## ​​

## **✅ Python Basics**

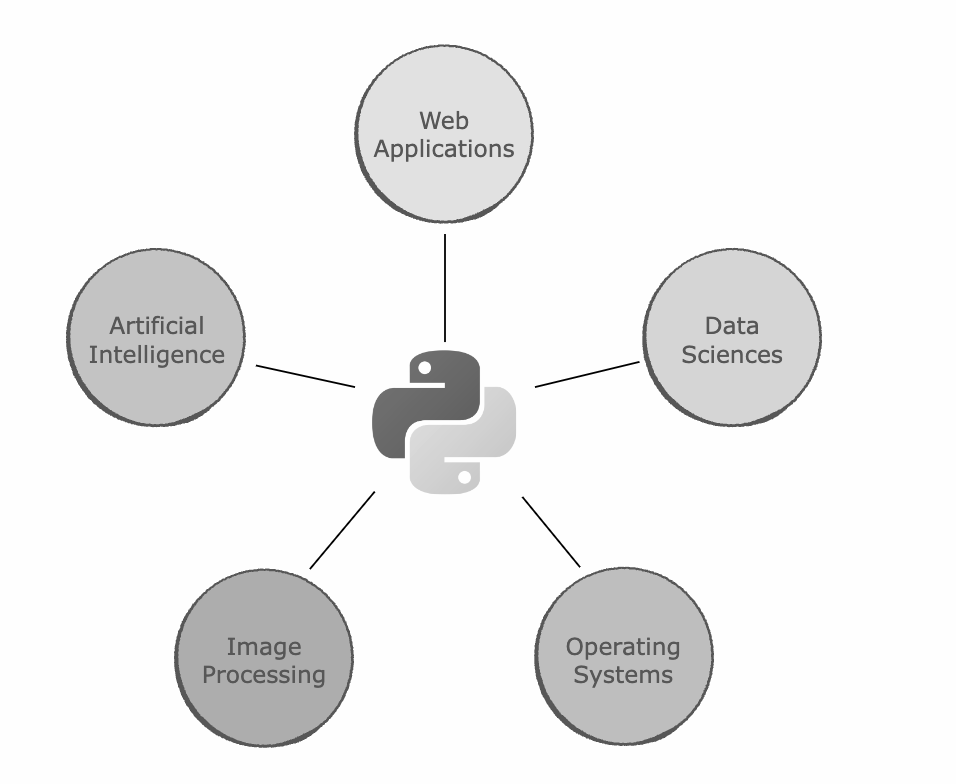
## **Why should you learn Python?**

* Easy to learn: Python has a simple and readable syntax
* Large community massive and supportive community of programmers.
* High demand : Python is in high demand in the job market.

## **➕What is Python?**

Developed in 1990, Python is one of the most popular general-purpose programming languages in modern times.

* High-level programming language
* General-purpose
* Interpreted
* Object-oriented
* Dynamically typed
* Emphasizes readability
* Extensive libraries and frameworks
* Cross-platform compatibility
* Large supportive community
* Popular in web development, data analysis, AI, and more.



### 

### # Python Case Sentive

### # True , true

### #Indentation in Python

### print("Hello World") print("Pramod")

✅ Dynamically typed

### Python determines the type of a variable during runtime, rather than during compilation.

| age = 25 # The interpreter automatically determines the type of the variable: print(type(age)) # Output: <class 'int'>  # You can also reassign a variable to a different type without any issues: age = "twenty-five" print(type(age)) # Output: <class 'str'> |
| --- |

✅ Python 2.7 Vs 3.x ?



👉 Highly recommended to use Python 3 for new projects and migrate existing projects to Python 3, if possible.

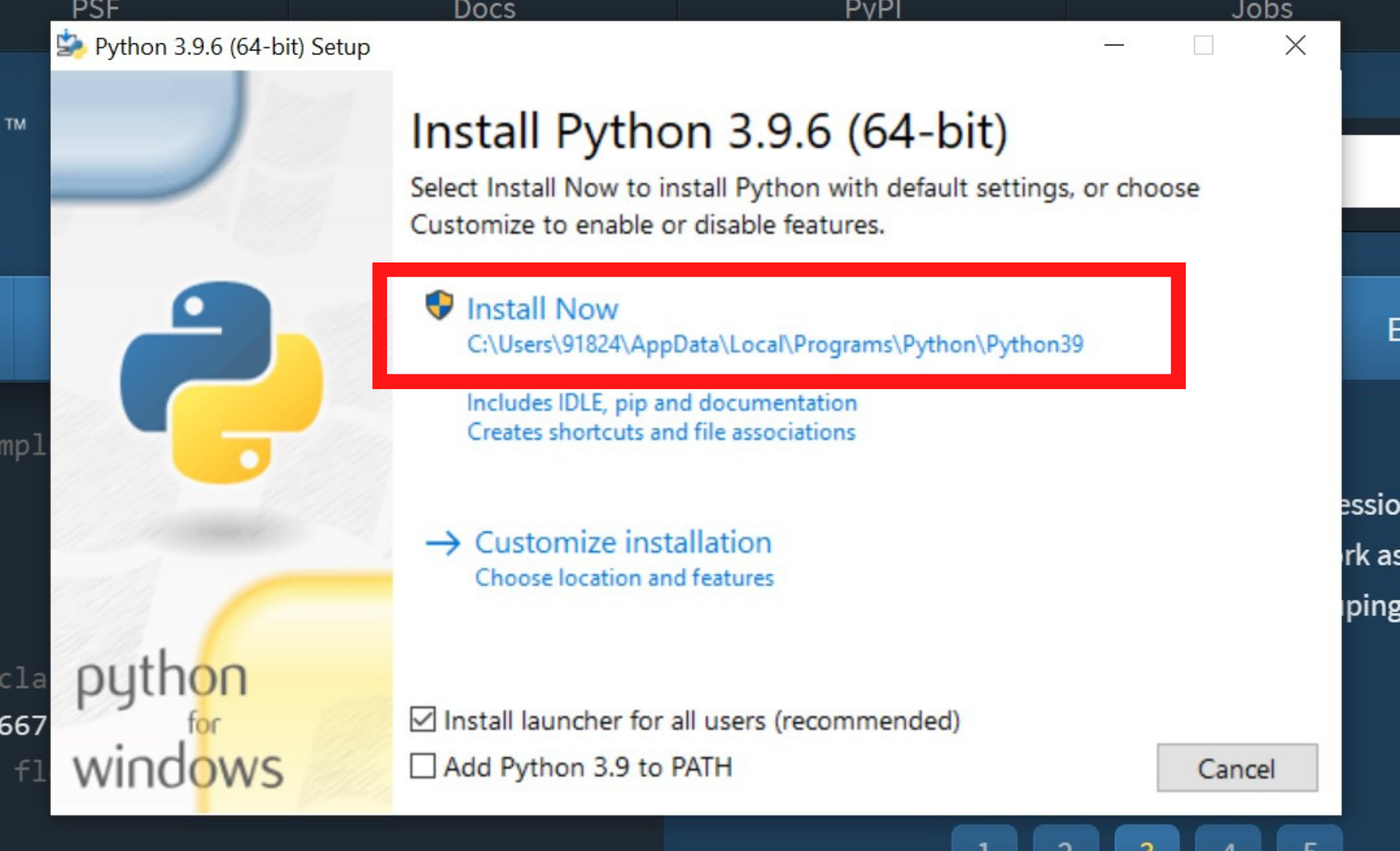
## 

## 

## ✅ Installation of Python 3.x?

### For Windows:

* Visit the official Python website: https://www.python.org/downloads/
* Click the "Download Python 3.x.x" button (replace 'x' with the latest version number).
* Run the downloaded installer (python-3.x.x.exe).
* In the installer, check the box next to "Add Python 3.x to PATH" to add Python to your system's environment variables.



* Choose the "Customize installation" option if you want to change the installation location or features, or just click "Install Now" for a default installation.
* Wait for the installation to finish.
* Click "Close" once the installation is complete.
* Open the Command Prompt or PowerShell and type python --version to verify the installation. The output should display the installed Python version.

### For macOS:

* Visit the official Python website: https://www.python.org/downloads/
* Click the "Download Python 3.x.x" button (replace 'x' with the latest version number).
* Open the downloaded installer (python-3.x.x.pkg).
* Follow the on-screen instructions in the installer.
* Click "Continue" and then "Install" to begin the installation process.
* You might be prompted to enter your macOS user password. Enter it and click "Install Software."
* Wait for the installation to finish.
* Click "Close" once the installation is complete.
* Open Terminal and type python3 --version to verify the installation. The output should display the installed Python version.
* Note that macOS comes with Python 2.x pre-installed, so using the python command will still refer to Python 2.x. Use the python3 command to run Python 3.x.

##### Writing Our First Code

The print Statement

| print("Hello World") |
| --- |

| print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False) |
| --- |

* **\*objects:** This argument represents a variable number of objects to be printed. You can pass any number of objects separated by commas. These objects will be converted to strings using the str() function before being printed.
* **sep:** This is an optional argument that specifies the string that separates multiple objects when printed. The default separator is a space ' '.
* **end:** This is an optional argument that specifies the string that is printed at the end of the line. The default value is a newline character '\n', which causes the output to move to the next line after printing.
* **file:** This is an optional argument that defines where the output is printed. By default, it is set to sys.stdout, which represents the console. You can change this to a file object if you want to print the output to a file.
* **flush**: This is an optional argument that, when set to True, forces the output to be written immediately. By default, it is set to False. When set to False, the output may be buffered until enough data is available to write or until the file is closed.

Here's an example using all the arguments:

python

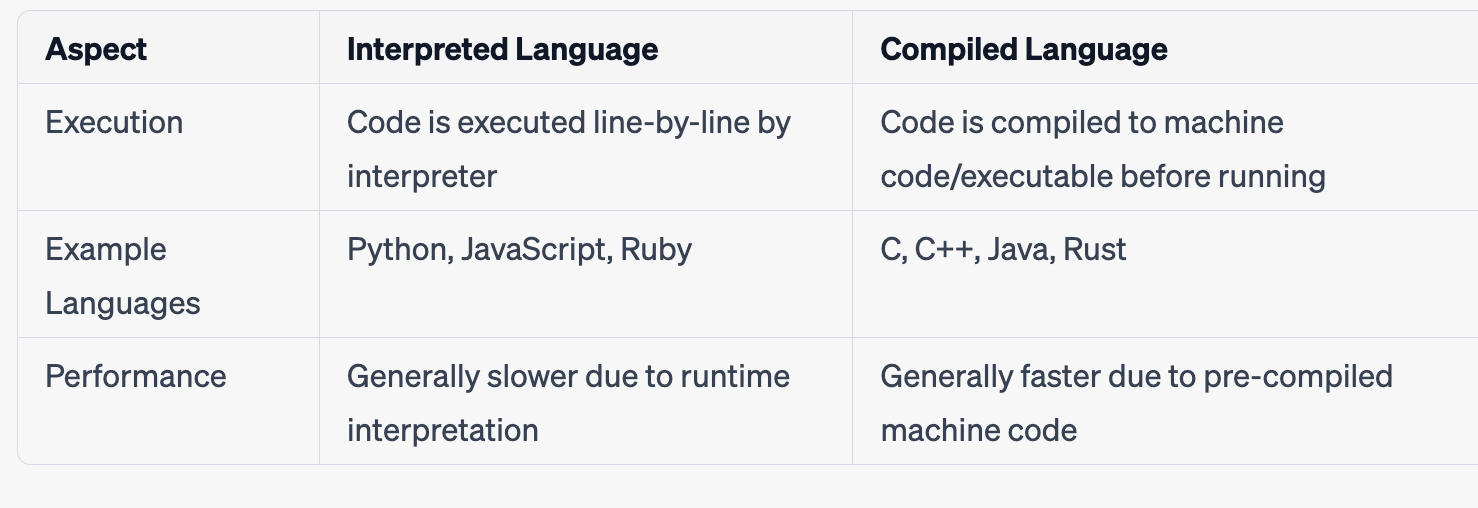
| with open("output.txt", "w") as file:  print("Hello", "World", sep="-", end="!", file=file, flush=True) |
| --- |

