Learn Python Easily, from Beginner to Expert

MASTER PYTHON

Technology Channel.

BY BISHWORAJ POUDEL

🌟 Introduction to Python

Python is World's most popular programming language. It is a beginner-friendly programming language widely used in web development, artificial intelligence, data analysis, automation, networking, and many other fields.

🚀 What Can You Do with Python?

- Web Development
- Automation
- Backend API Development
- AI and Machine Learning
- Data Analysis
- Web Scraping
- Game Development
- Cloud Computing
- Cybersecurity
- QA Testing
- Networking
- Internet of Things (IoT), etc.

🌟 Why Learn Python?

- Easy to Learn: Start coding quickly, even if you're a beginner
- Global Opportunities: Python skills are in demand worldwide
- Top Companies Use Python: Google, Facebook, NASA, and more
- Great Salary: Python developers earn an average of $126,952 annually in the US (according to Indeed.com)

🌐 Python is Used By:

- Google
- Instagram
- Netflix
- NASA
- And almost every tech company you can think of!

🖥️ Installing Python

Let's get Python up and running on your machine!

On Windows:

1. Download Python from the official website:
[python.org/downloads](https://www.python.org/downloads/)
2. Run the installer.
3. Check the box that says “Add Python to PATH”.
4. Click "Install Now".
5. Open Command Prompt and type `python --version` to check the installation.

On Mac:
1. Download Python from the official website:
[python.org/downloads](https://www.python.org/downloads/)
2. Run the installer.
3. Open Terminal and type `python3 --version` to check the installation.

On Linux:
1. Open Terminal.
2. Type `sudo apt-get update`.
3. Type `sudo apt-get install python3`.
4. Type `python3 --version` to check the installation.

💻 Installing Visual Studio Code (VS Code)

VS Code is a great code editor to write Python. Let's get it set up!

On Windows:
1. Download VS Code from the official website:
code.visualstudio.com
2. Run the installer.
3. Open Visual Studio Code.
4. Click on Extensions.
5. Search for "Python".
6. Click "Install".

On Mac:
1. Download VS Code from the official website:
code.visualstudio.com
2. Run the installer.
3. Open Visual Studio Code.
4. Click on Extensions.
5. Search for "Python".
6. Click "Install".

On Linux:
1. Download VS Code from the official website:
code.visualstudio.com
2. Run the installer.
3. Open Visual Studio Code.
4. Click on Extensions.
5. Search for "Python".
6. Click "Install".

🏃 Running Python Code in VS Code

Let's write and run your first Python code in VS Code!
1. Create a folder named `python-course`.
2. Open Command Prompt at the folder location.
3. Type `code .` to open VS Code.
4. Create a new file with a `.py` extension (e.g., `hello.py`).
5. Write your Python code in the file:

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

6. Save the file.
7. Open Command Prompt and type `python hello.py` to run the code.

🎉 Python Examples

Example 1: Print Your Name

```
print("John Doe")
```

Example 2: Print Your Name and Address

```
print("John Doe")
print("123 Main Street")
print("New York, NY 10001")
```

🧙 Escape Characters in Python

Escape characters let you include special characters in strings. Here are some handy ones:

Escape Character	Description
\n	New Line
\t	Tab
\“	Double Quotes
\’	Single Quotes
\\	Backslash

💡 Challenge 1: Write About Yourself

Write a little about yourself using 10 different print statements.

🌟 Variables In Python

Variables are like containers that store data values. In Python, you don’t need to declare the type of a variable, as the language is dynamically typed. This means you can just assign a value to a variable, and Python will handle the rest.

Syntax

```
variable_name = value
```

🚀 Why Use Variables?

- **Store Data:** Variables allow you to store data that can be used and manipulated throughout your code.
- **Readability:** They make your code more readable by giving meaningful names to data.
- **Flexibility:** Variables enable you to easily update and manage data values.

🚀 Rules For Variable Name

- Variable names are case sensitive, i.e., a and A are different.
- A variable name can consist of letters and alphabets.
- A variable name cannot start with a number.

- Keywords are not allowed to be used as a variable name.
- Blank spaces are not allowed in a variable name.

🎉 Variable Examples

Example 1: Storing a Name

```
name = "John Doe"
print(name)
```

Example 2: Print Your Name and Address

```
name = "John Doe"
address = "California, USA"

print(f"My name is {name}")
print(f"My address is {address}")
```

Example 3: Print Your Full Name From First and Last Name

```
first_name = "Bishworaj"
last_name = "Poudel"
print(f"My full name is {first_name} {last_name}")
```

Example 4: Find Difference Between Income and Expenses

```
income = 70000
expenses = 41000

print(f"Income is Rs. {income}")
print(f"Expenses is Rs. {expenses}")
print(f"Balance is Rs. {income-expenses}")
```

Example 5: Basic Calculation in Python Session

```
num1 = 10
num2 = 3
```

```
sum = num1 + num2
diff = num1 - num2
mul = num1 * num2
div = num1 / num2

# Mod is remainder after division
mod = num1 % num2
intdiv = num1 // num2
cube = num1 ** 3

print(f"Sum is {sum} and diff is {diff}")
print(f"Mul is {mul}")
print(f"Div is {div}")
print(f"Mod is {mod}")
print(f"Int div is {intdiv}")
print(f"Cube is {cube}")
```

💡 Challenge 2: Your Finance Manager

Write down what you spend each day from Sunday to Saturday, add them up to find the total for the week, and figure out the average amount spent per day.

🌟 Data Types in Python

In Python, a data type defines the type of a value that a variable can hold. Python has several built-in data types that allow you to store and manipulate different kinds of data. Understanding these data types is fundamental to writing effective and efficient code.

🚀Common Data Types in Python

Data Type	Description	Example
int	Positive and Negative numbers without decimal parts.	15, -11, 0, 12, etc.

float	Positive and Negative numbers with decimal parts.	12.5, 6.76, 0.0, -7.41, etc.
str	Texts wrapped inside quotation marks are called strings.	“USA”, ‘Raj’, ‘Mountain View CA 390’, etc.
boolean	Represents True or False.	is_married = False, is_voter = True, etc.
list **[]**	A collection of items in a specific order. It is changeable and allows duplicate items.	[1, 2, 3, 4], ["apple", "banana"], [1, "apple", 3.14], etc.
set **{}**	A collection of items that are unordered, unchangeable, and unindexed. No duplicates.	{"apple", "banana", "cherry", "apple"}, {1,20,3}, etc.
tuple **()**	A Collection of items that are unordered, unchangeable, and unindexed. No duplicates.	("apple", "banana", "cherry"), (1,2,50,100), etc.
None	None of the types.	result=None, data=None, etc.

🎉 Real Life Example

Imagine you're managing a small library and need to keep track of the following:

Total number of books	int
Average book rating	float
Name of librarian	str

Wheather library is currently open	bool
List of book names available	list
Location coordinates of the library	tuple
Contact details of the library staff	dict
A set of unique genres available	set

🎉 Example Demo

Here is the Python code demonstrating the above information:

```
# Variables representing library data
total_books = 500  # int
average_rating = 4.5  # float
librarian_name = "John Doe"  # str
is_open = True  # bool
book_names = ["Python Programming", "Introduction to Data Science",
"History of Art"]  # list
location_coordinates = (40.7128, -74.0060)  # tuple (latitude,
longitude)
contact_details = {
    "phone": "123-456-7890",
    "email": "library@example.com",
    "address": "123 Library St, City, State"
}  # dict
unique_genres = {"Fiction", "Non-fiction", "Science Fiction"}  # set
# Printing out the library information
print("Library Information:")
print(f"Total number of books: {total_books}")
print(f"Average book rating: {average_rating}")
print(f"Librarian's name: {librarian_name}")
print(f"Is library open?: {'Yes' if is_open else 'No'}")
print(f"List of book names available: {book_names}")
print(f"Location coordinates: {location_coordinates}")
print(f"Contact details: {contact_details}")
print(f"Unique genres available: {unique_genres}")
```

🎉 Using Type int

Here **number_of_books** represents the total count of books available in a library. It is an integer (int) data type.

```
# Total number of books
number_of_books = 350
print(f"Total number of books: {number_of_books}")
print(f"Type  of number_of_books: {type(number_of_books)}")
```

🎉 Using Type float

Here **temperature** represents the current temperature outside. It is a floating-point number (double) data type, used for values that can have decimal points, such as temperatures, measurements, or calculations involving fractions.

```
#Example: Temperature in Celsius
temperature = 28.5
print(f"Current temperature: $temperature degrees Celsius")
print(f"Type of temperature: ${type(temperature)}")
```

🎉 Using Type str

Here **username** stores the username of a user's social media account. It is a string (str) data type, used for storing textual data like names, addresses, or any sequence of characters.

```
# Example: Username of a social media account
username = "johndoe85"
print(f"Username: {username}")
print(f"Type of username: {type(username)}")
```

```
print("This is String")
print('This is also String')
```

Note: Whether you choose single quotes or double quotes to create strings, the end result remains the same.

