to Learn and Use

- **Readability**: Python has a clear and easy-to-read syntax which makes it accessible for beginners.
- **Minimal Syntax**: Python code is concise and easy to write, reducing the need for complex boilerplate code.

2. Interpreted Language

- **No Compilation**: Python is an interpreted language, meaning you can run the code directly without needing to compile it first.
- **Interactive Mode**: Python provides an interactive mode, allowing you to execute code line by line and test small snippets quickly.

3. High-Level Language

- **Abstracted Details**: Python abstracts many low-level details such as memory management, making it easier to focus on the logic of the code.
- **Built-in Data Types**: Python includes powerful built-in data types such as lists, tuples, sets, and dictionaries.

4. Dynamically Typed

- **No Explicit Declarations**: Variable types are determined at runtime, eliminating the need for explicit type declarations.
- **Flexibility**: This dynamic typing provides flexibility in coding and faster prototyping.

5. Extensive Standard Library

- **Wide Range of Modules**: Python's standard library includes modules for various tasks such as file I/O, system calls, web development, and data manipulation.
- **Out-of-the-Box Functionality**: Many common programming tasks can be accomplished without the need for external libraries.

6. Cross-Platform Compatibility

- **Portable**: Python code can run on various operating systems such as Windows, macOS, and Linux without modification.
- **Platform Independence**: Python's platform independence makes it ideal for developing cross-platform applications.

7. Object-Oriented Programming

- **Class Support**: Python supports object-oriented programming with classes and inheritance, allowing for modular and reusable code.
- **Encapsulation and Polymorphism**: Python supports encapsulation and polymorphism, essential features for building complex applications.

8. Large Community and Ecosystem

- **Active Community**: Python has a large and active community that contributes to its continuous improvement.
- **Rich Ecosystem**: There are numerous third-party libraries and frameworks available, such as NumPy for numerical computing, Pandas for data manipulation, Flask and Django for web development, and TensorFlow and PyTorch for machine learning.

9. Integration Capabilities

- **Interoperability**: Python can easily integrate with other languages and technologies, such as C, C++, Java, and .NET.
- **Scripting**: It can be used as a scripting language to automate tasks and enhance the functionality of existing applications.

10. Strong Support for Data Science and Machine Learning

- **Data Analysis**: Python is extensively used in data analysis with libraries like Pandas, Matplotlib, and Seaborn.
- **Machine Learning**: Python is a popular choice for machine learning and artificial intelligence with libraries like TensorFlow, Keras, and Scikit-Learn.

Python's combination of readability, flexibility, and an extensive ecosystem makes it an excellent choice for both beginners and experienced developers. Whether you're building web applications, data analysis tools, or machine learning models, Python provides the tools and features you need to succeed.

Python Data Types

Python has several built-in data types that allow you to store and manipulate different kinds of data. Here are the primary data types in Python:

Data Type	Description	Examples
int	Integer numbers	42, -7
float	Floating-point numbers (decimal)	3.14, -0.001
complex	Complex numbers	1+2j, -3+4j
bool	Boolean values	True, False
str	String, a sequence of characters	"hello", 'world'

Data Type	Description	Examples
bytes	Immutable sequence of bytes	b'hello'

Operators in Python

Operators are special symbols in Python that carry out computations. The value that the operator operates on is called as operand.

1. Arithmetic Operators

These operators perform arithmetic operations on numeric values.

- `+` : Addition
- `-` : Subtraction
- `*` : Multiplication
- `/` : Division
- `%` : Modulus (remainder of division)
- `**` : Exponentiation (power)
- `//` : Floor division (division that results in the largest integer less than or equal to the quotient)

2. Comparison Operators

These operators compare two values and return a boolean result (True or False).

- `==` : Equal to
- `!=` : Not equal to
- `>` : Greater than
- `<` : Less than
- `>=` : Greater than or equal to
- `<=` : Less than or equal to

3. Logical Operators

These operators are used to combine conditional statements.

- `and` : Returns True if both statements are true
- `or` : Returns True if at least one of the statements is true
- `not` : Reverses the result, returns False if the result is true

4. Basic Assignment Operator

`=` : Assigns the value on the right to the variable on the left.

Compound Assignment Operators

These operators perform an operation on a variable and then assign the result back to that variable.

- `+=` : Adds the right operand to the left operand and assigns the result to the left operand.
- `-=` : Subtracts the right operand from the left operand and assigns the result to the left operand.
- `*=` : Multiplies the left operand by the right operand and assigns the result to the left operand.
- `/=` : Divides the left operand by the right operand and assigns the result to the left operand.
- `%=` : Takes the modulus of the left operand by the right operand and assigns the result to the left operand.
- `//=` : Performs floor division on the left operand by the right operand and assigns the result to the left operand.
- `**=` : Raises the left operand to the power of the right operand and assigns the result to the left operand.