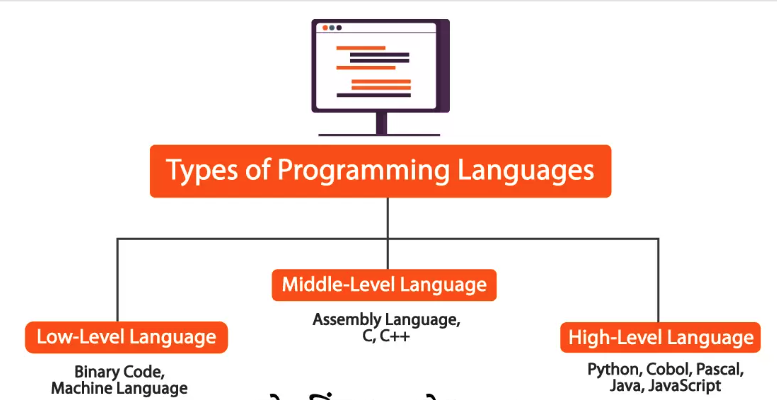
In this example, we print "Hello" and "World" with a dash separator, followed by an exclamation mark at the end.

The output is written to a file named "output.txt" instead of the console. The flush=True argument ensures the output is written immediately to the file.

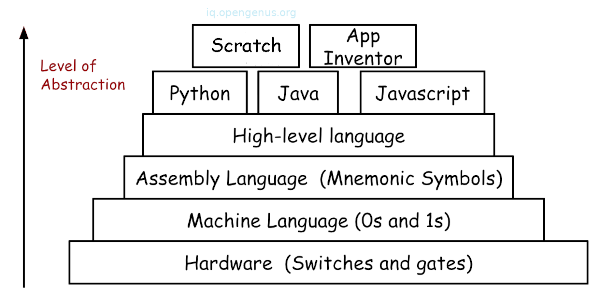
| print(50, 1000, 3.142, "Hello World") |
| --- |

Python is Interpreted Language



* 

Check the Languages, with the Level of Abstraction



**Comments in Python**

| print(99) # This line prints 99 print("Hello World") # This line prints Hello World  # This is just a comment hanging out on its own!  # For multi-line comments, we must # add the hashtag symbol # each time |
| --- |

**Multiple lines**

""" Docstrings are pretty coofor writing longer comments or notes about the code"""

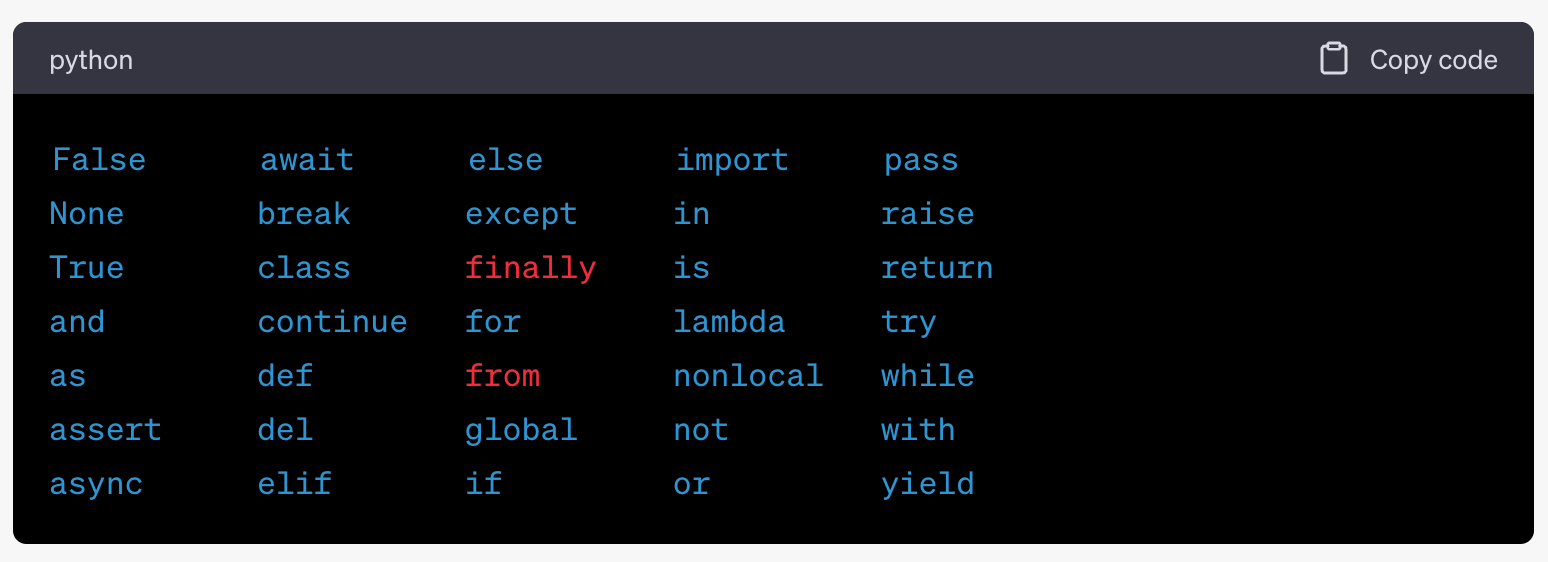
**⚠️ What is Source Code?**

Human understandable code written using High Level Programming language is called as Source Code. (NameOfFile.py)

## 

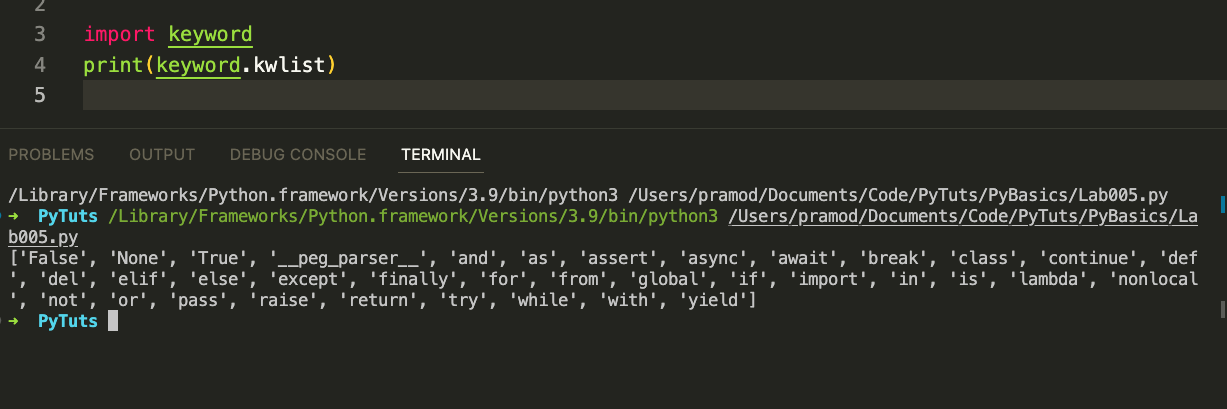
## 📕 **Keywords & Identifiers**

* Keywords are also called as Reserved Words.
* All the keywords can be in Lower Case or upper Case.
* We cannot use a keyword as a variable name, function name or any other identifier.
* They are used to define the syntax and structure of the Python language. In Python, keywords are case-sensitive.



| import keyword print(keyword.kwlist) |
| --- |

**Lab005.py**



### Identifiers

**Identifiers** are the names used to identify a variable, function, class, module, or other objects.

* They start with a letter (A-Z or a-z) or an underscore (\_) followed by zero or more letters, underscores, and digits (0-9).
* Python is case-sensitive, so myVariable and myvariable are two different identifiers.

Rules for writing identifiers:

Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (\_).

* An identifier cannot start with a digit.
* Keywords cannot be used as identifiers.
* We cannot use special symbols like !, @, #, $, %, etc. in our identifier.
* An identifier can be of any length.

Examples of valid identifiers: myVar, var1, \_var, \_1\_var

Examples of invalid identifiers: 1var, my-var, my@, my var, break

(break is a keyword)

## **✅ Variables and Data Types**

Variables

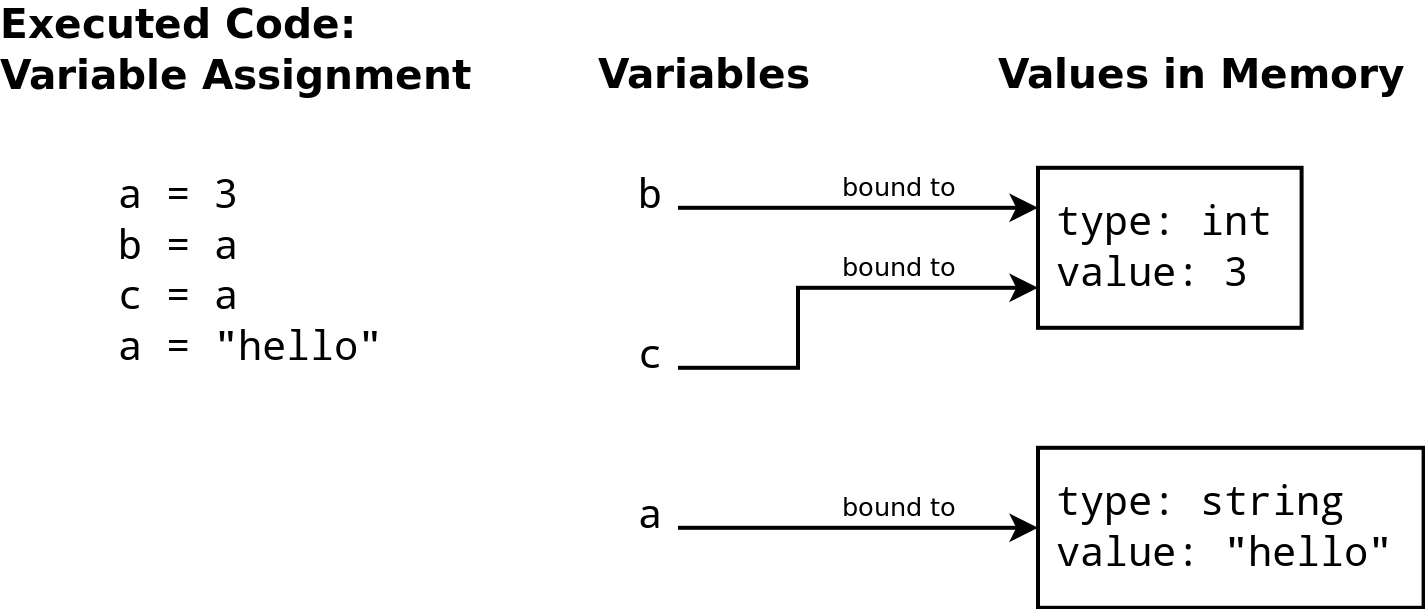
* A variable is a container (storage area) used to hold data.
* Each variable should be given a unique name (identifier).
* Variables are created on demand whenever a value is assigned to them using the equals sign = which is known as the assignment operator.
* Value of the variable can be changed any number of times during the program execution

| x = 5 # x is an integer pi = 3.14 # pi is a floating point number name = "Python" # name is a string  x,y,z = 0,1,2 |
| --- |

Three main Types of Variables

There are two types of variables based on data type used to declare the variable.