🎉 Using Type bool

Here **is_subscribed** indicates whether a user is subscribed to a service (True) or not (False). It is a boolean (bool) data type, used for storing logical values, typically used for conditions or flags.

```
# Example: Subscription status of a service
is_subscribed = True
print(f"Is the user subscribed?: {'Yes' if is_subscribed else 'No'}")
print(f"Type of is_subscribed: {type(is_subscribed)}")
```

🎉 Using Type list

Here **student_grades** contains the grades of students. It is a list (list) data type, used for storing ordered collections of items, allowing easy access and modification.

```
# Example: Grades of students
student_grades = [85, 90, 75, 95, 88]
print(f"Student grades: {student_grades}")
print(f"Type of student_grades: {type(student_grades)}")
```

🎉 Using Type tuple

Here **city_coordinates** stores the latitude and longitude coordinates of a specific city. It is a tuple (tuple) data type, used for fixed collections where the order and number of elements should not change.

```
# Example: Geographic coordinates of a city
city_coordinates = (40.7128, -74.0060)
print(f"City coordinates: {city_coordinates}")
print(f"Type of city_coordinates: {type(city_coordinates)}")
```

🎉 Using Type dict

Here **product_info** contains various details about a product, such as its name, brand, price, and availability. It is a dictionary (dict) data type, used for storing key-value pairs and enabling efficient retrieval and manipulation of data based on keys.

```
# Example: Details of a product
product_info = {
    "name": "Smartphone",
    "brand": "Apple",
    "price": 999.99,
```

```
    "in_stock": True
}
# Print product information
print("Product information:")
print(f"  Name: {product_info['name']}")
print(f"  Brand: {product_info['brand']}")
print(f"  Price: ${product_info['price']}")
print(f"  In Stock: {'Yes' if product_info['in_stock'] else 'No'}")
# Print type of product_info
print(f"Type of product_info: {type(product_info)}")
```

🎉 Using Type set

In this example, **blog_post_tags** represents unique tags associated with a blog post, such as topics or keywords. It is a set (set) data type, used for storing unique and unordered collections of items, ensuring each element is distinct.

```
# Example: Tags associated with a blog post
blog_post_tags = {"Python", "Data Science", "Machine Learning"}
print(f"Tags for the blog post: {blog_post_tags}")
print(f"Type of blog_post_tags: {type(blog_post_tags)}")
```

💡 Challenge 3: Your Finance Manager

Write down what you spend each day from Sunday to Saturday using a dictionary, add them up to find the total for the week, and figure out the average amount spent per day.

🌟 Convert Data Types in Python

When working with Python, you'll often need to convert data from one type to another. This process is known as type conversion or type casting.

Python offers several functions to convert data from one type to another. Here are some of the most commonly used type conversion functions:

Function	Description
int()	Converts data to an integer
float()	Converts data to a floating-point number
str()	Converts data to a string
bool()	Converts data to a boolean
list()	Converts data to a list
tuple()	Converts data to tuple
set()	Converts data to set
dict()	Converts data to dictionary

🚀String Vs Numbers

Consider these two examples: **2024 and "2024"**. At first look, they might look the same. But in Python, they represent two different types of data:

- **2024** is a number, specifically an integer.
- **"2024"** is a string because it is enclosed in double quotes.

Always remember: **if something is inside quotes, it's a string.**

String Examples	Number Examples
"456"	**485**
"0"	**0**
"99.9"	**600.85**
"Hello, Technology Channel!"	**-212**

🎉Example 1: User Input

When you take input from a user, it's always a string by default. You often need to convert this input to the appropriate type for calculations or comparisons.

```
# Taking user input for age and converting it to an integer
user_age = input("Enter your age: ")  # e.g., user enters "25"
age = int(user_age)
print(f"Your age is: {age}")
print(f"Type of age: {type(age)}")  # Output: Your age is: 25, Type of
age: <class 'int'>
```

The **int()** function converts a value to an integer.

🎉Example 2: Finance Manager

Imagine you are managing your finances and want to calculate your total expenses for the week. You have a dictionary where each day is a key, and the value is the amount spent that day as a string. You need to convert these string values to integers or floats to calculate the total and average expenses.

```
# Expenses for each day of the week as strings
expenses = {
    "Sunday": "45.50",
    "Monday": "60.75",
    "Tuesday": "39.90",
    "Wednesday": "75.25",
    "Thursday": "22.10",
    "Friday": "58.40",
    "Saturday": "33.85"
}

# Convert string values to float and calculate total and average
expenses
total_expenses = sum(float(value) for value in expenses.values())
average_expenses = total_expenses / len(expenses)

print(f"Total expenses for the week: ${total_expenses:.2f}")
print(f"Average daily expense: ${average_expenses:.2f}")
```

💡 Challenge 4: Your Age Calculator

Write a program that takes user input for year of birth (as a string) in the format 'YYYY' and finds the age of the person.

🌟 Comments In Python

Comments are text in a program that are ignored by the interpreter or compiler during execution. They are used to explain the code, making it more readable and maintainable.

```
# This is a comment. comment start with # symbol
print("I am proud python developer.")
```

🚀Advantages

- You can describe your code.
- Other people will understand your code more clearly.
- They help in understanding the code's logic and purpose.
- Comments provide context for easier maintenance and updates.
- They facilitate better collaboration by explaining complex logic and decisions.
- For beginners, comments are a valuable tool for learning and understanding programming concepts and structures.

🎉Example 1: Comments

This code performs and prints the result of adding two numbers, **num1** and **num2**. The comments explain that **result_add** is calculated by adding **num1** and **num2**, and the result is printed using an f-string for formatted output.

```
# Define two numbers to perform operations on
num1 = 10
num2 = 5

# Addition
```

```
# Adding num1 and num2
result_add = num1 + num2
# Printing the result of addition
print(f"Addition of {num1} and {num2} is: {result_add}")
```

💡 Challenge 5: Odd or Even Calculator

Write a program that checks if a given number is even or odd. Take user input for the number, perform the check, and print the result. Include comments explaining each part of the code.

🌟 Operators In Python

Operators are used to perform mathematical and logical operations on the variables. Each operation uses a symbol called the operator to denote the type of operation it performs.

Operands: It represents the data.
Operator: It represents how the operands will be processed to produce a value.

```
2 + 3
```

Note: Suppose the given expression is 2 + 3. Here **2** and **3** are operands, and **+** is the operator.

🚀Arithmetic Operators

These are used to perform basic arithmetic operations like addition, subtraction, multiplication, etc.

Symbol	Name	Description
+	Addition	For adding two operands
-	Subtraction	For subtracting two operands

*	Multiplication	For multiplying two operands
/	Division	For dividing two operands
//	Integer Division	Divides and returns the integer value
%	Modulus	Returns the remainder from division
**	Exponentiation	Raises one operand to the power of another

🎉Example 1: Arithmetic Operator

This code performs and prints the results of various arithmetic operations on num1 and num2.

```
# Initialize the variables
num1 = 10
num2 = 3

# Performing Operations
addition = num1 + num2
subtraction = num1 - num2
multiplication = num1 * num2
division = num1 / num2
floor_division = num1 // num2
modulus = num1 % num2
exponentiation = num1 ** num2

# Displaying Results
print(f"Addition: {num1} + {num2} = {addition}")
print(f"Subtraction: {num1} - {num2} = {subtraction}")
print(f"Multiplication: {num1} * {num2} = {multiplication}")
print(f"Division: {num1} / {num2} = {division}")
print(f"Floor Division: {num1} // {num2} = {floor_division}")
print(f"Modulus: {num1} % {num2} = {modulus}")
print(f"Exponentiation: {num1} ** {num2} = {exponentiation}")
```

🚀Comparison Operators

These are used to compare two values.

Symbol	Name	Description
==	Equal	Checks if the value of two operands are equal
!=	Not Equal	Checks if the value of two operands are not equal
>	Greater Than	Checks if the value of the left operand is greater than the right operand
<	Less Than	Checks if the value of the left operand is less than the right operand
>=	Greater or Equal	Checks if the value of the left operand is greater than or equal to the right operand

🎉Example 2: Comparison Operator

This code performs and prints the results of various comparison operations on num1 and num2.

```
# Initialize the variables
num1 = 10
num2 = 3

# Performing Comparison Operations
equal = num1 == num2
not_equal = num1 != num2
greater_than = num1 > num2
less_than = num1 < num2
greater_or_equal = num1 >= num2
less_or_equal = num1 <= num2

# Displaying Results
print(f"Equal: {num1} == {num2} is {equal}")
print(f"Not Equal: {num1} != {num2} is {not_equal}")
print(f"Greater Than: {num1} > {num2} is {greater_than}")
```

```
print(f"Less Than: {num1} < {num2} is {less_than}")
print(f"Greater or Equal: {num1} >= {num2} is {greater_or_equal}")
print(f"Less or Equal: {num1} <= {num2} is {less_or_equal}")
```

🚀Logical Operators

These are used to combine conditional statements.