Membership Operators

Membership operators are used to test whether a value or variable is found in a sequence (such as a string, list, tuple, set, or dictionary). There are two membership operators in Python:

- `in` - The in operator checks if a value is present in a sequence.
- `not in` - The not in operator checks if a value is not present in a sequence.

In [2]:
```
# Comments in python
```

Examples -

Ex. WAP to take 2 numbers as input and print their addition

In [3]:
```
num1 = input("Enter a number - ")
num2 = input("Enter a number - ")
print(num1 + num2)
```

```
57
```

- **Note - input() will always store the result in str format**
- use type(`var`) to get the datatype of the variable

In [4]:
```
type(num1)
```

Out[4]:
```
str
```

In [6]:
```
num1 = int(input("Enter a number - "))
num2 = int(input("Enter a number - "))
print("Addition - ", num1 + num2)
```

```
Addition -  12
```

Type Conversion Functions - int(), float(), str(), bool()

Ex. Calculate Gross Pay

- Example 1 - WAP to accept hours and rate per hour from user and compute gross pay.

In [7]:
```
hrs = int(input("Enter number of hrous worked - "))
rate = int(input("Enter rate per hour - "))
gross_pay = hrs * rate
print("Gross Pay - ", gross_pay)
```

```
Gross Pay -  40000
```

- Define variables
- take input from user
- operators in python
- data type conversion

Ex. Convert the sales value in thousands and concate k as suffix

In [13]:
```
sales = 53000
str(sales//1000) + "k"
```

Out[13]:
```
'53k'
```

In [16]:
```
sales = 53875
str(round(sales/1000, 1)) + "k"
```

Out[16]:
```
'53.9k'
```

In [19]:
```
sales = 53875
f"${round(sales/1000, 1)}k"  # generates a str object by combining text and {calcul
```

Out[19]:
```
'$53.9k'
```

In [ ]:
```
str(53)  # - converting int value to str format
"abcd" + "pqr"  # - concatatenation
```

In [20]:
```
a = 5
b = 10
c = a + b
f"Addition of {a} and {b} is {c}"  # this is an example of formatted string
```

Out[20]: 'Addition of 5 and 10 is 15'

Decision Making

Decision-making statements in Python allow you to control the flow of execution based on certain conditions. These statements include if, elif, and else and can be used to execute different blocks of code depending on whether conditions are True or False.

- ## Basic if Statement
  - The if statement evaluates a condition and executes a block of code if the condition is True.
  - Syntax:
    ```
    if condition:
        # block of code
    ```
- ## if-else Statement
  - The if-else statement evaluates a condition and executes one block of code if the condition is True, and another block if the condition is False.
  - Syntax:
    ```
    if condition:
        # block of code if condition is True
    else:
        # block of code if condition is False
    ```
- ## if-elif-else Statement
  - The if-elif-else statement allows you to check multiple expressions for True and execute a block of code as soon as one of the conditions evaluates to True.
  - If none of the conditions are True, the else block is executed.
  - Syntax:
    ```
    if condition1:
        # block of code if condition1 is True
    elif condition2:
        # block of code if condition2 is True
    elif condition3:
        # block of code if condition3 is True
    else:
        # block of code if none of the conditions are True
    ```
- ## Nested if Statements

- You can nest if statements within other if statements to check multiple conditions in a hierarchical manner.
- Syntax:

```
if condition1:
    # block of code if condition1 is True
    if condition2:
        # block of code if condition2 is True
    else condition3:
        # block of code if condition2 is False
else:
    # block of code if condition1 is False
```

One-Line if-else

- Syntax:

```
value_if_true if condition else value_if_false
```

Examples -

Ex. WAP to check if a number is even or odd

In [22]:
```
num = int(input("Enter a number - "))
if num % 2 == 0 :
    print(num, "is even number")
else :
    print(num, "is odd number")
```

```
5 is odd number
```

Ex. Calculate Gross Pay

- Example 2 - Compute gross pay based on a condition
  - Take hours worked and rate per hour as input from the user.
  - If the hours worked are 40 or less, apply the given rate.
  - If the hours worked exceed 40, apply the given rate for the first 40 hours and 1.5 times the rate for the additional hours as overtime pay.