1. Numbers
2. Strings
3. Booleans





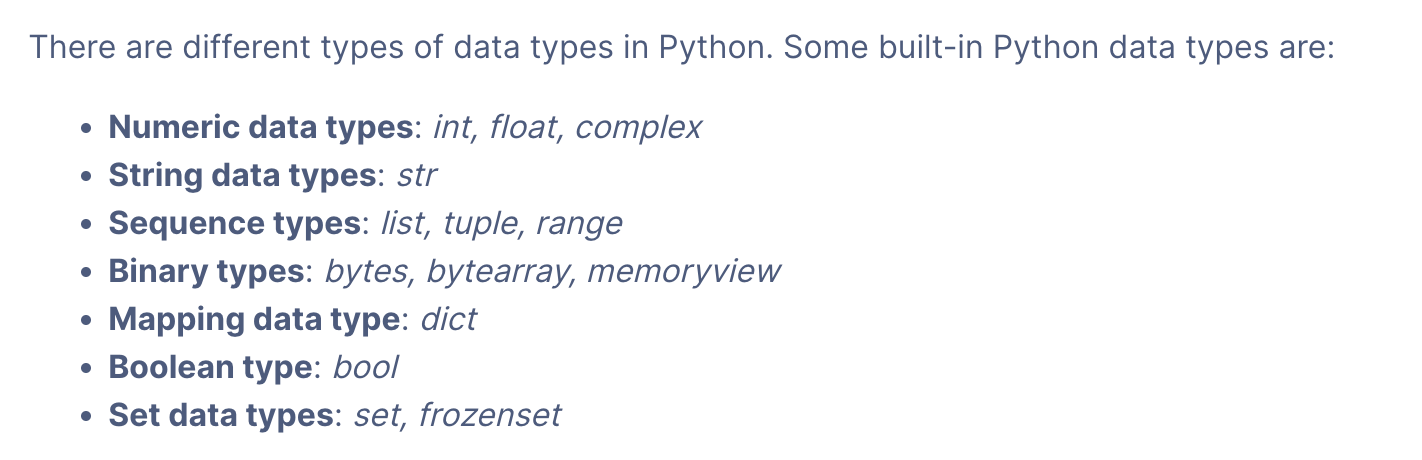
Naming Conventions

1. Variables: Use lowercase letters and separate words with underscores (snake\_case). For example: user\_name, count.
2. Constants: Use uppercase letters and separate words with underscores. For example: PI, MAX\_SIZE.
3. Functions: Use lowercase letters and separate words with underscores (snake\_case). For example: calculate\_sum, read\_file.
4. Classes: Use PascalCase (or CamelCase) for class names, capitalizing the first letter of each word. For example: MyClass, Person.
5. Modules: Use lowercase letters and separate words with underscores (snake\_case). For example: my\_module, file\_handler.
6. Packages: Use lowercase letters and avoid underscores if possible. For example: mypackage, utilities.
7. Protected instance variables: Start with a single underscore followed by lowercase letters and underscores (snake\_case). For example: \_protected\_variable.
8. Private instance variables: Start with two underscores followed by lowercase letters and underscores (snake\_case). For example: \_\_private\_variable.
9. Methods: Use lowercase letters and separate words with underscores (snake\_case). For example: get\_name, set\_value.
10. Avoid using single character names: Do not use single character names like i, x, etc., except in specific cases like loop counters. Use descriptive names that provide context.

### **Data Types**

Numbers

1. Integers - Positive and negative whole numbers.
2. Floating Points Numbers
3. Complex Numbers
4. Strings, ​​Lists, Tuples, Dictionaries, Booleans
5. Sets

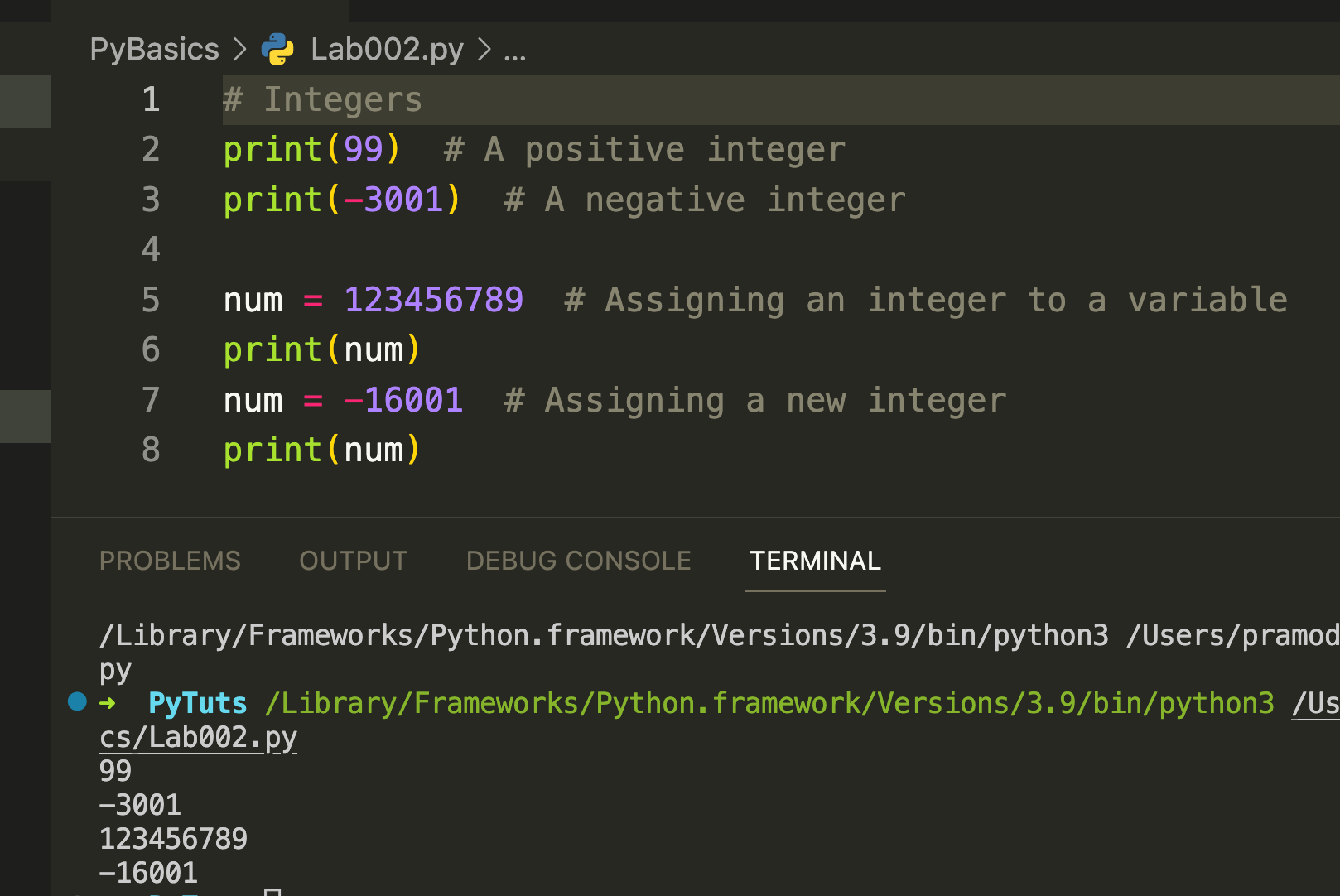


Ref

[Python Data Types (With Complete List) | DigitalOcean](https://www.digitalocean.com/community/tutorials/python-data-types)

##### Integers

0 will take up 24 bytes whereas 1 would occupy 28 bytes



Add Two Numbers with User Input

In Python, the input() function is used to take user input. By default, the input() function returns data as a string.

To get a numerical input from a user, you should convert it to an integer or float using the int() or float() function, respectively.

x = input("Type a number: ")

y = input("Type another number: ")

sum = int(x) + int(y)

print("The sum is: ", sum)

##### 

##### **Floating Point Numbers**

Python allows us to create decimals up to a very high decimal place.

This ensures accurate computations for precise values.

A float occupies 24 bytes of memory.

**9 is considered to be an integer while 9.0 is a float.**

##### **Complex Numbers -**

Complex numbers are useful for modelling physics and electrical engineering models in Python.

Here’s the template for making a complex number:  
  
complex(real, imaginary)

| print(complex(2.5, -18.2)) # Represents the complex number (2.5 - 18.2j) |
| --- |

##### **Strings**

Strings are sequences of characters. In Python, you can define strings using either single quotes (''), double quotes ("") or triple quotes (''' or """) for multiline strings.

| **s = "Hello, world!"** |
| --- |

raw strings

print('C:\some\name') # - ​​C:\some ame

Proper - print(r'C:\some\name')

###### Length of a String

| random\_string = "I am Batman" # 11 characters print(len(random\_string)) |
| --- |

###### Indexing and Reverse Indexing

A string in Python is indexed from 0 to n-1 where n is its length.

| batman = "Bruce Wayne" first = batman[0] # Accessing the first character print(first) space = batman[5] # Accessing the empty space in the string print(space) last = batman[len(batman) - 1] print(last) # The following will produce an error since the index is out of bounds # err = batman[len(batman)] |
| --- |

| batman = "Bruce Wayne" print(batman[-1]) # Corresponds to batman[10] print(batman[-5]) # Corresponds to batman[6] |
| --- |

String Immutability

| string = "I am Immutable" string[0] = '0' # Will give error |
| --- |

How to verify that String are immutable in python?

With ID method

| str1 = "Pramod" print(id(str1)) str1 = "Dutta" print(id(str1)) |
| --- |

**NoneType**

We can assign None to any variable, but we can not create other NoneType variables.