Symbol	Name	Description
and	AND	Returns True if both statements are true
or	OR	Returns True if one of the statements is true
not	NOT	Reverses the result, returns False if the result is true

🎉Example 3: Logical Operator

This code performs and prints the results of various logical operations (AND, OR, NOT) on boolean expressions involving num1 and num2.

```
# Initialize the variables
num1 = 10
num2 = 3

# Performing Logical Operations
and_operation = (num1 > num2) and (num1 < 20)
or_operation = (num1 < num2) or (num1 < 20)
not_operation = not(num1 > num2)

# Displaying Results
print(f"AND Operation: (num1 > num2) and (num1 < 20) is
{and_operation}")
print(f"OR Operation: (num1 < num2) or (num1 < 20) is {or_operation}")
print(f"NOT Operation: not(num1 > num2) is {not_operation}")
```

🚀Assignment Operators

These are used to assign values to variables.

Symbol	Name	Description
=	Assignment	Assigns the right-hand operand to the left-hand operand
+=	Add and Assignment	Adds right operand to the left operand and assigns the result to the left operand
-=	Subtract and Assignment	Subtracts right operand from the left operand and assigns the result to the left operand
*=	Multiply and Assignment	Multiplies the left operand with the right operand and assigns the result to the left operand
/=	Divide and Assignment	Divides the left operand by the right operand and assigns the result to the left operand

🎉Example 4: Assignment Operators

This code demonstrates various assignment operations on the variable num1 using num2 and prints the results.

```
# Initialize the variables
num1 = 10
num2 = 3

# Performing Assignment Operations
initial_assignment = num1
num1 += num2
add_and_assign = num1
```

```
num1 -= num2
subtract_and_assign = num1

num1 *= num2
multiply_and_assign = num1

num1 /= num2
divide_and_assign = num1

# Displaying Results
print(f"Initial assignment: num1 = {initial_assignment}")
print(f"Add and assign: num1 += num2 -> num1 = {add_and_assign}")
print(f"Subtract and assign: num1 -= num2 -> num1 =
{subtract_and_assign}")
print(f"Multiply and assign: num1 *= num2 -> num1 =
{multiply_and_assign}")
print(f"Divide and assign: num1 /= num2 -> num1 = {divide_and_assign}")
```

🚀Membership Operators

These are used to test if a value is available in a sequence or not.

Symbol	Name	Description
in	In	Returns True if a specified value is found in the sequence
not in	Not In	Returns True if a specified value is not found in the sequence

🎉Example 5: Membership Operators

This code demonstrates the use of membership operators to check for the presence of items in a list.

```
# Initialize the list
fruits = ["apple", "banana", "cherry"]
```

```python
# Performing Membership Operations
is_banana_in_fruits = "banana" in fruits
is_grape_in_fruits = "grape" in fruits
is_apple_not_in_fruits = "apple" not in fruits
is_grape_not_in_fruits = "grape" not in fruits

# Displaying Results
print(f'"banana" in fruits: {is_banana_in_fruits}')
print(f'"grape" in fruits: {is_grape_in_fruits}')
print(f'"apple" not in fruits: {is_apple_not_in_fruits}')
print(f'"grape" not in fruits: {is_grape_not_in_fruits}')
```

💡 Challenge 6: Solve this questions

1. Write a program that takes the ages of two people as input from the user. Compare their ages using comparison operators (==, !=, >, <, >=, <=) and print messages indicating the results (e.g., "Person 1 is older than Person 2").
2. Write a program that takes the age and citizenship status of a person as input from the user. Use logical operators (and, or, not) to determine if the person is eligible to vote (age >= 18 and is a citizen). Print an appropriate message based on the result.
3. Write a program that initializes a list of books available in a library. Take the name of a book as input from the user and check if it is available in the library using membership operators (in, not in). Print a message indicating whether the book is available or not.

🌟 User Input In Python

User input is when you type information into a computer program while it's running. This allows the program to ask you questions and use your answers to do things. For example, a program might ask for your name, your age, or any other information it needs to work.

Note: User input in Python is handled using the input() function.

🎉Example 1: Simple User Input

This code takes one name as input and greets that name.

```
# Asking the user for their name
name = input("What is your name? ")

# Printing the user's name
print("Hello, " + name + "!")
```

🎉Example 2: Calculator Using User Input

This code asks the user to enter 2 numbers and find their sum.

```
# Asking for user input
number1 = int(input("Enter first number: "))
number2 = int(input("Enter second number: "))

# Calculation
total = number1 + number2

# Display the result
print(f"Total is {total}")
```

💡 Challenge 7

Create a program that finds cube numbers using user input.

🌟 String In Python

A string in Python is a sequence of characters. It can include letters, numbers, symbols, and whitespace. Strings are used to represent text data.

Note: You can create a string by enclosing characters in either single quotes **(')** or double quotes **(")**.

```
print("This is String")
print('This is also String')
```

🎉Example 1: Greeting Message

Imagine you are developing a chatbot that greets users.

```
user_name = "John"
greeting = "Hello, " + user_name + "! Welcome to our service."
print(greeting)
```

🚀Common String Method in Python

Method	Description
str.upper()	Converts all characters in the string to uppercase
str.lower()	Converts all characters in the string to lowercase
str.capitalize()	Capitalizes the first character of the string
str.title()	Capitalizes the first character of each word
str.strip()	Removes leading and trailing whitespace
str.replace(a, b)	Replaces all occurrences of substring a with substring b
str.split(sep)	Splits the string into a list using sep as the delimiter

🎉Example 2: String Methods

Here are real life examples of different string methods in Python

```
# Strings
user_input = "hello world"
email = "John.Doe@Example.com"
sentence = "welcome to the party."
book_title = "to kill a mockingbird"
username = "  alice  "
```

```
comment = "This is a bad example."
items = "apple,banana,cherry"

# 1. str.upper()
uppercase_input = user_input.upper()
print(f"Uppercase: {uppercase_input}")

# 2. str.lower()
normalized_email = email.lower()
print(f"Lowercase email: {normalized_email}")

# 3. str.capitalize()
capitalized_sentence = sentence.capitalize()
print(f"Capitalized sentence: {capitalized_sentence}")

# 4. str.title()
formatted_book_title = book_title.title()
print(f"Title Case: {formatted_book_title}")

# 5. str.strip()
cleaned_username = username.strip()
print(f"Stripped username: '{cleaned_username}'")

# 6. str.replace(a, b)
censored_comment = comment.replace("bad", "good")
print(f"Censored comment: {censored_comment}")

# 7. str.split(sep)
split_items = items.split(",")
print(f"Split items: {split_items}")
```

💡 Challenge 8

Write a Python program that takes a user's full name as input and Convert the full name to title case (capitalize the first letter of each word). Also remove trailing and leading space.

🌟 Randomisation In Python

Randomization is a powerful tool in programming, enabling the creation of random values. This code generates a random float between 0.0 and 1.0.

```
import random
print(random.random())
```

🎉Example 1: Generate Random Number Between 1 to 10

The following code generates random numbers between 1 and 10. You can change the value according to your need.

```
import random

randomnum = random.randint(1, 10)
print(randomnum)
```

🎉Example 2: Random Value From List

Shuffle function shuffles the sequence in place. Here is a code:

```
import random

cards = ['ace', 'king', 'queen', 'jack']
random.shuffle(cards)
print(cards)
```

💡 Challenge 9

Write a number guess game which generates numbers from 1 to 6.

🌟 PIP In Python

PIP is a tool that helps you add new features to Python by downloading and installing additional libraries (packages) from the internet. Think of it like an app store for Python libraries.

You can view all the packages from https://pypi.org/

🎉Example 1: Generate QR Code Using Python

First you need to install the qrcode package. For this course I am using https://pypi.org/project/qrcode/. Go to your terminal and type the following code:

```
pip install qrcode
pip install pillow
```

Here is code to generate QR Code:

```
import qrcode
img = qrcode.make('Bishworaj Poudel')
type(img)  # qrcode.image.pil.PilImage
img.save("bishworaj.png")
```

Run this program and you will see the qr code image.

💡 Challenge 10

Create a QR code that stores URLs. https://technologychannel.org/

🌟 Condition In Python

Conditions allow you to execute certain pieces of code based on whether a condition is true or false. In python this is often done using **if, elif,** and **else** statements. Here is the rule on how to write conditions:

```
if condition:
    # code to execute if condition is true
elif another_condition:
    # code to execute if another_condition is true
else:
    # code to execute if none of the above conditions are true
```

🎉Example 1: Checking Temperature

This program finds what you should wear based on the temperature.

```
if temperature > 25:
    print("It's hot outside! Wear shorts and a t-shirt.")
elif 15 <= temperature <= 25:
    print("It's a pleasant day! Wear jeans and a t-shirt.")
else:
    print("It's cold outside! Wear a jacket.")
```

🎉Example 2: Checking Age for Movie Tickets

This program checks if a person is eligible for a child, adult, or senior movie ticket.

```
age = 70

if age < 13:
    print("You are eligible for a child ticket.")
elif 13 <= age <= 59:
    print("You are eligible for an adult ticket.")
else:
    print("You are eligible for a senior ticket.")
```

🎉Example 3: Grade Evaluation

This program assigns a grade based on a student's score

```
score = 85

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
```

```
    grade = 'D'
else:
    grade = 'F'

print(f"The student's grade is: {grade}")
```

💡 Points To Remember

- Use **`if`** to start a conditional statement.
- Use **`elif`** (short for "else if") to add additional conditions.
- Use **`else`** to specify what to do if none of the previous conditions are met.