In [25]:
```
hrs = int(input("Enter number of hrous worked - "))
rate = int(input("Enter rate per hour - "))
if hrs <= 40 :
    gross_pay = hrs * rate
else:
    gross_pay = (40 * rate) + ((hrs - 40) * 1.5 * rate)

print("Gross Pay - ", gross_pay)
```

```
Gross Pay -  47500.0
```

In [ ]:
```
40 1000 = 40000
45 1000 = 47500
```

Loops/Iteration

Loops are used to execute of a specific block of code in repetitively

while loop

- The 'while loop' in Python is used to iterate over a block of code as long as the test expression holds true
- Event based loop
- Event occurring inside the loop determines the number of iterations
- This loop is used when the number of times to iterate is not known to us beforehand

for loop

- The 'for loop' in Python is used to iterate over the items of a sequence object like list, tuple, string and other iterable objects
- The iteration continues until we reach the last item in the sequence object
- Counter driven loop
- This loop is used when the number of times to iterate is predefined

break statement

- The 'break' statement ends the loop and resumes execution at the next statement
- The break statement can be used in both 'while' loop and 'for' loop
- It is always used with conditional statements

continue statement

- The 'continue' statement in Python ignores all the remaining statements in the iteration of the current loop and moves the control back to the beginning of the loop
- The continue statement can be used in both 'while' loop and 'for' loop
- It is always used with conditional statements

Examples -

```
In [26]: "abcd"  # - character sequence
```

```
Out[26]: 'abcd'
```

```
In [29]: [10, 20, 30, 40, "abcd", "pqr"] # - container sequence - sequence of objects
```

```
Out[29]: [10, 20, 30, 40, 'abcd', 'pqr']
```

In [28]:
```
range(1, 6) # - generate a sequence of int from 1 to 5, doesnt return readable outp
```

Out[28]:
```
range(1, 6)
```

Ex. WAP to print square of numbers in the given list

In [31]:
```
lst = [10, 20, 30, 40, 50 , 60 , 70]
for i in lst :
    print(i, " - ", i**2)
```

```
10  -  100
20  -  400
30  -  900
40  -  1600
```

Can we print square of only even numbers from the list

In [35]:
```
lst = [1, 2, 3, 4, 5, 6, 7]
for i in lst :
    if i % 2 == 0 :
        print(i, " - ", i**2)
```

```
2  -  4
4  -  16
6  -  36
```

Ex. WAP to perform summation and product of first 10 natural numbers

In [38]:
```
sum(range(1, 11)) # - returns summation of sequence of int numbers,
```

Out[38]:
```
55
```

In [39]:
```
import math  # import a library

math.prod(range(1, 11))
```

Out[39]:
```
3628800
```

Ex. WAP to accept input from till he enters and int in the range of 1- 9

In [42]:
```
num = 0
while num not in range(1, 10) :  # 1 - 9
    num = int(input("Enter a number - "))
num
```

Out[42]:
```
5
```

In [45]:
```
while True :   # infinite loop - never terminates
    num = input("Enter a number in range of 1 - 9 : ")
    if num.isdigit()  :  # str method to check if all characters are digit or not
        num = int(num)
        if num in range(1, 10) :
            break  # terminates the loop - mandatory in case of infinite loops
        else:
            print("Number must be in range of 1-9")
```

```
    else:
        print("Please enter an int value")

num
```

```
Please enter an int value
Number must be in range of 1-9
```

Out[45]: 5

Ex. Seven-up Seven-down

Game Rules

1. **Initial Setup:**
   - The player starts with an initial amount of Rs. 1000.
   - The balance amount from the previous game will be carried forward to the next round.
2. **Game Mechanics:**
   - An outcome is generated as a random number in the range of 1-17, which is displayed as "Score".
   - Depending on the outcome:
     - If the outcome is exactly 7, the player hits the jackpot and wins Rs. 1,00,000.
     - If the outcome is less than 7, the player loses an amount equal to (outcome * 100).
     - If the outcome is greater than 7, the player wins an amount equal to (outcome * 100).
3. **Game Continuation:**
   - After each round, the player can choose to continue playing or to "quit" the game.
   - If the player hits the jackpot (outcome = 7), the game automatically stops.
4. **Financial Management:**
   - If the player's balance falls below Rs. 600 after a round, they have the option to top-up their balance by Rs. 1,000 to continue playing.
   - If the player chooses not to top-up and their balance is insufficient to continue (less than Rs. 600), the game ends.
5. **Outcome Display:**
   - For each round, the outcome (score) and the resulting financial changes (win, lose, jackpot) are clearly displayed to the player.
6. **Game Termination:**
   - The game ends either when the player decides to quit or when they hit the jackpot (outcome = 7).

In [55]:
```python
import random
amount = 1000
choice = "yes"

while choice == "yes" :
    outcome = random.randint(1, 14)
    print("Your Score - ", outcome)

    if outcome < 7 :
        amount -= (outcome * 100)
    elif outcome > 7 :
        amount += (outcome * 100)
    else:
        print("Congratulations!!! you have hit the jackpot")
        amount += 10000000
        break

    if amount < 600 :
        print(f"Insufficient Balance - {amount}.")
        choice = input("Do you wish to top by 1000 to continue? yes/no").lower()
        if choice == "yes" :
            amount += 1000
            continue  # - skips the rest of the code and jumps to next iteration
        break

    print(f"Your current balance is {amount}.")
    choice = input("Do you wish to continue? yes/no ?").lower()

print(f"Your final amount is - {amount}")
```

```
Your Score -  6
Insufficient Balance - 400.
Your Score -  13
Your current balance is 2700.
Your final amount is - 2700
```