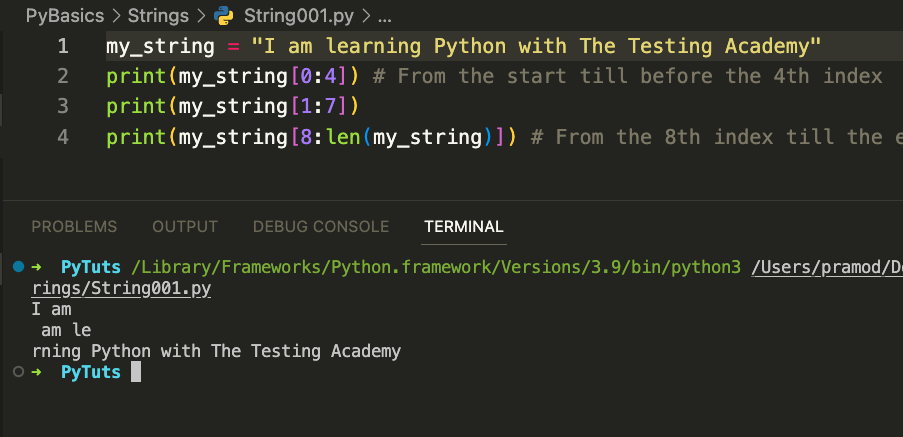
* None is not a default value for the variable that has not yet been assigned a value.
* None is not the same as False.
* None is not an empty string.
* None is not 0.

| val = None print(val) # prints "None" and returns None print (type(val)) |
| --- |

###### **String Slicing**

Slicing is the process of obtaining a portion (substring) of a string by using its indices.

string[start:end]



Slicing with a Step

The default step is 1.Python 3 also allows us to slice a string by defining a step through which we can skip characters in the string.

String002.py



Reverse Slicing

| # Reverse Slice my\_string = "This is Pramod!" print(my\_string[13:2:-1]) # Take 1 step back each time print(my\_string[17:0:-2]) # Take 2 steps back. The opposite of what happens in the slide above |
| --- |

##### 

Partial Slicing

| my\_string = "This is MY string!" print(my\_string[:8]) # All the characters before 'M' print(my\_string[8:]) # All the characters starting from 'M' print(my\_string[:]) # The whole string print(my\_string[::-1]) # The whole string in reverse (step is -1) |
| --- |

**How to Reverse a String in Python**

| txt = "Hello World"[::-1] print(txt) |
| --- |

##### **Lists**

A list in Python is a collection of items which can be of different types. The items are enclosed within brackets [] and separated by commas.

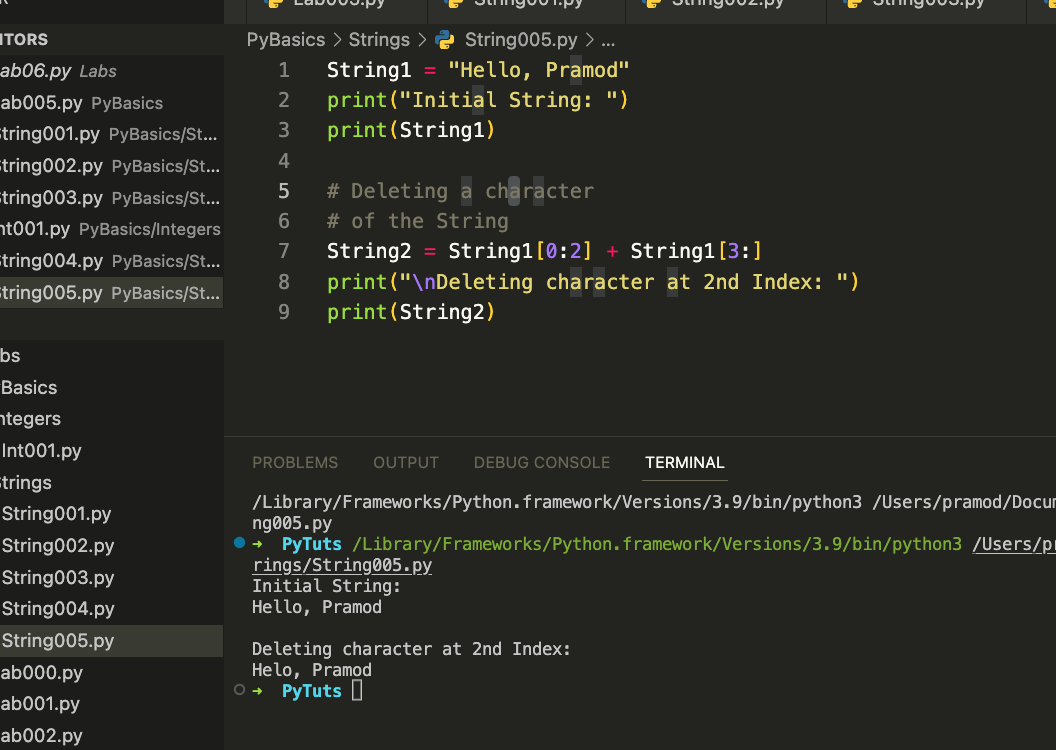
| **my\_list = [1, 2, 3, 'four', 5.0]** |
| --- |

##### 

Deleting/Updating from a String

| String1 = "Hello, I'm a Pramod" list1 = list(String1) print(list1) list1[2] = 'p' String2 = ''.join(list1) print(String2) |
| --- |

Deletion of a character ( String005.py)



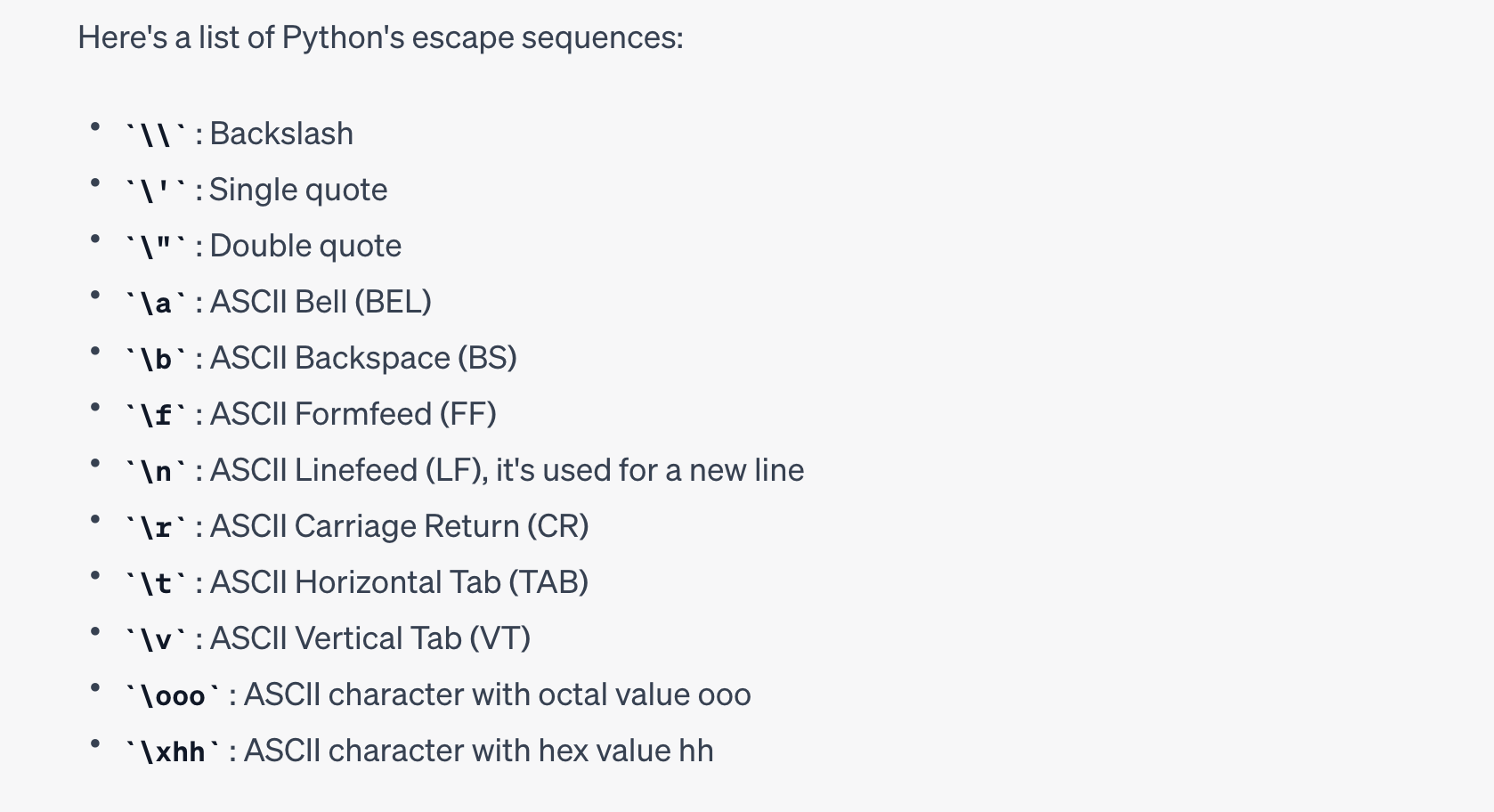
Delete String

| del String1 |
| --- |

##### 

##### **Escape Sequence**

Escape sequences in Python are special combinations of characters that have a meaning other than the literal characters



| print("\nEscaping Backslashes: ") |
| --- |

| **print("This is a backslash \\") print("She said: \"Hello, world!\"") print("This is a new line \nThis is the second line") print("This is a tab\tThis is after the tab")** |
| --- |

In Python, if you don't want escape sequences to be interpreted, you can use raw strings by prefixing the string literal with an r or R. For example:

| print(r"This is a backslash \\") #output // This is a backslash \\ |
| --- |

##### **String Formatting**

String formatting means substituting values into a string. Following are some use cases of string formatting:

* Inserting strings within a string
* Inserting integers within a string
* Inserting floats within a string

| string1 = "I like %s" % "Python" print(string1) # 'I like Python'  temp = "Pramod" string2 = "I like %s" % temp print(string2) # 'I like TTA'  string3 = "I like %s and %s" % ("Python", temp) print(string3) # 'I like Python and TTA' |
| --- |

| my\_string = "%i + %i = %i" % (1,2,3) print(my\_string) # '1 + 2 = 3 |
| --- |

| string1 = "%f" % (1.11) print(string1) # '1.110000'  string2 = "%.2f" % (1.11) print(string2) # '1.11'  string3 = "%.2f" % (1.117) print(string3) # '1.12' |
| --- |

##### **Booleans**

The Boolean data type can have two values: True or False. It is used to represent the truth values of expressions.

| **is\_true = True is\_false = False** |
| --- |

##### 

##### **Sets**

A set is an unordered collection of unique items. Sets are defined within braces {}.

| **my\_set = {1, 2, 3, 4, 4, 5} # Duplicates will be removed, so this is equivalent to {1, 2, 3, 4, 5}** |
| --- |

You can use the type() function to find out the type of a variable:

| **x = 5 print(type(x)) # Outputs: <class 'int'>** |
| --- |

### 

**UNICODE Characters**

* UNICODE stands for UNIversal CODE.
* Every character will have UNICODE value.
* UNICODE Notation
* Syntax:
  + \uXXXX - X will be hexadecimal digit
* Starts with \u followed by four hexadecimal digits.
* UNICODE Range
* \u0000 (0) to \uFFFF (65535)

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

## ✅ Literals

* Literals are the actual values assigned
* Literals can be Numeric and Non Numeric.

| Literal Type | Description |
| --- | --- |
| Integer literals | Whole numbers without decimal points, such as 42 or -123 |
| Floating-point literals | Numbers with decimal points, such as 3.14 or -0.0025 |
| Character literals | A single character enclosed in single quotes, such as 'a' |
| Boolean literals | A value of either true or false |
| String literals | A sequence of characters enclosed in double quotes, such as "hello world" |
| None literals | A special literal that represents None |

Types of Literals

1) Boolean Literals - true, false

2) Character Literals -

3) String Literals

4) Integral Literals

5) Floating Literals

6) None Literal

#### 1. Boolean Literals

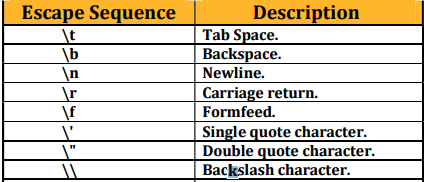
There are two boolean literals 1) true 2) false

#### 2. Character Literals

A char type variable can hold following:

* Single character enclosed in single quotation marks
* Escape Sequence
* ASCII Value
* UNICODE Character
* Octal Character

#### Escape Sequence - Task\_12.java



ASCII stands for American Standard Code for Information Interchange.