💡 Challenge 11

Write a program that checks if a number is even or odd.

🌟 Assert In Python

The assert keyword checks if something is true, and if not, it stops the program and shows an error message. Imagine you are building an online store and you want to make sure the price of an item is never negative. You can use assert to check this:

```
price = -5
assert price >= 0, "Price cannot be negative"
print("After Price Negative")
```

If the price is negative, the program will stop and show the message "Price cannot be negative."

💡 Challenge 12

Write a program that uses assert to check if a person's age is positive.

🌟 Loop In Python

In Programming, loops are used to repeat a block of code until certain conditions are not completed. For, e.g., if you want to print your name 100 times, then rather than typing **print("your name")** 100 times, you can use a loop.

🎉 Most Used Loop In Python

- For Loop
- While Loop

🌟 For Loop In Python

A for loop is used to repeat actions for each item in a sequence (like a list, string, or range).

Syntax:

```
for element in sequence:
    # Code to be executed
```

🎉Example 1: Display Your Name 100 Times

Let's say you want to print your name 100 times. This can be easily done using a for loop in Python:

```
for i in range(100):
    print("Bishworaj Poudel")
```

🎉Example 2: Display 1 To 100

If you want to display numbers from 1 to 100, you can use either a for loop in Python:

```
for i in range(1, 101):
    print(i)
```

🎉Example 3: Display Even Number Between 1 To 100

To display even numbers between 1 and 100 using a for loop, you can follow these steps:

```
for i in range(1, 101):
    if i % 2 == 0:
        print(i)
```

🎉Example 4: Looping Over a List of Names

Suppose you have a list of names and want to print each name. Here's how you can do it using a for loop:

```
names = ["Alice", "Bob", "Sadikshya", "David"]
for name in names:
    print(name)
```

🎉Example 5: Calculating the Sum of Numbers

Consider a scenario where you need to calculate the sum of numbers in a list:

```
numbers = [1, 2, 3, 4, 5]
total = 0
for number in numbers:
    total += number
print("Total:", total)
```

🌟 While Loop In Python

The while loop continues to execute a block of code as long as a specified condition is true.

Syntax:

```
while condition:
    # Code to be executed
```

🎉Example 6: Countdown Timer

A practical example of a while loop is a countdown timer that starts from 10 and counts down to 1:

```
countdown = 10
while countdown > 0:
    print(countdown)
    countdown -= 1
print("Countdown complete!")
```

🎉Example 7: Validating User Input

A while loop can also be used to validate user input. For instance, you may want to keep asking for a valid password until the user provides one:

```
password = ""
while password != "python123":
    password = input("Enter the password: ")
print("Access granted!")
```

💡 Challenge 13

Write a program that finds the total and average of the following list:

```
expenses = [889,788,5656,4455,455,45]
```

🌟Break and Continue In Python

In Python, the break and continue statements help control the flow of loops. These simple yet powerful commands allow you to manage how loops behave in a more flexible way.

🎉 The break Statement

The break statement is used to exit a loop immediately. Imagine you're searching for a specific book in a library. Once you find it, you stop searching. This is exactly what the break statement does in a loop.

```
break
```

🎉 The continue Statement

The continue statement skips the current iteration and moves to the next one. Think of it as sorting through a basket of fruits and skipping over any rotten ones.

```
continue
```

🎉 Example 1: Display Numbers from 1 to 100, Skipping 10

If you want to display numbers from 1 to 100 but skip 10, you can use a for loop with a continue statement in Python:

```
for i in range(1, 101):
    if i == 10:
        continue
    print(i)
```

🎉 Example 2: Display Numbers from 1 to 100, Break At 10

If you want to display numbers from 1 to 100 but stop the loop entirely when it reaches 10, you can use a for loop with a break statement in Python:

```
for i in range(1, 101):
    if i == 10:
        break
    print(i)
```

🎉Example 3: Finding a Specific Book

If you want to find a specific book in a list and stop searching once you've found it, you can use a for loop with a break statement in Python:

```
books = ["Book A", "Book B", "Book C", "Book D"]
for book in books:
    if book == "Book C":
        print("Found the book! Stopping search.")
```

```
        break
    print(f"Checking {book}")

print("Search ended.")
```

💡 Challenge 14

Write a program that calculates the total and average of the following list of numbers, stopping if a negative value is encountered (using break), and skipping any zero values (using continue).

```
values = [10, 0, 25, 0, 50, -1, 40]
```

🌟List in Python

If you want to store multiple values in the same variable, you can use List. **E.g.** to store the names of multiple items, you can use a List. The List is represented by Square Braces**[].**

```
grocery_list = ["apples", "bananas", "milk", "bread"]
```

Here, we have created a list named grocery_list. It contains 4 items.

🎉 List Features

- **Ordered:** The items stay in the order you list them. If you write "milk" before "bread," that order is remembered.
- **Changeable:** You can update your list. If you decide you want "orange juice" instead of "milk," you can change it.
- **Mixed Items:** You can mix different things in a list, like numbers and words.

🎉 Using Lists in Everyday Life

Create a List. Just like making a to-do list.

```
to_do_list = ["finish homework", "buy groceries", "go for a walk", "read a book"]
```

Access List Items: Want to see what's first on your list?

```
first_task = to_do_list[0]  # "finish homework"
```

Change Item of List: If you decide to "call a friend" instead of "read a book":

```
to_do_list[3] = "call a friend"
```

Add Item to List: If you remember you need to "water the plants":

```
to_do_list.append("water the plants")
```

Delete Item from List: Once you've bought groceries, you can remove that task:

```
to_do_list.remove("water the plants")
```

Count the Items: To see how many tasks you have left:

```
tasks_left = len(to_do_list)
```

🎉 Example 1: Planning a Party

Imagine you're planning a party and need to keep track of tasks:

```
# Step 1: Create a list of tasks
party_tasks = ["buy decorations", "send invitations", "order food"]

# Step 2: Add a new task to the list
party_tasks.append("set up music")
print("Updated Party Tasks:", party_tasks)

# Step 3: Check the first task you need to do
```

```
first_task = party_tasks[0]
print("First task to do:", first_task)

# Step 4: Mark the task as done by removing it from the list
party_tasks.remove("send invitations")
print("Remaining Tasks:", party_tasks)

# Step 5: Add another task you remembered
party_tasks.append("clean the house")
print("Final Task List:", party_tasks)

# Step 6: Count how many tasks are left
tasks_left = len(party_tasks)
print("Number of tasks left to do:", tasks_left)
```

🎉Example 2: Monthly Expense Tracker

Imagine you want to keep track of your expenses for the month.

```
# Step 1: Create a list of expenses
monthly_expenses = [150, 200, 50, 75, 100]

# Step 2: Add a new expense to the list
monthly_expenses.append(60)
print("Updated Expenses:", monthly_expenses)

# Step 3: Calculate the total amount spent
total_expense = sum(monthly_expenses)
print("Total Expense:", total_expense)

# Step 4: Remove an unnecessary expense
monthly_expenses.remove(75)
print("Expenses after removing utilities:", monthly_expenses)

# Step 5: Calculate the new total after removing an expense
```

```
total_expense = sum(monthly_expenses)
print("New Total Expense:", total_expense)

# Step 6: Find the highest expense
highest_expense = max(monthly_expenses)
print("Highest Expense:", highest_expense)

# Step 7: Find the lowest expense
lowest_expense = min(monthly_expenses)
print("Lowest Expense:", lowest_expense)

# Step 8: Calculate the average expense
average_expense = sum(monthly_expenses) / len(monthly_expenses)
print("Average Expense:", average_expense)
```

🎉 Example 3: Reverse a List

You can use the reverse method to reverse a list.

```
my_list = [1, 2, 3, 4, 5]
my_list.reverse()
print(my_list)
```

🎉 Example 4: Display Last Item of List

You can use negative indexing to display the last item from the list. -1 for last, -2 for second last and so on.

```
my_list = [1, 2, 3, 4, 5]
last_item = my_list[-1]
print(last_item)  # Output: 5
```

💡 Challenge 15

1. Create an empty list of type strings called days. Use the add method to add names of 7 days and print all days.
2. Add your 7 friend names to the list. Use where to find a name that starts with the alphabet a.