Sequence objects or Iterables

- collection of elements - str, range(), enumerate(), zip(), map(), filter()
- Container sequences/Objects - list, tuple, dict, set
- Note - range(), enumerate(), zip(), map(), filter() - these are generating non-readable output when printed.
- Note - Any sequence can be converted to list/tuple

`Operations on Generic Sequences`

```
- Membership - in | not in
- Iteration - for-loop`
```

Operations on Ordered/Indexed Sequences

```
- Indexing - obj[index_pos]
- Slicing - obj[start : stop]
- Concatenation - `+` operator
- Repeatition - `*` operator
```

Functions on Generic Sequences

```
 - len() - gives the number of elements in the sequence
 - max() - gives the largest element in the sequence
 - min() - gives the smallest element in the sequence
 - sum() - applicable to numeric sequences, returns the sum of all
elements in the sequence
 - math.prod() - applicable to numeric sequences, returns the
product of all elements in the sequence
 - sorted() - sorts the elements in the sequence in ASC order and
returns a list object
```

Python Sequences and Containers

Object	Container Object	Sequence Type	Element Type	Enclosed in	Immutability	Duplicates
str()	No	ordered/indexed	characters	"" or ''	Yes	Yes
tuple()	Yes	ordered/indexed	mixed data (heterogeneous)	()	Yes	Yes
list()	Yes	ordered/indexed	mixed data (heterogeneous)	[]	No	Yes
set()	Yes	unordered	heterogeneous (immutable objects)	{}	No	No
dict()	Yes	unordered	Key - immutable Value - any type	{}	No	Key - No Value - Yes

In [56]:
```
"abcd" # str
```

Out[56]:
```
'abcd'
```

In [57]:
```
(1, 2, 3, 4)  # tuple
```

Out[57]: (1, 2, 3, 4)

In [58]:
```
[1, 2, 3, 4] # list
```

Out[58]: [1, 2, 3, 4]

In [59]:
```
{1, 2, 3, 2, 4, 5} # set
```

Out[59]: {1, 2, 3, 4, 5}

In [60]:
```
{"Jane" : 50000, "Jack" : 80000, "Sam" : 40000} # dict
```

Out[60]: {'Jane': 50000, 'Jack': 80000, 'Sam': 40000}

Examples -

Ex. Vowels in a String

WAP to extract and replace all the vowels in a string

In [61]:
```
string = "singapore"
for i in string :
    if i in "aeiou" :
        print(i)
```

```
i
a
o
e
```

In [62]:
```
string.replace("a", "*")
```

Out[62]: 'sing*pore'

In [64]:
```
string = "singapore"
for i in string :
    if i in "aeiou" :
        string = string.replace(i, "*")  # strings are immutable
string
```

Out[64]: 's*ng*p*r*'

In [70]:
```
obj = str.maketrans("aeiou", "*****")
string.translate(obj)
```

Out[70]: 's*ng*p*r*'

Ex. Working on currency values

WAP to convert the given sales value to int. profit = $1,200

In [69]:
```
sales = "$5000"
int(sales.replace("$", ""))
```

Out[69]: 5000

In [73]:
```
profit = "($1,200)"
obj = str.maketrans("(", "-", "$),")
int(profit.translate(obj))
```

Out[73]: -1200

In [74]:
```
print(dir(str))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq_
_', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__
getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__l
e__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '_
_str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalp
ha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isp
rintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'mak
etrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'str
ip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Ex. Calculate Percentage

Write a program to compute the percentages of 10 students from the provided list. Display the results in a tabular format showing each student's ID and percentage. The student IDs should be generated sequentially starting from 101.

In [78]:
```
marks = [(51, 67, 83), (41, 93, 36), (50, 31, 87), (94, 46, 52), (80, 61, 69), (72,

for i in marks :
    print(round(sum(i)/3, 1))
```

```
67.0
56.7
56.0
64.0
70.0
63.0
57.3
52.3
67.3
60.3
```

In [81]:
```
# The enumerate object yields pairs containing a count (from start, which defaults
lst = [10, 20, 30, 40]
list(enumerate(lst))
```

Out[81]: [(0, 10), (1, 20), (2, 30), (3, 40)]

In [82]:
```
lst = [10, 20, 30, 40]
list(enumerate(lst, start = 101))
```

Out[82]: [(101, 10), (102, 20), (103, 30), (104, 40)]

In [84]:
```
for i, j in enumerate(lst, start = 101) :
    print(i, " - ", j)
```

```
101  -  10
102  -  20
103  -  30
104  -  40
```

In [91]:
```
marks = [(51, 67, 83), (41, 93, 36), (50, 31, 87), (94, 46, 52), (80, 61, 69), (72,
print(f"ID \t Percentage")
print("-"*20)
for i, j in enumerate(marks, start = 101) :
    print(f"{i} \t {round(sum(j)/3, 1)}%")
```