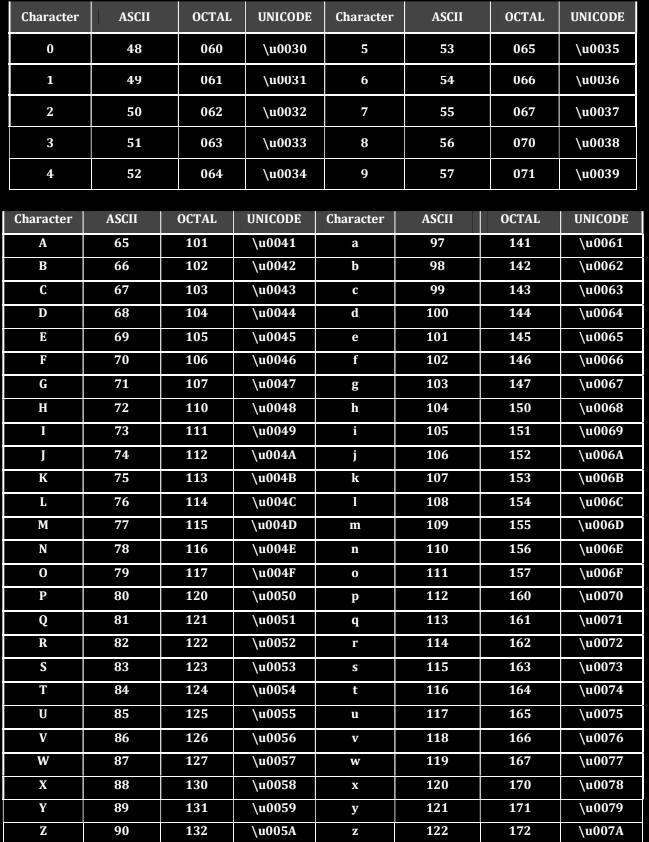
* Every character enclosed in single quotation marks will have an integer equivalent value
* called as ASCII value.
* ASCII Value Range is 0 – 255.
* ASCII Value can be assigned to a char type variable

Octal Value as char type ( 0)

061 -> 49

061 = (0 × 8²) + (6 × 8¹) + (1 × 8⁰) = 49

<https://www.rapidtables.com/convert/number/octal-to-decimal.html>



Syntax: \DDD - D will be octal digit

OCTAL Range Range in Decimal 0 - 255 Range in Octal \0 - \377

In Python, a literal refers to any number or text that appears directly in your code. In other words, literals are data given in a variable or constant. Python supports various types of literals, including:

1. \*\*Numeric Literals\*\*: Numeric Literals are immutable. There are three types of Numeric literals:

- \*\*Integer\*\*: It consists of a set of all positive and negative integers without a fractional component. For example, 123, -786, 0, 777.

- \*\*Float\*\*: Float literals comprise integer and fractional components. For example, 12.23, -97.56, 0.123.

- \*\*Complex\*\*: Complex literals are in the form a+bj, where a forms the real part, and b forms the imaginary part of complex number. For example, 3.14j, 4+3.14j.

2. \*\*String Literals\*\*: String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string. For example, "Hello", 'World'. A string in Python can be multi-line if it's enclosed in triple quotes: """...""" or '''...'''.

3. \*\*Boolean Literals\*\*: A Boolean literal can have any of the two values: True or False.

4. \*\*Special Literals\*\*: Python contains one special literal i.e., None. 'None' is used to specify that a field is not created.

5. \*\*Literal Collections\*\*: There are four types of literal collection: List literals, Tuple literals, Dict literals, and Set literals.

- \*\*List literals\*\*: Lists are enclosed in square brackets [ ] and each item is separated by a comma. For example, [1, 2, "a", "b"]

- \*\*Tuple literals\*\*: Tuples are enclosed in parentheses ( ) and each item is separated by a comma. For example, (1, 2, "a", "b")

- \*\*Dict literals\*\*: Dictionaries are enclosed in curly brackets { } and values are assigned by key-value pairs. Each key is separated from its value by a colon (:). For example, {"name": "John", "age": 30}

- \*\*Set literals\*\*: Sets are unordered collection of unique items. Set is defined by values separated by comma inside braces { }. For example, {1, 2, 3}

6. \*\*None Literal\*\*: None is a special constant in Python that represents the absence of a value or a null value. It is an object of its own datatype, the NoneType.

For example:

```python

| # Numeric literals a = 0b1010 #Binary Literals b = 100 #Decimal Literal  c = 0o310 #Octal Literal d = 0x12c #Hexadecimal Literal  # Floating Point Literal float\_1 = 10.5  float\_2 = 1.5e2  # Complex Literal  x = 3.14j  print(a, b, c, d) print(float\_1, float\_2) print(x, x.imag, x.real)  # String literals char = "hello world" multiline\_str = """This is a multiline string with more than one line code.""" unicode = u"\u00dcnic\u00f6de" raw\_str = r"raw \n string"  print(char) print(multiline\_str) print(unicode) print(raw\_str)  # Boolean literals x = (1 == True) y = (1 == False) a = True + 4 b = False + 10  print("x is", x) print("y is", y) print("a:", a)   print("b:", b)  # Special literal drink = "Available" food = None  def menu(x):  if x == drink:  print(drink)  else:  print(food)  menu(drink) menu(food)  # Literal Collections fruits = ["apple", "mango", "orange"] #list numbers = (1, 2, 3) #tuple alphabets = {'a':'apple', 'b':'ball', 'c':'cat'} #dictionary vowels = {'a', 'e', 'i' , 'o', 'u'} #set  print(fruits) print(numbers) print(alphabets) print(vowels) ``` |
| --- |

| Question | Answer |
| --- | --- |
| What is a literal? | A literal is a fixed value that is directly used in a program without needing to be computed or evaluated. |
| How many types of literals are available? | There are six types of literals available in programming languages: numeric literals, character literals, boolean literals, string literals, array literals, and null literals. |
| How many boolean literals are available? | There are only two boolean literals available, which are true and false. |
| What will happen when I assign 1 to boolean type variable? | When you assign the value 1 to a boolean type variable, it will be interpreted as true because any non-zero value is considered as true in boolean expressions. |
| Can we store empty character in char type variable? | No, you cannot store an empty character in a char type variable because it requires at least one character. |
| How to store single quote in char variable? | To store a single quote in a char variable, you need to use the escape sequence ' because the single quote is a reserved character in programming languages. |
| What is Escape Sequence? | An escape sequence is a combination of characters used to represent special characters or non-printable characters in a string literal. It usually starts with a backslash () character. |
| What will be displayed when UNICODE value is found in String Literal? String str="UNICODE of A is \u0041"; | The string "UNICODE of A is A" will be displayed because \u0041 represents the Unicode value of the letter A. |
| What will be displayed when Octal representation is found in String Literal? String str="Octal of A is \101"; | The string "Octal of A is A" will be displayed because \101 represents the octal value of the letter A. |
| What will happen when Escape Sequence is found in String Literal? String str="Hello\nGuys"; | The string "Hello" will be displayed on the first line, and "Guys" will be displayed on the next line because \n represents the newline character. |

#### Integer Literals

* Decimal Literals -
* Octal Literals - int b=0101;
* Hexadecimal Literals - int c=0Xface; // base 16 rather than base 10
* Binary Literals () - int b2 = 0b101;

#### 

## ✅ Operators

Operators are used to perform operations by using operands.

There are three types of operator depending on the number of operands required.

* 1) Unary Operator
  + Only one operand is required.
* 2) Binary Operator
  + Two operands are required.
* 3) Ternary Operator
  + Three Operands are required.

##### **Unary Operator:**

Unary operators operate on just one operand. An example in Python would be the negative sign in front of a number, which flips the sign of the number, or the "not" operator, which inverts a boolean value.

| n = 7 print(-n) # Prints: -7 flag = True print(not flag) # Prints: False |
| --- |

In Python, unary operators are operators that operate on a single operand. Here are the common unary operators:

1. \*\***Unary Plus** (+)\*\*: This operator doesn't really do anything. It's more for clarity and symmetry in your code.

```python

| a = 5  print(-a) # Prints: -5 |
| --- |

```

2. \*\***Unary Minus** (-)\*\*: This operator negates the value of the operand.

```python

| a = 5 print(-a) # Prints: -5 |
| --- |

```

3. \*\***Logical Negation (not)\*\***: This operator returns `True` if the operand is `False`, and `False` if the operand is `True`.

```python

| a = True  print(not a) # Prints: False |
| --- |

```

4. \*\***Bitwise Not (~)\*\*:** This operator returns the bitwise complement of the operand. In other words, it switches each 1 for a 0 and each 0 for a 1. It's equivalent to `-x - 1`.

| a = 5 # binary: 101  print(~a) # Prints: -6, binary: -110 |
| --- |

5. \***\*Identity Operators (is, is not)\*\*:** These are used to check if two variables are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

| a = 5  b = 5  print(a is b) # Prints: True   list1 = [1, 2, 3]  list2 = [1, 2, 3]  print(list1 is list2) # Prints: False   print(a is not b) # Prints: False  print(list1 is not list2) # Prints: True |
| --- |

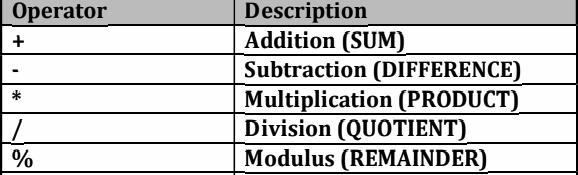
```

**Even though `list1` and `list2` contain the same elements, they are not the same object, so `list1 is list2` is `False`.**

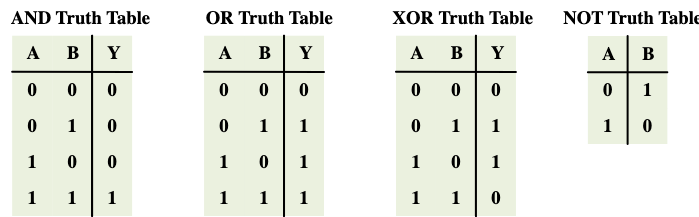
**Remember, it's important to know that `is` and `==` are different** operators: `is` checks for identity, while `==` checks for equality.