🌟Tuple in Python

If you want to store multiple values in the same variable but don't want them to change, you can use a Tuple. For example, to store a collection of fixed values, you can use a Tuple. The Tuple is represented by parentheses ().

```
coordinate = (10.0, 20.0)
```

Here, we have created a tuple named coordinate. It contains 2 items, representing the X and Y coordinates.

🎉 Tuple Features

- **Ordered:** Just like lists, the items in a tuple maintain the order you put them in.
- **Unchangeable:** Once you create a tuple, you cannot change it. This makes tuples useful for storing values that should not be modified.
- **Mixed Items:** You can store different types of data, like numbers and strings, together in a tuple.

🎉 Using Tuples in Everyday Life

Create a Tuple: Just like a list, but use parentheses instead of square brackets.

```
colors = ("red", "green", "blue")
```

Access Tuple Items: Want to see what's first in your tuple?

```
first_color = colors[0]  # "red"
```

Unchangeable Nature: If you try to change an item in a tuple, Python will raise an error:

```
colors[0] = "yellow"  # Error
```

Tuple with Mixed Items: You can mix numbers, strings, and other types.

```
person = ("Alice", 30, "Engineer")
```

🎉 Example 1: Storing Days of the Week

Imagine you want to store the days of the week in a tuple, which shouldn't change.

```
# Step 1: Create a tuple for the days of the week
days_of_week = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday")

# Step 2: Access the first and last days of the week
first_day = days_of_week[0]
last_day = days_of_week[-1]
print("First Day of the Week:", first_day)
print("Last Day of the Week:", last_day)
```

🎉Example 2: Combining Tuples

You can combine two or more tuples using the + operator.

```
# Step 1: Create two tuples
odd_numbers = (1, 3, 5)
even_numbers = (2, 4, 6)

# Step 2: Combine the tuples
all_numbers = odd_numbers + even_numbers
print(f"Combined Tuple: {all_numbers}")
```

🎉 Example 3: Tuple Length

You can find out how many items are in a tuple using the **len()** function.

```
# Step 1: Create a tuple
fruit_tuple = ("apple", "banana", "cherry")

# Step 2: Get the length of the tuple
num_fruits = len(fruit_tuple)
print(f"Number of Fruits: {num_fruits}")
```

💡 Challenge 16

1. Create a tuple of your favorite fruits and display all the fruits.
2. Access and print the first and last fruit in the tuple.
3. Attempt to change the second fruit in the tuple and observe what happens (this should raise an error since tuples are unchangeable).
4. Combine this tuple with another tuple of your favorite vegetables, and print the resulting combined tuple.
5. Find and print the length of the combined tuple using the len() function.

🌟Set In Python

If you want to store multiple unique values in the same variable without worrying about the order or duplicates, you can use a **Set**. For example, to store a collection of unique items, you can use a Set. The Set is represented by curly braces `{}`.

```
fruits = {"apple", "banana", "cherry"}
```

Here, we have created a set named fruits. It contains 3 items.

🎉 Set Features

- **Unordered**: The items in a set do not have a defined order. Even if you print the set multiple times, the order of items might change.
- **Unchangeable Items**: Once you add an item to a set, you cannot change it, but you can add or remove entire items.

- **Unique Items**: Sets do not allow duplicate items. If you try to add an item that already exists in the set, it will be ignored.

🎉 Using Sets in Everyday Life

Create a Set: Just like making a list, but use curly braces instead of square brackets.

```
fruits = {"apple", "banana", "cherry"}
```

Check for Item in Set: Want to see if an item exists in your set?

```
is_in_set = "banana" in fruits  # True
```

Add Item to Set: You can add a new item to the set.

```
fruits.add("orange")
```

Remove Item from Set: You can remove an item from the set.

```
fruits.remove("banana")
```

Set with Mixed Items: You can store different types of data together in a set.

```
mixed_set = {1, "apple", 3.14}
```

🎉 Example 1: Unique Email Addresses

Imagine you're organizing a workshop and people are registering multiple times using the same email address. You want to keep only unique email addresses.

```
# List of registered emails, with some duplicates
emails = ["alice@example.com", "bob@example.com", "alice@example.com",
"bob@example.com"]

# Convert the list to a set to keep only unique email addresses
unique_emails = set(emails)
print("Unique Emails:", unique_emails)
```

🎉 Example 2: Finding Common Elements (Intersection)

Let's say you have two lists of students who enrolled in different courses, and you want to find out which students are enrolled in both courses.

```
# Courses in sets

course_A_students = {"Alice", "Bob", "Charlie"}
course_B_students = {"Charlie", "David", "Alice"}

common_students = course_A_students.intersection(course_B_students)
print(common_students)
```

🎉 Example 3: Union, Difference, and Symmetric Difference

Suppose you are managing two different projects, and you want to know which tasks are shared between them, which are unique to each project, and which tasks are in either project.

```
project_A_tasks = {"task1", "task2", "task3"}
project_B_tasks = {"task2", "task4", "task5"}

# Union: All tasks in either project
all_tasks = project_A_tasks.union(project_B_tasks)
print("Union:", all_tasks)

# Difference: Tasks in Project A but not in Project B
diff_tasks = project_A_tasks.difference(project_B_tasks)
print("Difference:", diff_tasks)

# Symmetric Difference: Tasks in either project, but not in both
sym_diff_tasks = project_A_tasks.symmetric_difference(project_B_tasks)
print("Symmetric Difference:", sym_diff_tasks)
```

💡 Challenge 16

Create a set of your favorite hobbies. Add a few more hobbies to the set, then remove one. Finally, perform a union with another set of hobbies and find the common ones using intersection.

🌟Dictionary In Python

If you want to store data in key-value pairs, you can use a **Dictionary**. For example, to store a collection of related data like a person's name and age, you can use a Dictionary. The Dictionary is represented by curly braces `{}` with keys and values separated by a colon `:`.

```
person = {"name": "Alice", "age": 25, "city": "New York"}
```

Here, we have created a dictionary named person. It contains three key-value pairs: **"name": "Alice", "age": 25, and "city": "New York"**.

🎉 Dictionary Features

- **Unordered**: The items in a dictionary are stored without any specific order. The order of items may vary.
- **Changeable**: You can change, add, or remove key-value pairs in a dictionary after it is created.
- **Key-Value Pairs**: Each item in a dictionary consists of a key and a corresponding value.

🎉 Using Dictionaries in Everyday Life

Create a Dictionary: Just like making a list, but each item is a key-value pair.

```
car = {"brand": "Toyota", "model": "Corolla", "year": 2024}
```

Access a Value by Key: Want to see the value associated with a specific key?

```
car_model = car["model"]  # "Corolla"
```

Change a Value: If you want to update a value, simply use the key.

```
car["year"] = 2025
```

Add a New Key-Value Pair: You can easily add a new key-value pair to the dictionary.

```
car["color"] = "blue"
```

Remove a Key-Value Pair: You can remove a key-value pair from the dictionary.

```
car.pop("model")
```

🎉 Example 1: Storing Contact Information

Imagine you want to store the contact information of a person.

```
# Step 1: Create a dictionary with contact details
contact = {"name": "Bob", "phone": "555-1234", "email":
"bob@example.com"}

# Step 2: Access the phone number
phone_number = contact["phone"]
print("Phone Number:", phone_number)

# Step 3: Update the email address
contact["email"] = "bob.new@example.com"
print("Updated Contact:", contact)

# Step 4: Add an address
contact["address"] = "123 Main St"
print("Contact with Address:", contact)

# Step 5: Remove the phone number
contact.pop("phone")
print("Contact after Removing Phone:", contact)
```

🎉Example 2: Language Translation

Imagine you want to create a simple dictionary to translate words from English to Spanish.

```
# Step 1: Create a dictionary for translation
translation = {"hello": "hola", "goodbye": "adiós", "please": "por
favor", "thank you": "gracias"}

# Step 2: Translate "hello" to Spanish
spanish_hello = translation["hello"]
print("Hello in Spanish:", spanish_hello)
```

🎉Example 3: Check if Key Available

Imagine you want to store the squares of certain numbers and check if specific numbers are present in the dictionary.

```
# Step 1: Create a dictionary with numbers and their squares
squares = {2: 4, 4: 16, 6: 36, 8: 64, 10: 100}

# Step 2: Check if the number 2 is in the dictionary
is_two_present = 2 in squares
print("Is 2 in the dictionary?", is_two_present)  # True

# Step 3: Check if the number 4 is in the dictionary
is_four_present = 4 in squares
print("Is 4 in the dictionary?", is_four_present)  # True

# Step 4: Check if the square 64 is in the dictionary (as a key)
is_sixty_four_key = 64 in squares
print("Is 64 a key in the dictionary?", is_sixty_four_key)  # False
```

💡 Challenge 16

Create a dictionary of your favorite movies and their release years. Add a new movie to the dictionary, update the release year of one of the movies, and then remove a movie.

🌟List Comprehension in Python

List comprehension means a simple way to create a list.

Syntax

```
[expression for item in iterable if condition]
```

Expression: This is the value that will be added to the list.
Item: Each element in the iterable.
Iterable: A collection you are looping over (like a list, range, etc.).
Condition (optional): A filter that only includes elements that satisfy the condition.