```
ID       Percentage
--------------------
101      67.0%
102      56.7%
103      56.0%
104      64.0%
105      70.0%
106      63.0%
107      57.3%
108      52.3%
109      67.3%
110      60.3%
```

Python Revision

Ex. WAP to take name of book as input from user and check if it is present in the inventory

In [3]:
```
available_books = ["Python 101", "Data Science Handbook", "Machine Learning", "Deep
book_name = input("Enter book name - ")
if book_name in available_books  :
    print("Yes")
else:
    print("No")
```

```
Yes
```

Ex. WAP to calculate total amount credited to the account and total amount debited from account based on following transactions -

In [6]:
```
transactions = [2000, -500, 1500, -1200, -100, 3000]
total_credit = 0
total_debit = 0
for amount in transactions :
    if amount > 0 :
        total_credit += amount
    else:
        total_debit += amount
```

```
print(f"Total amount credited - {total_credit} | total debited amount - {abs(total_
```

```
Total amount credited - 6500 | total debited amount - 1800
```

Ex. WAP to display the names of the products which are out of stock from the given inventory

In [8]:
```
inventory = {"Shoes": 10, "Bags": 5, "Watches": 0, "Belts": 3}
inventory["Bags"] # Extract value from the dict based on keys
```

Out[8]:
```
5
```

In [10]:
```
# option 1 - to extract all the values
for product in inventory:
    print(product, " - ", inventory[product])
```

```
Shoes  -  10
Bags  -  5
Watches  -  0
Belts  -  3
```

In [12]:
```
inventory.items() # Sequence of tuples of key value pairs
```

Out[12]:
```
dict_items([('Shoes', 10), ('Bags', 5), ('Watches', 0), ('Belts', 3)])
```

In [13]:
```
# option 2 - to extract all the values
for i, j in inventory.items():
    print(i, " - ", j)
```

```
Shoes  -  10
Bags  -  5
Watches  -  0
Belts  -  3
```

In [16]:
```
for product, quantity in inventory.items():
    if quantity == 0 :
        print(f"{product} is out of stock")
    elif quantity < 5 :
        print(f"{product} is low stock")
```

```
Watches is out of stock
Belts is low stock
```

Comprehensions in Python

- A **`comprehension`** is a concise and readable way to create and manipulate collections such as lists, dictionaries, and sets.
- Comprehensions provide a compact syntax to generate new sequences by applying an expression to each item in an existing sequence or iterable.
- Syntax - `[<expression> for <var> in <sequence> if <condition>]`
- The steps to work on a comprehension:

1. Identify the iterable or sequence.
2. Determine any conditions or filters.
3. Define the expression or operation to apply.
4. Specify the target mutable data structure.

- Note - Do not work with while loop, break and continue statements

Ex. WAP to add 7% service tax to all the values in the "sales" list

In [18]:
```
sales = [290, 500, 800, 650]
result = [i * 1.07 for i in sales]
print(result)
```

```
[310.3, 535.0, 856.0, 695.5]
```

Ex. WAP to create a list of squares of numbers from 1-5

In [19]:
```
squares = [i**2 for i in range(1, 6)]
print(squares)
```

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

Ex. WAP to create a list of squares of even numbers from 1-20

In [21]:
```
squares = [i**2 for i in range(1, 21) if i % 2 == 0]
print(squares)
```

```
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400]
```

Ex. WAP to create a dict with key as number and its square as its value for numbers from 1-5

In [22]:
```
squares = {i : i**2 for i in range(1, 6)}
print(squares)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Ex. Convert the list of sales value in thousands and concate k as suffix and store them in a new list

In [23]:
```
sales = [53000, 20000, 55000, 85000]
[f"{i//1000}k" for i in sales]
```

Out[23]:
```
['53k', '20k', '55k', '85k']
```

Ex. Create a list of products with low stock levels.

In [27]:
```
inventory = {"apple": 50, "banana": 10, "cherry": 75, "date": 5}
[product for product, quantity in inventory.items() if quantity <= 10 ]
```

Out[27]:
```
['banana', 'date']
```

Functions in Python

Function arguments in Python are the inputs passed to a function to enable it to perform its intended operation.

Types of Function Arguments

1. **Positional Arguments**
   - These are the most common type.
   - Passed in order, matching the function's parameters.
2. **Default Arguments**
   - Predefined values are set; they are optional while calling the function.
3. **Keyword Arguments**
   - Arguments passed with the name of the parameter explicitly.
   - Useful for clarity and flexibility in function calls.
4. **Variable-Length Arguments**
   - `*args` **(Non-Keyword Arguments):** Allows passing a variable number of arguments as a tuple.
   - ** `**kwargs` (Keyword Arguments):** Allows passing variable keyword arguments as a dictionary.