##### Binary Arithmetic Operators

Binary operators operate on two operands. Common examples include arithmetic operators like +, -, \*, /, and logical operators like and, or



| x = 10 y = 20  print(x + y) # Prints: 30 print(x < y) # Prints: True  # Logical Operators a = True b = False  print(a and b) # Prints: False print(a or b) # Prints: True |
| --- |



**Ternary Operator:**

The ternary operator is a more concise way of writing an if-else statement. It allows to quickly test a condition instead of a multiline if-else statement.

| x = 10 y = 20 # "a if condition else b" - if the condition is true, it returns a, else it returns b. print("x is greater" if x > y else "y is greater") # Prints: "y is greater" |
| --- |

#### 

#### String Concatenation Operator (+)

+ operator can be used for two purposes:

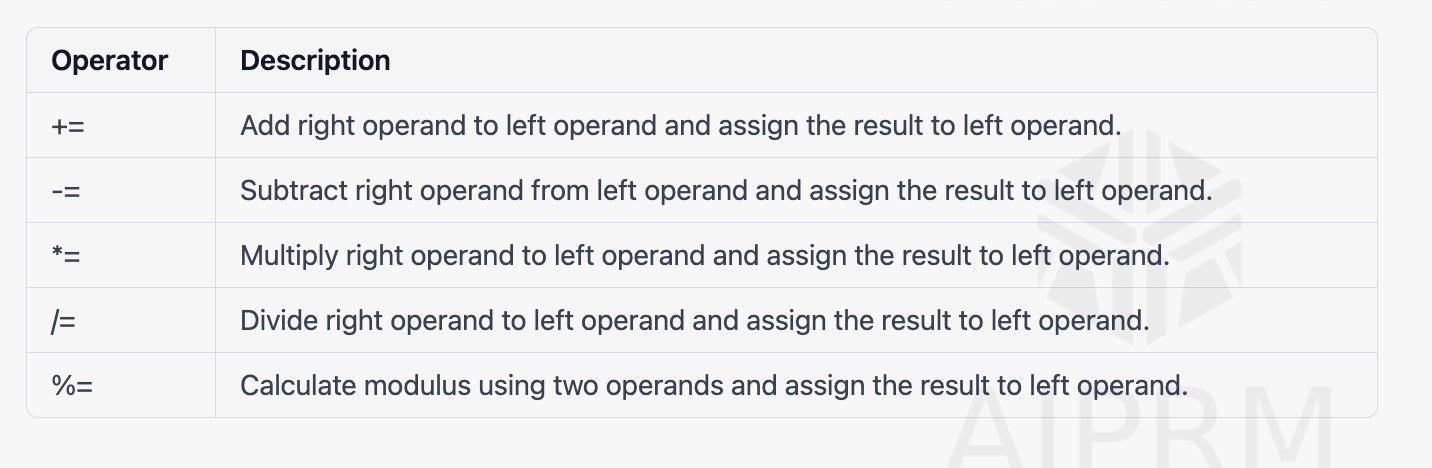
| str1 = "Hello, " str2 = "World!" str3 = str1 + str2 print(str3) # Prints: Hello, World! |
| --- |

If you try to concatenate a string with a non-string, you will get a TypeError

| str1 = "Hello, " num = 7 str3 = str1 + num # This will raise a TypeError  # Fixed - str3 = str1 + str(num) |
| --- |

Python also supports string concatenation using the += operator:

| greeting = "Hello, " greeting += "World!" print(greeting) # Prints: Hello, World! |
| --- |



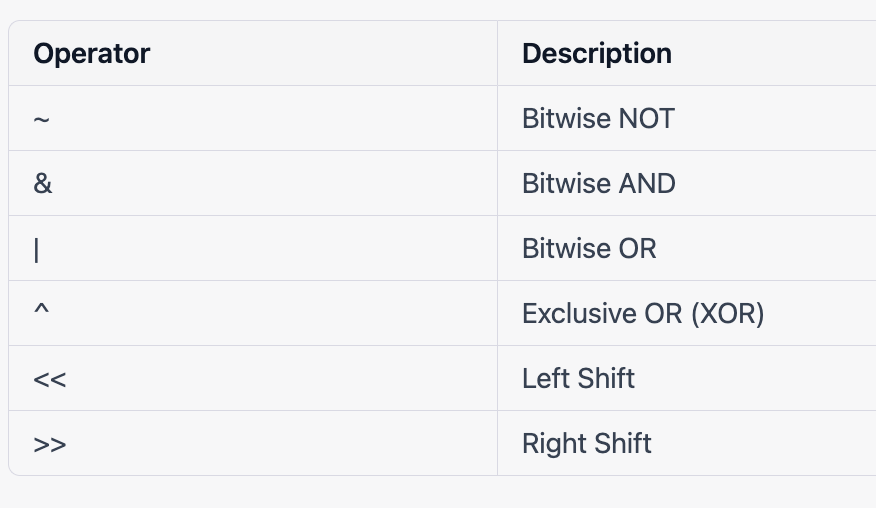
#### **Increment (++) / Decrement (--) Operators**

| # Increment x = 5 x += 1 # This is equivalent to x = x + 1 print(x) # Prints: 6  # Decrement y = 10 y -= 1 # This is equivalent to y = y - 1 print(y) # Prints: 9 |
| --- |

#### Relational Operators

#### Logical Operators Logical OR and Logical AND

#### Bitwise Operators



Bitwise operators in Python operate on binary representations of integers. Here are the main bitwise operators:

1. \*\*AND (&)\*\*: Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

| ```python  a = 10 # 1010 in binary  b = 4 # 0100 in binary   print(a & b) # Prints: 0 (0000 in binary)  ``` |
| --- |

2. \*\*OR (|)\*\*: Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

| ```python  a = 10 # 1010 in binary  b = 4 # 0100 in binary   print(a | b) # Prints: 14 (1110 in binary)  ``` |
| --- |

3. \*\*NOT (~)\*\*: Takes one number and inverts all bits of it.

| ```python  a = 10 # 1010 in binary   print(~a) # Prints: -11 (Inverts all bits and adds one due to two's complement)  ``` |
| --- |

4. \*\*XOR (^)\*\*: Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

| ```python  a = 10 # 1010 in binary  b = 4 # 0100 in binary   print(a ^ b) # Prints: 14 (1110 in binary)  ``` |
| --- |

5. \*\*Right Shift (>>)\*\*: The right shift operator shifts the bits of the number to the right and fills 0 on voids left as a result. Similar effect as of dividing the number with some power of two.

| ```python  a = 10 # 1010 in binary   print(a >> 1) # Prints: 5 (Shifts all bits to the right, 0101 in binary)  ``` |
| --- |

6. \*\*Left Shift (<<)\*\*: The left shift operator shifts the bits of the number to the left and fills 0 on voids right as a result. Similar effect as of multiplying the number with some power of two.

| ```python  a = 10 # 1010 in binary   print(a << 1) # Prints: 20 (Shifts all bits to the left, 10100 in binary)  ``` |
| --- |

<https://bit-calculator.com/bit-shift-calculator>

## ✅ Conditions and Loop

#### If Else

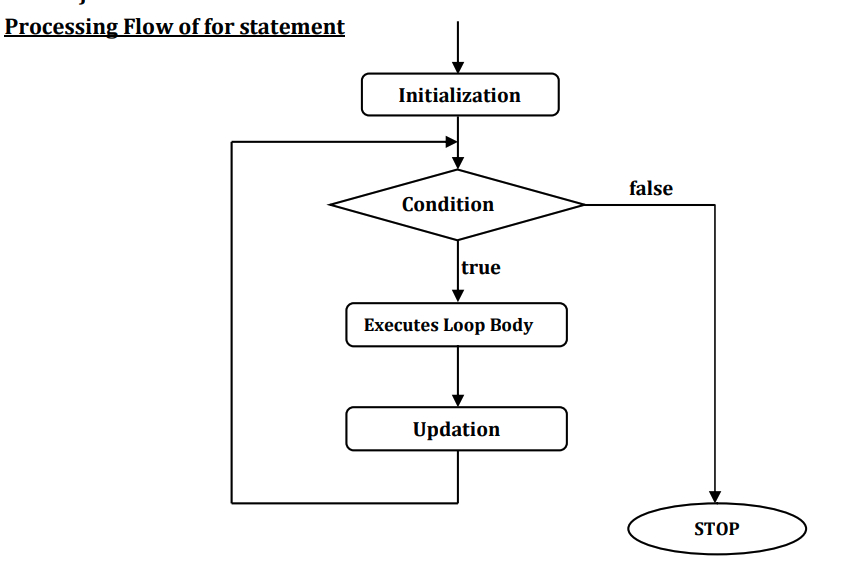
| x = 10 y = 20  if x < y:  print("x is less than y") elif x > y:  print("x is greater than y") else:  print("x is equal to y") |
| --- |

Problem to find the MAX three

| If a>b and a> c :   max=a; else if(b> a and b>c)  max = b; else  max=c; System.out.println("Max value is "+max); |
| --- |



#### For



**For Loop**  
  
A for loop in Python is used to iterate over a sequence (like a list, tuple, string, or range) or other iterable objects. Iterating over a sequence is called traversal.

| for i in range(5):  print(i) |
| --- |

range() and xrange() are both used to generate a sequence of numbers, but they have a key difference: range() returns a list, and xrange() returns an xrange object,

| # Python 3 for i in range(10): # Generates each number on-the-fly, more memory-efficient  print(i)  # If you really need a list for some reason: numbers = list(range(10)) |
| --- |

#### 

Python 3, xrange() was removed, and range() now behaves like xrange()

#### 

#### While

* condition of while statement is mandatory and must be boolean type



| i = 0 while i < 5:  print(i)  i += 1 |
| --- |

#### 

# Fibonacci series

#### 

#### **Break**

break is used to escape the loop when the condition is not met. Here's a concrete example.

Python does not have a built-in do-while loop structure that you might find in other programming languages like C++ or Java

#### 

| count = 0  while True:  count += 1  print(f"This will print at least once. Count = {count}")   if count >= 5:  break |
| --- |

| fruits = ['apple', 'banana', 'cherry']  for fruit in fruits:  print(fruit) |
| --- |

### 

#### **Continue**

The continue statement, also borrowed from C, continues with the next iteration of the loop:

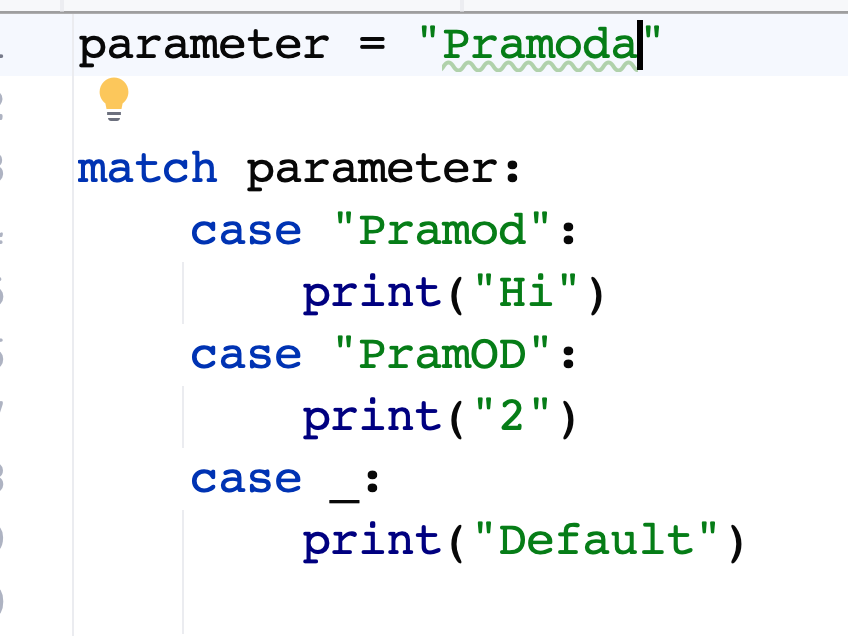
| for num in range(2, 10):  if num % 2 == 0:  print("Found an even number", num)  continue  print("Found an odd number", num) |
| --- |

pass Statements

The [pass](https://docs.python.org/3.11/reference/simple_stmts.html#pass) statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

#### **Match Statements**

* 3.10 Python
* Similar to the Switch Statment
* Mulitple -> If else loops



## ✅ Functions and Strings

* A function is a reusable set of operations.
* A function in Python is a block of organized, reusable code that is used to perform a specific task.
* Functions are defined in Python using the **def** keyword, followed by the function name and parentheses ().
* They may or may not return something.
* You have to call the function.
* Functions can take parameters, Return Values

| def greet():  print("Hello!") |
| --- |
|  |

greet() # Calls the function and prints "Hello!"