🎉 Example 1: Generating a List of Squares

Imagine you want to generate a list of squares for numbers from 1 to 5.

```
# Step 1: Use list comprehension to generate squares
squares = [x**2 for x in range(1, 6)]
print("List of Squares:", squares)
```

Explanation

- The list comprehension `[x**2 for x in range(1, 6)]` generates a list of squares for each number in the range from 1 to 5.
- The resulting list is `[1, 4, 9, 16, 25]`.

🎉Example 2: Filtering Out High-Scoring Students

Imagine you have a list of student scores, and you want to create a new list containing only the scores that are above 80.

```
# Step 1: List of student scores
scores = [65, 85, 90, 78, 88, 92, 56]

# Step 2: Filter out scores above 80 using list comprehension
high_scores = [score for score in scores if score > 80]
print("High Scores:", high_scores)
```

🎉Example 3: Filtering Words by Length

Imagine you have a list of words, and you want to create a new list containing only words that have exactly 5 letters.

```
# Step 1: List of words
words = ["apple", "banana", "grape", "kiwi", "peach", "plum"]

# Step 2: Filter words with exactly 5 letters using list comprehension
five_letter_words = [word for word in words if len(word) == 5]
print("Five Letter Words:", five_letter_words)
```

🌟 Dictionary Comprehension in Python

Dictionary comprehension means a simple way to create a dictionary.

Syntax

```
{key_expression: value_expression for item in iterable if condition}
```

Key_expression: The expression that defines the key in the dictionary.
Value_expression: The expression that defines the value in the dictionary.
Item: Each element in the iterable.
Iterable: A collection you are looping over.
Condition (optional): A filter that only includes elements that satisfy the condition.

🎉 Example 4: Generating a Dictionary of Squares

Imagine you want to create a dictionary where the keys are numbers from 1 to 5, and the values are their squares.

```
# Step 1: Use dictionary comprehension to generate squares
squares_dict = {x: x**2 for x in range(1, 6)}
print("Dictionary of Squares:", squares_dict)
```

Explanation

- The dictionary comprehension `{x: x**2 for x in range(1, 6)}` creates a dictionary where each number is a key, and its square is the value.
- The resulting dictionary is `{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}`.

🎉 Example 5: Filtering a Dictionary by Value

Imagine you have a dictionary of student names and their scores, and you want to create a new dictionary containing only the students who scored above 80.

```
# Step 1: Dictionary of student scores
student_scores = {"Alice": 85, "Bob": 76, "Charlie": 90, "David": 65,
"Eve": 88}

# Step 2: Filter out students who scored above 80 using dictionary
comprehension
high_scorers = {name: score for name, score in student_scores.items() if
score > 80}
print("High Scorers:", high_scorers)
```

💡 Challenge 19

1. Write a program that takes a list of numbers and creates a new list containing the squares of all even numbers from the original list.
2. Write a program that takes a dictionary of student names and their grades, and creates a new dictionary containing only the students who passed the exam (grade 50 or above).

🌟ATM Project

This program simulates a simple ATM where a user can check their balance, deposit money, or withdraw money. The user should be able to:

- Check their current balance.
- Deposit money into their account.
- Withdraw money from their account.

```
balance = 1000.0  # Initial balance

print("Welcome to Simple Bank ATM!")

while True:
    print("\nPlease select an option:")
    print("1. Check Balance")
    print("2. Deposit Money")
    print("3. Withdraw Money")
    print("4. Exit")

    choice = int(input("Enter your choice (1-4): "))

    if choice == 1:
        print(f"Your current balance is: ${balance:.2f}")

    elif choice == 2:
        deposit_amount = float(input("Enter amount to deposit: $"))
        if deposit_amount > 0:
            balance += deposit_amount
            print(f"Successfully deposited ${deposit_amount:.2f}. Your
new balance is ${balance:.2f}.")
        else:
            print("Deposit amount must be positive.")

    elif choice == 3:
            withdrawal_amount = float(input("Enter amount to withdraw:
```

```
$"))
            if withdrawal_amount > balance:
                print("Insufficient funds. Please try a smaller
amount.")
            elif withdrawal_amount <= 0:
                print("Withdrawal amount must be positive.")
            else:
                balance -= withdrawal_amount
                print(f"Successfully withdrew ${withdrawal_amount:.2f}.
Your new balance is ${balance:.2f}.")

    elif choice == 4:
            print("Thank you for using Simple Bank ATM. Goodbye!")
            break

    else:
            print("Invalid choice. Please try again.")
```

🌟Function In Python

Functions are the block of code that performs a specific task. They are created when some statements are repeatedly occurring in the program. The function helps reusability of the code in the program. The main objective of the function is DRY(**Don't Repeat Yourself**).

Function Advantages

- Avoid code repetition
- Easy to divide the complex program into smaller parts
- Helps to write a clean code

Function takes input (called parameters), processes it, and returns an output.

🎉 Example 1: Greet User Using Function

Imagine you want to greet 3 people using a single function.

```
# Function that greet the name
def greet(name):
    print(f"Hello, {name}!")

# Calling function
greet("Hari")
greet("Mark")
greet("John")
```

Explanation

- Here, `greet` is the function name, and `name` is the parameter.
- `greet("Hari")` is calling a function. Once you've defined a function, you can call it to use it.

🌟Function Types

- No Parameter And No Return Type
- Parameter And No Return Type
- No Parameter And Return Type
- Parameter And Return Type

🌟1. No Parameter and No Return Type

In this type of function, no parameters are passed to the function, and the function does not return any value. It simply performs an action. Here is an example:

```
# Function that greet the name
def greet():
    print(f"Hello, How are you!")

# Calling function
greet()
```

- In this example, `greet()` is a function that doesn't take any parameters. When you call `greet()`, it simply prints the message `"Hello, How are you!"` to the screen.
- Notice that when calling `greet()`, no arguments are passed because the function does not expect any.

🌟2. Parameter and No Return Type

In this type of function, parameters are passed to the function, but the function does not return any value. It performs an action using the provided parameters. Here is an example:

```
# Function that greets a user with their name
def greet(name):
    print(f"Hello, {name}! How are you?")

# Calling the function with a parameter
greet("Hari")
greet("Mark")
greet("John")
```

- In this example, `greet(name)` is a function that takes one parameter, `name`.
- When you call `greet("Hari")`, the function prints `"Hello, Hari! How are you?"`.
- The function uses the value of the `name` parameter to perform its action, but it does not return any value.

🌟3. No Parameter and Return Type

In this type of function, no parameters are passed to the function, but the function returns a value after performing its task. Here is an example:

```
# Function that returns a greeting message
def get_greeting():
    return "Hello, How are you!"

# Calling the function and storing the return value
```

```
greeting_message = get_greeting()

# Printing the returned value
print(greeting_message)
```

- In this example, `get_greeting()` is a function that does not take any parameters.
- The function performs its task and returns the string `"Hello, How are you!"`.
- When you call `get_greeting()`, it returns the greeting message, which is then stored in the `greeting_message` variable.
- You can then print or use the returned value as needed.

🌟4. Parameter and Return Type

In this type of function, parameters are passed to the function, and the function returns a value after performing its task. Here is an example:

```
# Function that takes two numbers and returns their sum
def add_numbers(a, b):
    return a + b

# Calling the function and storing the return value
sum_result = add_numbers(5, 10)

# Printing the returned value
print(f"The sum is: {sum_result}")
```

- In this example, `add_numbers(a, b)` is a function that takes two parameters, `a` and `b`.
- The function calculates the sum of `a` and `b` and returns the result.
- When you call `add_numbers(5, 10)`, the function returns the sum, which is then stored in the `sum_result` variable.
- You can then print or use the returned value as needed.

🌟Useful Math Function

- math.sqrt(number)
- math.pow(number)
- math.factorial(number)
- math.floor(number)
- math.ceil(number)

```python
import math

# Given number
number = 16

# Calculate square root
square_root = math.sqrt(number)
print(f"The square root of {number} is: {square_root}")

# Calculate power (number^2)
power = math.pow(number, 2)
print(f"{number} to the power of 2 is: {power}")

# Calculate factorial (5!)
factorial = math.factorial(5)
print(f"The factorial of 5 is: {factorial}")

# Given float number
float_number = 16.75

# Calculate floor and ceiling
floored_value = math.floor(float_number)
print(f"The floor of {float_number} is: {floored_value}")

ceiled_value = math.ceil(float_number)
print(f"The ceiling of {float_number} is: {ceiled_value}")
```

💡 Challenge 21

1. Create a function that finds a cube of a number.
2. Create a function that finds age from birth year.

🌟OOP In Python

Object-oriented programming (OOP) is a popular technique to solve programming problems by creating objects. It is one of the most popular programming paradigms and is used in many programming languages, such as Python, Java, C++, etc..