Key Notes

- **Order Matters:** Positional arguments must precede keyword arguments.
- **Packing/Unpacking:** `*` and `**` are used to pack or unpack arguments, making functions more versatile.
- **Default vs Non-Default:** Non-default arguments must come first when defining functions.

In [32]:
```python
# Required Positional Argument
def demo(name, age) :
    print(f"Name - {name} | Age - {age}")

demo("Jane", 30)
demo(30, "Jane")
demo("Jane")
```

```
Name - Jane | Age - 30
Name - 30 | Age - Jane
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[32], line 6
      4 demo("Jane", 30)
      5 demo(30, "Jane")
----> 6 demo("Jane")

TypeError: demo() missing 1 required positional argument: 'age'
```

In [35]:
```
# Default Argument
def demo(name, age = 25) :
    print(f"Name - {name} | Age - {age}")

demo("Jane", 30)
demo(30, "Jane")
demo("Jane")
```

```
Name - Jane | Age - 30
Name - 30 | Age - Jane
Name - Jane | Age - 25
```

Examples

In [36]:
```
len("abcd")
```

Out[36]:
```
4
```

In [37]:
```
len()  # Required argument
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[37], line 1
----> 1 len()

TypeError: len() takes exactly one argument (0 given)
```

In [38]:
```
string = "mississippi"
string.replace("i", "*") # Positional argument
```

Out[38]:
```
'm*ss*ss*pp*'
```

In [39]:
```
string = "mississippi"
string.replace("*", "i") # Positional argument
```

Out[39]:
```
'mississippi'
```

In [45]:
```
string = "mississippi"
string.replace("i", "*", 2) # default argument
```

Out[45]:
```
'm*ss*ssippi'
```

In [40]:
```
help(string.replace)
```

```
Help on built-in function replace:

replace(old, new, count=-1, /) method of builtins.str instance
    Return a copy with all occurrences of substring old replaced by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count occurrences are
    replaced.
```

In [48]:
```python
string[::-1].replace("i", "*", 2)[::-1]
```

Out[48]:
```
'mississ*pp*'
```

In [52]:
```python
list(range(6))  # default argument
```

Out[52]:
```
[0, 1, 2, 3, 4, 5]
```

In [53]:
```python
# Variable length Argument
def demo(name, *args, age = 25) :
    print(f"Name - {name} | Age - {age} | Args - {args}")

demo("Jane", 10, 20, 30, 40, 50, 30)
```

```
Name - Jane | Age - 25 | Args - (10, 20, 30, 40, 50, 30)
```

In [54]:
```python
# key-word Argument
demo("Jane", 10, 20, 30, 40, 50, age = 30)
```

```
Name - Jane | Age - 30 | Args - (10, 20, 30, 40, 50)
```

In [58]:
```python
# Variable length Key - Argument
def demo(name, *args, age = 25, **kwargs) :
    print(f"Name - {name} | Age - {age} | Args - {args} | kwargs - {kwargs}")

demo("Jane", 10, 20, 30, 40, 50, age = 30, mob = 9876543, gender = "F")
```

```
Name - Jane | Age - 30 | Args - (10, 20, 30, 40, 50) | kwargs - {'mob': 9876543, 'ge
nder': 'F'}
```

Significance of `/` and `*` in function definition

- `/` - All the arguments before `/` must be position only
- `*` - All the arguments after `*` must be key-word only

In [59]:
```python
def demo(name, age) :
    print(f"Name - {name} | Age - {age}")

demo("Jane", 30) # positional-only
demo("Jane", age = 30) # name - positional | age = key-word
demo(name = "Jane", age = 30) # key-word only
```

```
Name - Jane | Age - 30
Name - Jane | Age - 30
Name - Jane | Age - 30
```

In [61]:
```python
def demo(name, /, age) :
    print(f"Name - {name} | Age - {age}")

demo("Jane", 30)
demo("Jane", age = 30)
demo(name = "Jane", age = 30) # Error
```

```
Name - Jane | Age - 30
Name - Jane | Age - 30
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[61], line 6
      4 demo("Jane", 30)
      5 demo("Jane", age = 30)
----> 6 demo(name = "Jane", age = 30) # Error

TypeError: demo() got some positional-only arguments passed as keyword arguments: 'n
ame'
```

In [63]:
```python
string.replace("i", "*", 2)
```

Out[63]:
```
'm*ss*ssippi'
```

In [64]:
```python
string.replace("i", "*", count = 2) # / is present in the function definition
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[64], line 1
----> 1 string.replace("i", "*", count = 2)

TypeError: str.replace() takes no keyword arguments
```