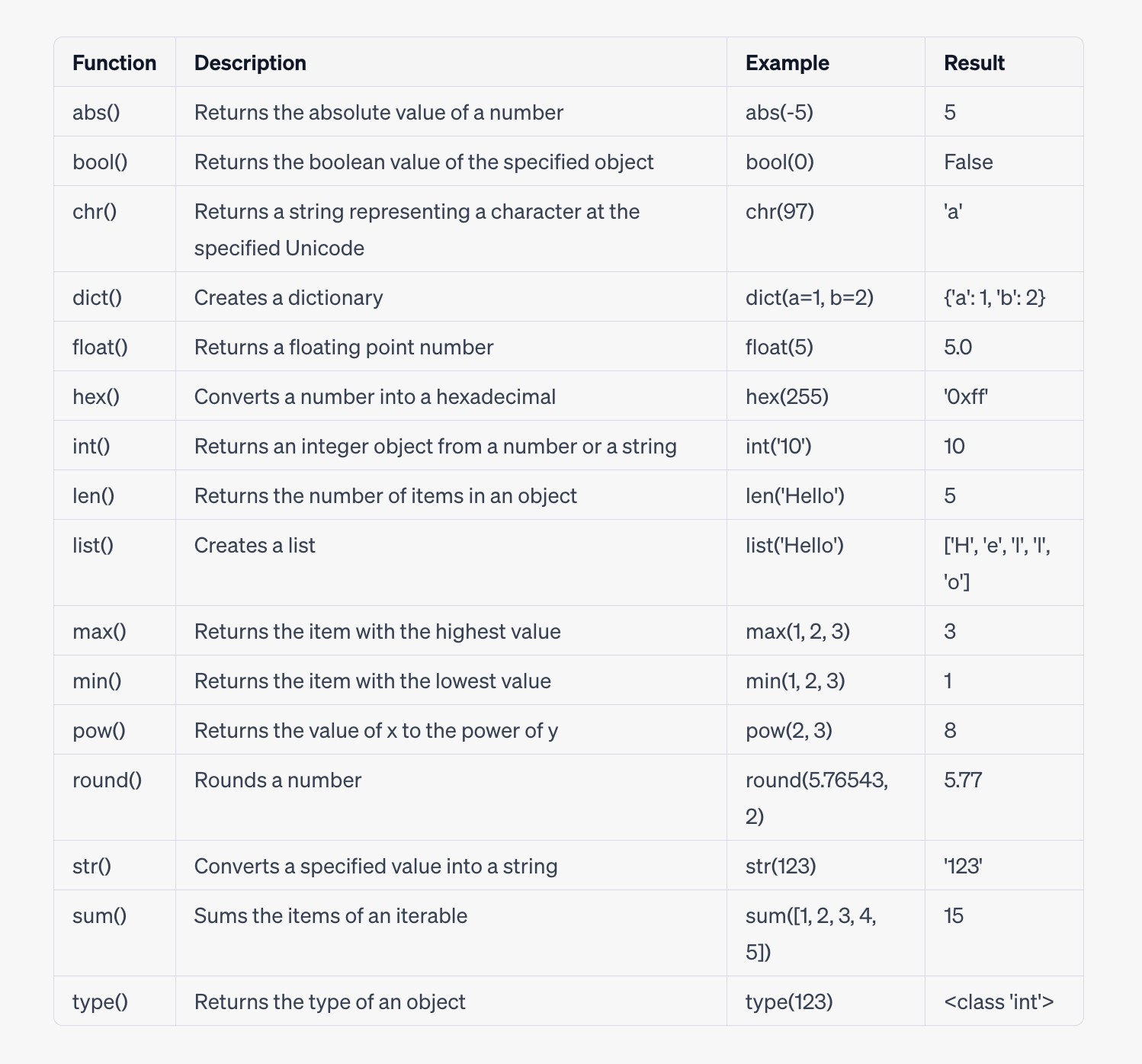
* If you want to repeat a task or delegate a task, reuse code you have to use the functions.

**Types of Functions in Python**

1. **Built-in functions**
2. **User-defined functions**

### **💡Built-in functions**

len(), min(), and print() are examples of built-in functions.



More details

<https://docs.python.org/3/library/functions.html>

## 

### 📕 **String Built In**

1. BO1.py

## 

## 

### 📕 **Components of a Function**

How do we actually make a function? In Python, a function can be defined using the **def keyword** in the following format:

The function name is simply the name we’ll use to identify the function.

The parameters of a function are the inputs for that function. We can use these inputs within the function. **Parameters are optional.**

## 

| def minimum(first, second):  if (first < second):  print(first)  else:  print(second) |
| --- |

## 

With Return

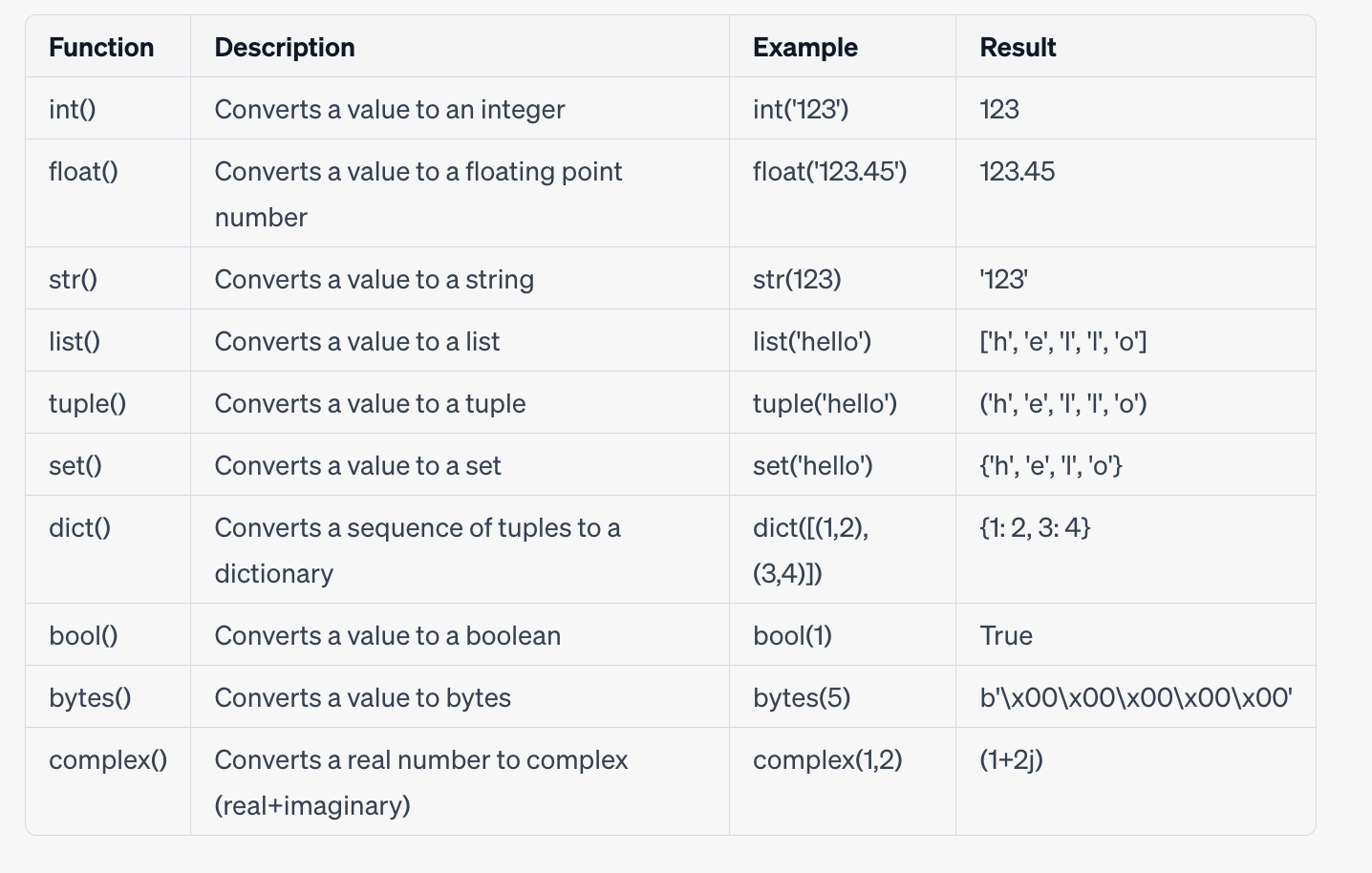
| def minimum(first, second):  if (first < second):  return first  return second |
| --- |

### 📕 Function Scope

1. Normal Data variables
2. Alterdata within (List)

## 

## ✅ **Type Conversions**



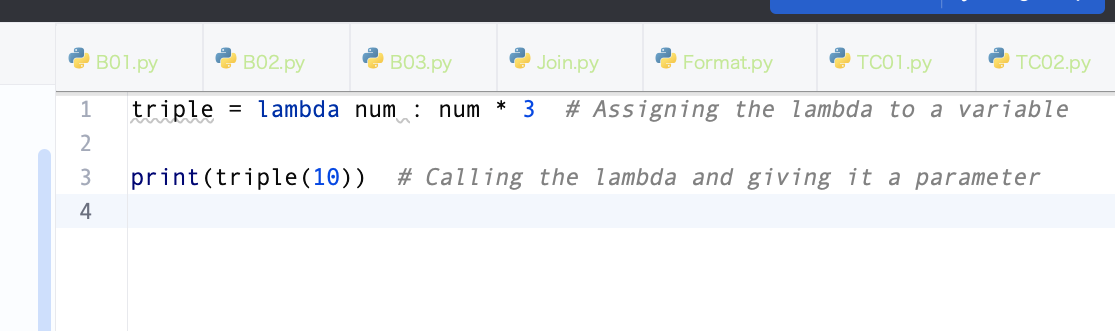
## 

## 

## ✅ Lambda



* A lambda is an anonymous function that returns some form of data.
* **Syntax:** The syntax to create a lambda function is lambda arguments: expression. The lambda keyword is used to define the anonymous function, followed by a list of arguments, a colon, and an expression.
* **Arguments**: Like a normal function, a lambda function can accept any number of arguments but must have only one expression. The arguments are specified before the colon.
* **Expression**: The expression is executed and the result is returned when the lambda function is called. This expression is written after the colon.
* **Return Value**: A lambda function can have a return value. The expression is evaluated and returned when the function is called.