🎇OOP Advantages

- It is easy to understand and use.
- It increases reusability and decreases complexity.
- The productivity of programmers increases.
- It makes the code easier to maintain, modify and debug.
- It promotes teamwork and collaboration.
- It reduces the repetition of code.

In OOP first you need to create a class and then an object.

🌟Class and Object In Python

In object-oriented programming, a class is a blueprint for creating objects. **A class defines the properties and methods** that an object will have. For example, a class called **Dog** might have properties like **breed, color** and methods like **bark, run.**

Class Syntax:

```
# create a class
class Classname:
    pass
```

As mentioned earlier, you need to create a class first before you can create objects from it. Each class has attributes and methods.

🎉 Example 1: Student Class

In this example, the **Student** class allows you to create student objects with specific details and provides methods to retrieve their information and check if they are passing their course.

```
class Student:

    def __init__(self, name, age, grade):
        # Attribute: Student's name, age, and grade
        self.name = name
        self.age = age
        self.grade = grade

    def get_details(self):
        return f"Name: {self.name}, Age: {self.age}, Grade:
{self.grade}"

    def is_passing(self):
        # Method: Checks if the student is passing
        return self.grade >= 60
```

You always need to use `self` as the first argument in a class method. It represents the object that calls the method.

🌟Object

An object in programming is like a real-world thing or entity that you can work with in your code. For example, a student object might have attributes like **name, age, or grade**. It might have methods like checking if a student is passing which are the actions the object can perform.

🎉 Example 2: Student Object

In this example, you'll see how to create and interact with Student objects. After defining the Student class with attributes like name, age, and grade, and methods such as get_details() and is_passing(), you can create your own student objects.

```
class Student:

    def __init__(self, name, age, grade):
        # Attribute: Student's name, age, and grade
        self.name = name
        self.age = age
        self.grade = grade

    def get_details(self):
        return f"Name: {self.name}, Age: {self.age}, Grade:
{self.grade}"

    def is_passing(self):
        # Method: Checks if the student is passing
        return self.grade >= 60


# Create 2 Student object
st1 = Student("Tom", 20, 75)
st2 = Student("Mark", 21, 45)

# Access the student's details
print(st1.get_details())
print(st2.get_details())

# Check if the student is passing
print(f"Student 1 is passed: {st1.is_passing()}")
print(f"Student 2 is passed: {st2.is_passing()}")
```

By creating these objects, you can easily manage and work with individual student data. This example shows how you can access the details of each student and determine if they are passing their course.

💡 Challenge 22

Write a Python program to create a class `Laptop` with the properties `id`, `name`, and `ram`. Then, create three objects of this class and print all their details.

🌟 Inheritance In Python

Inheritance is one of the core concepts of object-oriented programming (OOP) in Python. It allows a class to inherit attributes and methods from another class, promoting code reusability and making it easier to maintain and extend.

🎇Inheritance Advantages

- **Code Reusability**: Inheritance allows you to reuse existing code without rewriting it.
- **Improved Maintainability**: Changes made to a base class automatically reflect in all derived classes.
- **Logical Representation**: Inheritance models real-world relationships, making the code more logical and easier to understand.
- **Extensibility**: You can add new features to a class without modifying its existing code.

🌟 Base Class and Derived Class

In Python, the class that is being inherited from is called the base class (or parent class), and the class that inherits from the base class is called the derived class (or child class).

Syntax

```
# Base class
class ParentClass:
    def method_in_parent(self):
```

```
        print("This is a method in the parent class")

# Derived class
class ChildClass(ParentClass):
    def method_in_child(self):
        print("This is a method in the child class")

# Example usage
child = ChildClass()
child.method_in_parent()  # Inherited from ParentClass
child.method_in_child()   # Defined in ChildClass
```

🎉Example 1: Basic Inheritance

In this example, the **Student** class inherits the **name** and **age** attributes, as well as the **get_details** method from the Person class. The Student class also introduces a new attribute grade and a method **is_passing** to check if the student is passing.

```
# Base class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def get_details(self):
        return f"Name: {self.name}, Age: {self.age}"

# Derived class
class Student(Person):
    def __init__(self, name, age, grade):
        # Call the constructor of the base class
        super().__init__(name, age)
        self.grade = grade

    def is_passing(self):
```

```
        return self.grade >= 60
```

Now, let's create objects using the **Student** class and see how inheritance works.

```
# Create a Student object
student = Student("Alice", 22, 85)

# Access inherited and new attributes and methods
print(student.get_details())  # Inherited method from Person
print(f"Is the student passing? {student.is_passing()}")  # Method from
Student
```

🌟 Method Overriding

Inheritance also allows method overriding, where a method in a derived class has the same name as a method in the base class. The derived class's method overrides the base class's method.

🎉 Example 2: Method Overriding

In this example, both Dog and Cat classes override the sound method from the Animal class. When you call sound on a Dog or Cat object, it returns the specific sound of that animal instead of the generic sound from the base class.

```
# Base class
class Animal:
    def sound(self):
        return "Some generic sound"

# Derived class
class Dog(Animal):
    def sound(self):
        return "Bark"
```

```
# Derived class
class Cat(Animal):
    def sound(self):
        return "Meow"
```

Let's see how method overriding works with objects of Dog and Cat classes.

```
# Create Dog and Cat objects
dog = Dog()
cat = Cat()

# Call the overridden method
print(dog.sound())   # Outputs: Bark
print(cat.sound())   # Outputs: Meow
```

💡 Challenge 23

Write a Python program to create a class `Vehicle` with properties `make` and `model`. Then, create a derived class `Car` with an additional property `year`. Create objects of both classes and print all their details.

🌟 Simple Bank Software

Here is a program using **OOP** to create a basic bank software. It includes a BankAccount class with methods to withdraw, deposit, and check the balance.

```
class BankAccount:
    def __init__(self, name, phone, balance=0.0):
        self.name = name
        self.phone = phone
        self.balance = balance
```

```
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited ${amount:.2f} to {self.name}'s account.")
        else:
            print("Invalid deposit amount.")

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew ${amount:.2f} from {self.name}'s account.")
        else:
            print("Invalid withdrawal amount or insufficient balance.")

    def get_balance(self):
        return f"{self.name}'s current balance: ${self.balance:.2f}"

# Create three bank account objects
account1 = BankAccount("Alice Smith", "123-456-7890", 500.0)
account2 = BankAccount("Bob Johnson", "987-654-3210", 1000.0)

# Demonstrate functionality
account1.deposit(200)
print(account1.get_balance())
account1.withdraw(100)
print(account1.get_balance())

account2.deposit(300)
print(account2.get_balance())
account2.withdraw(500)
print(account2.get_balance())
```

🌟 GUI In Python

A **Graphical User Interface (GUI)** is a type of user interface that allows users to interact with a software application visually. In a GUI, elements such as buttons, text fields, images, and windows provide intuitive ways for users to interact with the application. Instead of typing commands, users can simply click, drag, and drop using a mouse or touch input. Here are some examples of GUI Elements:

- **Buttons**
- **Labels**
- **Text Fields**

Python offers several libraries for building GUIs, allowing developers to create desktop applications with interactive components. Here are popular libraries for building GUI In Python.

- **Tkinter [What we will study]**
- **PyQt**
- **Kivy**
- **wxPython**

Learning Tkinter is a great way to get hands-on with GUI applications in Python. Let's learn it.

🎉 Example 1: Simple GUI Application

This code creates a simple window using the tkinter library. It displays a message "Hello, World!" in the center of the window, which stays open and responsive until manually closed.

```
# Import tkinter
import tkinter as tk
# Create window
root = tk.Tk()
# Set window title
root.title("Hello World GUI")
# Set window size
root.geometry("300x200")
# Create label with text
label = tk.Label(root, text="Hello, World!")
# Add label to window
label.pack()
# Start the GUI loop
```

```
root.mainloop()
```

🎉 Example 2: Interactive GUI Application

This code creates an interactive window using tkinter. It shows a message and a button, and when the button is clicked, the label changes to "Hello, Welcome to Tkinter!" to demonstrate interaction.

```
import tkinter as tk

def greet():
    label.config(text="Hello, Welcome to Tkinter!")

# Create the main window
root = tk.Tk()
root.title("Interactive GUI")

# Add a label
label = tk.Label(root, text="Click the button to get greeted",
font=("Arial", 14))
label.pack(pady=20)

# Add a button
button = tk.Button(root, text="Greet Me", command=greet)
button.pack(pady=10)

root.geometry("400x300")
root.mainloop()
```

🎉 Example 3: Calculator

This code creates a simple addition application using the tkinter library. It allows users to enter two numbers, click a button to calculate the sum, and display the result in a message box.

```
import tkinter as tk
```

```python
from tkinter import messagebox

# Function to calculate sum and show result
def find_sum():
    num1 = entry1.get()
    num2 = entry2.get()

    if num1.isdigit() and num2.isdigit():
        result = int(num1) + int(num2)
        messagebox.showinfo("Result", f"Sum: {result}")
    else:
        messagebox.showerror("Error", "Enter valid numbers")

# Create window
root = tk.Tk()
root.title("Addition App")

# Input fields
entry1 = tk.Entry(root)
entry1.pack(pady=5)
entry2 = tk.Entry(root)
entry2.pack(pady=5)

# Button to find sum
tk.Button(root, text="Find Sum", command=find_sum).pack(pady=10)

# Start app
root.geometry("200x150")
root.mainloop()
```