In [66]:
```python
list(enumerate([1, 2, 3, 4], 101))
```

Out[66]:
```
[(101, 1), (102, 2), (103, 3), (104, 4)]
```

In [67]:
```python
list(enumerate([1, 2, 3, 4], start = 101))
```

Out[67]:
```
[(101, 1), (102, 2), (103, 3), (104, 4)]
```

In [65]:
```python
help(enumerate)
```

```
Help on class enumerate in module builtins:

class enumerate(object)
 |  enumerate(iterable, start=0)
 |
 |  Return an enumerate object.
 |
 |    iterable
 |      an object supporting iteration
 |
 |  The enumerate object yields pairs containing a count (from start, which
 |  defaults to zero) and a value yielded by the iterable argument.
 |
 |  enumerate is useful for obtaining an indexed list:
 |      (0, seq[0]), (1, seq[1]), (2, seq[2]), ...
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(...)
 |      Return state information for pickling.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...)
 |      See PEP 585
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs)
 |      Create and return a new object.  See help(type) for accurate signature.
```

In [68]:

```python
def demo(name, *, age) : # Arguments after * must be key-word only
    print(f"Name - {name} | Age - {age}")

demo("Jane", age = 30)
demo(name = "Jane", age = 30)
demo("Jane", 30)
```

```
Name - Jane | Age - 30
Name - Jane | Age - 30
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[68], line 6
      4 demo("Jane", age = 30)
      5 demo(name = "Jane", age = 30)
----> 6 demo("Jane", 30)

TypeError: demo() takes 1 positional argument but 2 were given
```

In [69]:
```
help(sorted)
```

```
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

In [72]:
```
sorted([1, 3, 2, 5, 4])  # Returns a list in sorted order
```

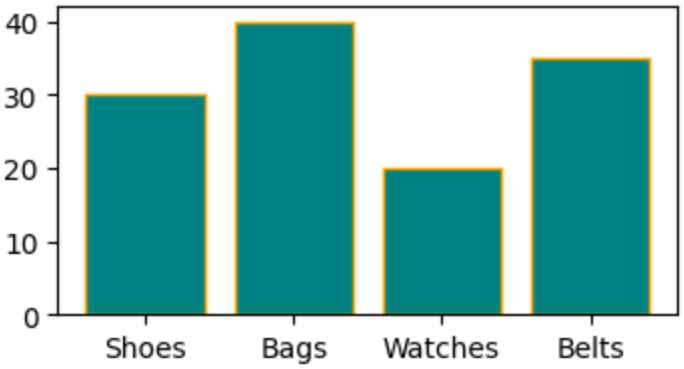
Out[72]:
```
[1, 2, 3, 4, 5]
```

In [74]:
```
sorted([1, 3, 2, 5, 4], reverse=True)  # Returns a list in sorted order DESC
```

Out[74]:
```
[5, 4, 3, 2, 1]
```

In [80]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(4, 2))
products = ["Shoes", "Bags", "Watches", "Belts"]
quantity = [30, 40, 20, 35]

# Create a bar chart
plt.bar(products, quantity, color = "Teal", edgecolor = "orange")
plt.show()
```



Packing/Unpacking of Tuples -

- In Python, unpacking of tuples refers to the process of assigning the individual elements of a tuple to multiple variables in a single statement.

- This feature allows you to extract values from a tuple and assign them to distinct variables in a convenient and readable way.

In [81]:
```
# packing of tuples
tup = 10, 20, 30
tup
```

Out[81]: (10, 20, 30)

In [82]:
```
# unpacking of tuples
a, b, c = tup
print(a)
```

```
10
```

Applications of packing and unpacking

1. Defining multiple variables in a single line

In [83]:
```
name, age = "Jane", 30
print(name, age)
```

```
Jane 30
```

2. Function returing multiple values

In [87]:
```
def add(num1, num2) :
    total = num1 + num2
    product = num1 * num2
    return total, product  # packing of tuples

print(add(5, 6))
```

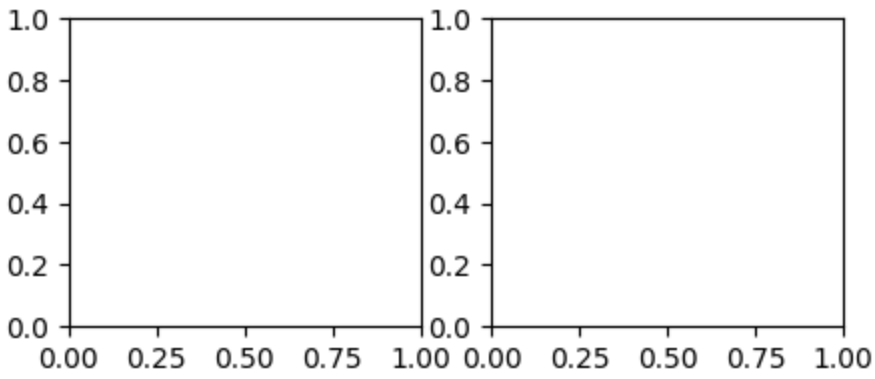
```
(11, 30)
```

In [90]:
```
t, p = add(5, 6) # - unpacking of tuples
print(f"Total - {t} | Product - {p}")
```

```
Total - 11 | Product - 30
```