🎉 Example 4: Todo List App

This code creates a simple to-do list using tkinter. Users can add tasks by typing in an entry box and clicking "Add Task." They can also delete tasks from the list by selecting an item and clicking "Delete Task."

```
import tkinter as tk

def add_task():
    task = entry.get()
    if task != "":
        listbox.insert(tk.END, task)
        entry.delete(0, tk.END)

def delete_task():
    listbox.delete(tk.ANCHOR)

# Create the main window
root = tk.Tk()
root.title("To-Do List")

# Add widgets
entry = tk.Entry(root, width=30)
entry.pack(pady=10)

add_button = tk.Button(root, text="Add Task", command=add_task)
add_button.pack(pady=5)

delete_button = tk.Button(root, text="Delete Task", command=delete_task)
delete_button.pack(pady=5)

listbox = tk.Listbox(root, width=40, height=10)
listbox.pack(pady=10)

# Set window size
root.geometry("400x400")
root.mainloop()
```

💡 Challenge 25

1. Create a GUI application that finds simple interest. **Simple Interest (SI) = (P * R * T) / 100**

Where:

- **P = Principal amount**
- **R = Rate of interest**
- **T = Time period**

🌟 Exception Handling In Python

In Python, an **exception** is an error that occurs during the execution of a program, disrupting the normal flow of the program's instructions. Exceptions occur for various reasons, such as invalid user input, a file not being found, or dividing by zero.

Python makes file handling easy with built-in functions and methods that help you work with different types of files, such as text files (.txt), binary files (.bin), or CSV files (.csv).

🚀 Why Use Exception Handling?

- Prevents your program from crashing due to unexpected errors.
- Allows you to handle different error situations without affecting the entire application.
- Ensures that critical cleanup actions are performed (like closing files, freeing resources).

🎉 Basic Syntax of Exception Handling

Python uses the `try` and `except` blocks to handle exceptions. Here's the basic structure:

```
try:
    # Code that might raise an exception
    risky_code()
except SomeSpecificException:
```

```
    # Code that handles the exception
    handle_error()
finally:
    # Optional block that will always be executed, regardless of an
exception
    cleanup_code()
```

try: The block of code that might throw an exception.
except: The block of code that handles the exception if it occurs.
finally: A block that always executes, even if there is no exception. It is typically used for cleanup tasks (e.g., closing files or database connections).

🎉 Example 1: Check For Number

This program prompts the user to enter two numbers. It raises an exception if the input is not a valid number and then calculates the sum of the two numbers:

```
try:
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    print("The sum is:", num1 + num2)
except ValueError:
    print("Invalid input. Please enter a number.")
```

🎉 Example 2: Handling a Specific Exception

In this example, we handle the ZeroDivisionError exception that occurs when dividing by zero.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

🎉 Example 3: Catching Multiple Exceptions

You can handle multiple types of exceptions by specifying different except blocks.

```
try:
    value = int(input("Enter a number: "))
    result = 10 / value
except ValueError:
    print("Error: Invalid input. Please enter a number.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

This code handles two exceptions:

- `ValueError`: Raised when the input is not a number.
- `ZeroDivisionError`: Raised when dividing by zero.

🎉 Example 4: Raising Exceptions Manually

You can raise exceptions manually using the raise keyword if you want to trigger an exception under certain conditions.

```
age = int(input("Enter your age: "))
if age < 18:
    raise ValueError("You must be at least 18 years old.")
```

💡 Challenge 26

Create a Python program that asks the user to input a number. The program should then determine if the number is odd or even and display the result. If the user enters a non-numeric value, handle the exception and prompt the user to enter a valid number.

🌟 File Handling In Python

File handling is an essential part of programming that allows you to work with files stored on your computer. In Python, file handling enables reading from and writing to files efficiently. You can use it to store data, log information, or even save user inputs for future use.

Python makes file handling easy with built-in functions and methods that help you work with different types of files, such as text files (.txt), binary files (.bin), or CSV files (.csv).

🚀 File Handling Modes

- **'r'**: Open a file for reading (default mode).
- **'w'**: Open a file for writing. If the file exists, its content is erased. If the file doesn't exist, it creates a new file.
- **'a'**: Open a file for appending. Data is written to the end of the file, and if the file does not exist, it creates a new file.

🎉 Example 1: Reading from a File

This code opens an existing text file, reads its entire content, and prints it. After reading, the file is closed manually to release system resources.

```
# Open the file in read mode
file = open("example.txt", "r")

# Read the file content
content = file.read()
print(content)

# Close the file
file.close()
```

🎉 Example 2: Writing to a File

This code opens a file in write mode and writes a string into it. If the file exists, its content will be overwritten. If the file doesn't exist, it will be created. The file is then closed to save changes.

```
file = open("example.txt", "w") # Open the file in write mode
# Write some text to the file
file.write("Hello, this is a Python file handling example!")
# Close the file
file.close()
```

🎉 Example 3: Appending to a File

This code opens a file in append mode, adds new content to the end of the file without modifying the existing data, and then closes the file to ensure the appended content is saved.

```
# Open the file in append mode
file = open("example.txt", "a")

# Append text to the file
file.write("\nThis text is appended to the file.")

# Close the file
file.close()
```

🎉 Example 4: Using `with` Statement for File Handling

This code uses the with statement to read a file. The file is automatically closed once the block of code is executed, making it a safer and more concise way to handle files.

```
# Open the file using 'with' statement
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

🎉 Example 5: Creating a Directory

This code demonstrates how to create a new directory using the os module in Python. The os.mkdir() function creates a directory with the specified name. If the directory already exists, it will raise an error.

```
import os

# Directory name
directory = "new_folder"

# Create directory
try:
    os.mkdir(directory)
    print(f"Directory '{directory}' created successfully!")
except FileExistsError:
    print(f"Directory '{directory}' already exists.")
```

🎉 Example 6: Deleting a Directory

In this example, the os.rmdir() function deletes the directory named new_folder.

```
import os

# Directory name
directory = "new_folder"

# Remove directory
try:
    os.rmdir(directory)
    print(f"Directory '{directory}' deleted successfully!")
except FileNotFoundError:
    print(f"Directory '{directory}' does not exist.")
except OSError:
    print(f"Directory '{directory}' is not empty. Unable to delete.")


with open("example.txt", "r") as file:
    # Read each line one by one
    for line in file:
        print(line.strip())
```

💡 Challenge 27

Create a Python program that reads a file containing a list of numbers (one number per line) and writes the sum of those numbers to another file called sum.txt.

🌟 Datetime In Python

Python's datetime module helps you work with dates and times, making it simple to create, manipulate, and format dates. Whether you want to get the current date, calculate time differences, or convert strings into date objects, datetime has got you covered.

🚀 Getting Started with Datetime

To use the `datetime` module, you need to import it first:

```
import datetime
```

🎉 Example 1: Get the Current Date and Time

This code gets the current date and time.

```
import datetime

# Get current date and time
now = datetime.datetime.now()
print("Current Date and Time:", now)
```

🎉 Example 2: Get the Current Date

This example gets only the current date (without the time).

```
import datetime

# Get current date
current_date = datetime.date.today()
print("Current Date:", current_date
```

🎉 Example 3: Get Specific Components of a Date

This code opens a file in append mode, adds new content to the end of the file without modifying the existing data, and then closes the file to ensure the appended content is saved.

```
import datetime

# Get current date
today = datetime.date.today()

# Extract year, month, and day
```

```
print("Year:", today.year)
print("Month:", today.month)
print("Day:", today.day)
```

🎉 Example 4: Creating a Date Object

You can create a date object by passing year, month, and day.

```
import datetime

# Create a specific date
my_birthday = datetime.date(1995, 5, 15)
print("My Birthday:", my_birthday)
```

🎉 Example 5: Time Differences with `timedelta`

You can calculate differences between dates using timedelta.

```
import datetime

# Get current date
today = datetime.date.today()

# Create a timedelta of 10 days
ten_days = datetime.timedelta(days=10)

# Calculate future and past dates
future_date = today + ten_days
past_date = today - ten_days

print("Date after 10 days:", future_date)
print("Date 10 days ago:", past_date)
```

🎉 Example 6: `time.sleep()` to Pause Your Program

Sometimes you might want to pause your program for a certain period. You can use time.sleep() to do this. It pauses the execution for the specified number of seconds.

```
import time

# Print a message
print("This message will be followed by a 3-second pause...")

# Pause the program for 3 seconds
time.sleep(3)

# Print after the pause
print("3 seconds have passed!")
```

💡 Challenge 28

Create a Python program that asks the user for their birthdate (in YYYY-MM-DD format) and calculates their age.