In [89]:
```
fig, ax = plt.subplots(ncols=2, figsize = (5, 2)) # Returns a tuple object - unpack
print(ax)
```

```
[<Axes: > <Axes: >]
```



3. Function call

In [93]:
```
def demo(*args) :
    print(args)
demo(1, 2, 3, 4, 5)  # Packing of tuples as function call
```

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

In [96]:
```
def demo(name, age, gender):
    print(f"Name - {name} | Age - {age} | Gender {gender}")

data = ("Jane", 30, "F")
demo(data[0], data[1], data[2])
demo(*data)  # unpacking of values in the tuple to arguments
```

```
Name - Jane | Age - 30 | Gender F
Name - Jane | Age - 30 | Gender F
```

Function Object

- In Python, a function object refers to the fact that functions are first-class citizens—they are treated like any other object.
- This means: A function object is an instance of the built-in function class created when a def statement or a lambda expression is executed.
- A function object can be:
  - Pass it to other functions,
  - Assign it to variables,
  - Store in data structures,
  - Return it from another function.

Note - Everything in python is an object. Function is also an object

In [97]:
```
def func(a, b):
    if a < b :
        return a
    else:
        return b
```

In [98]:
```
var = func(2, 3)
print(var)
```

```
2
```

In [99]:
```
var = func  # stores the function object in the variable. var variable will get spe
print(var)
```

```
<function func at 0x000001B7874FFA60>
```

In [100…
```
var(2, 3)
```

Out[100… 2

Note - Any function, built-in or user-defined can be assigned to a variable

In [101…
```
var = len
```

In [102…
```
var("abcd")
```

Out[102… 4

Categories of functions -

- built-in function; eg - sum(), len(), math.sqrt(), random.randint(), plt.bar()
- user-defined; defined using def keyword
- user-defined; one-liner function; defined using keyword - `lambda`

Ex. define a lambda function to find square of number

In [103…
```
lambda num : num**2  # lambda always generates a function object; it doesnot have a
```

Out[103… <function __main__.<lambda>(num)>

In [105…
```
sq = lambda num : num**2 # lambda function can be stored in a variable
sq(4)
```

Out[105… 16

Applications of Function Object

- function object can be -
  - a built-in function
  - a user-define; using def keyword
  - a user-define; using lambda keyword

Ex. WAP to sort a list of strings as per the last character.

In [112…
```
names = ["flight", "bike", "car",  "train"]
sorted(names)  # sorts alphabetically
```

Out[112… ['bike', 'car', 'flight', 'train']

In [113…
```
# sort by number of chaarcters in each word
sorted(names, key = len)  # key parameter take function object as an argument to so
```

Out[113… ['car', 'bike', 'train', 'flight']

In [115…
```
# sort by last character of each word
sorted(names, key = lambda word : word[-1])
```

Out[115… ['bike', 'train', 'car', 'flight']

In [116…
```
min(names)
```

Out[116… 'bike'

In [117…
```
min(names, key = len)
```

Out[117… 'car'

In [118…
```
min(names, key = max)
```

Out[118… 'bike'

In [119…
```
max(names, key = max)
```

Out[119… 'flight'

Ex. WAP to create a new list of values calculate as average of each tuple using comprehension

In [121…
```
marks = [(51, 67, 83), (41, 93, 36), (50, 31, 87), (94, 46, 52)]
[round(sum(tup)/len(tup), 2) for tup in marks]
```

Out[121… [67.0, 56.67, 56.0, 64.0]

In [123…
```
[sum(tup) for tup in marks]
```

Out[123… [201, 170, 168, 192]

Ex. WAP to sort the given list of tuples by sum of the values in each tuple

In [122…
```
sorted(marks, key = sum)
```

Out[122… [(50, 31, 87), (41, 93, 36), (94, 46, 52), (51, 67, 83)]

Ex. WAP to sort the given list of tuples by avg of the values in each tuple

In [124…
```
sorted(marks, key = lambda tup : sum(tup)/len(tup))
```

Out[124… [(50, 31, 87), (41, 93, 36), (94, 46, 52), (51, 67, 83)]

Ex. Write a program to sort the given dictionary by values

HINT - use sorted(), dict.items()

In [130…
```
employee = {"Rosie" : 40000, "Jane" : 30000, "Jack" :50000, "George" : 45000}
sorted(employee.items(), key = lambda tup : tup[1])
```

Out[130… [('Jane', 30000), ('Rosie', 40000), ('George', 45000), ('Jack', 50000)]

In [131…
```
dict(sorted(employee.items(), key = lambda tup : tup[1]))
```

Out[131… {'Jane': 30000, 'Rosie': 40000, 'George': 45000, 'Jack': 50000}