## 

## 

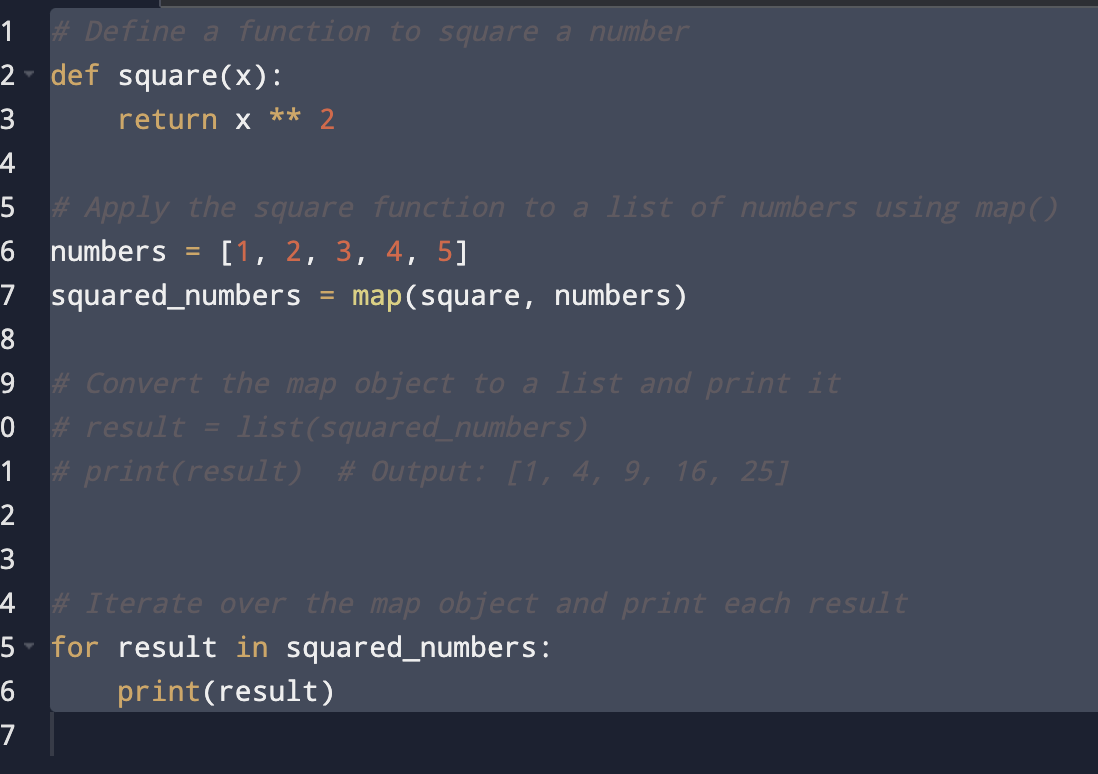
lambdas are really useful when a function requires another function as its argument.

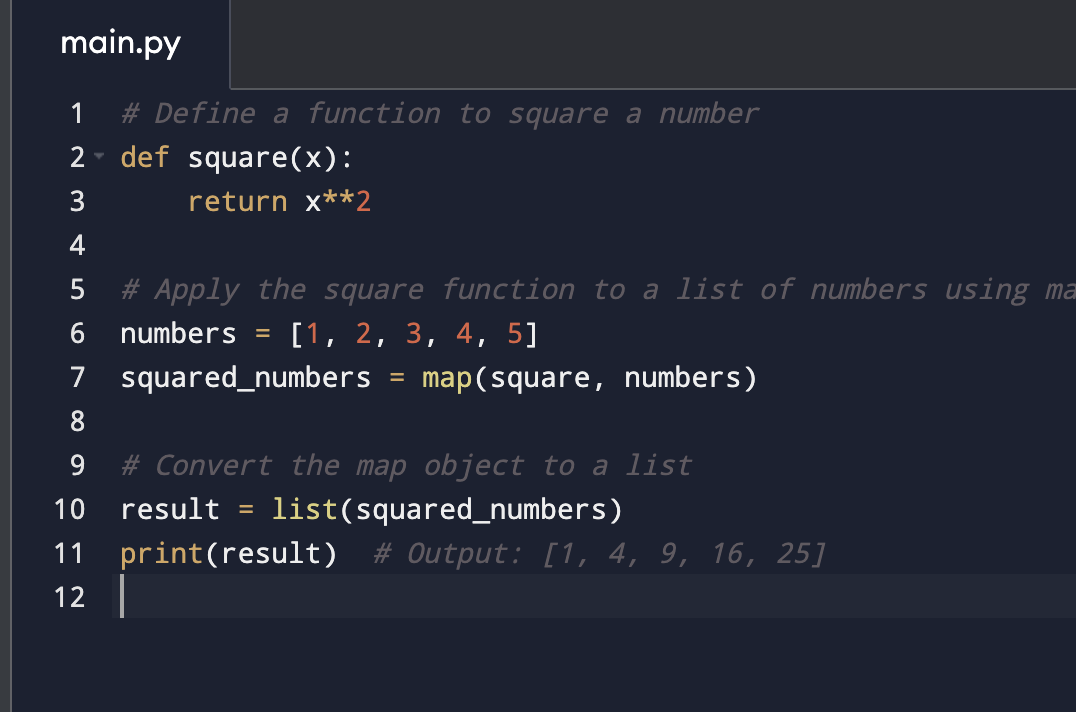
## ✅ Map and Filters

Map() Functions

1. Python is a built-in function.
2. Applies a given function to each item of an iterable (such as a list, tuple, or string) and returns an iterator with the results.
3. map() function is often used when you need to transform each element of an iterable using a specific function and collect the results

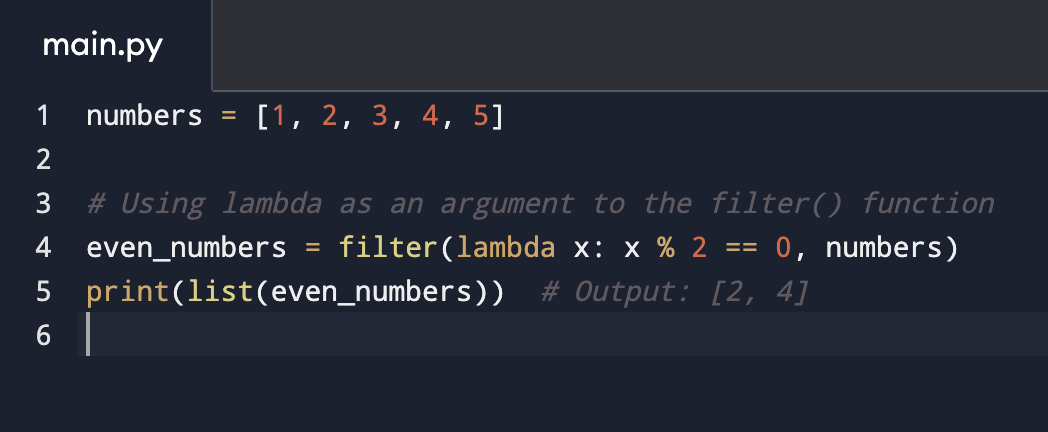
map(function, iterable)

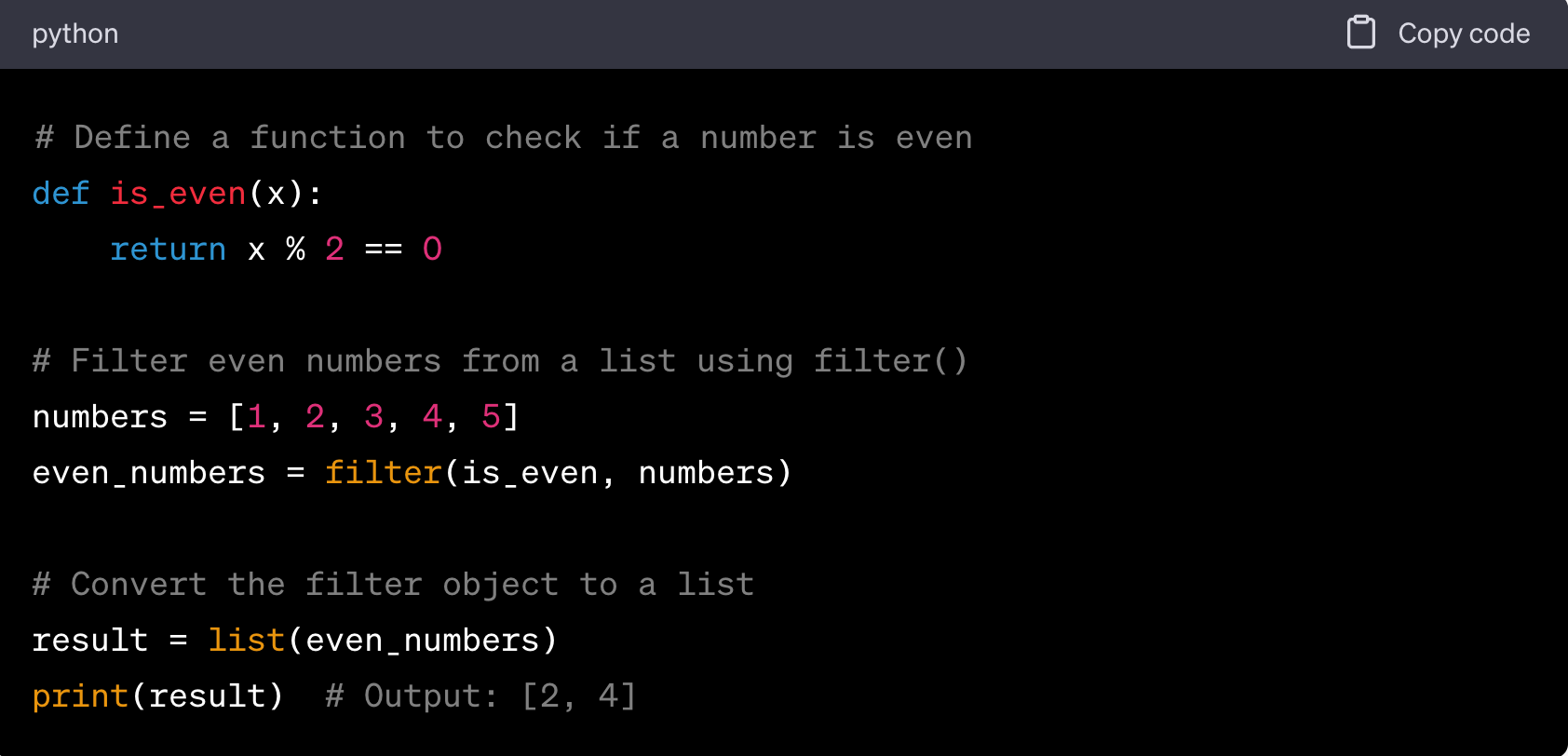




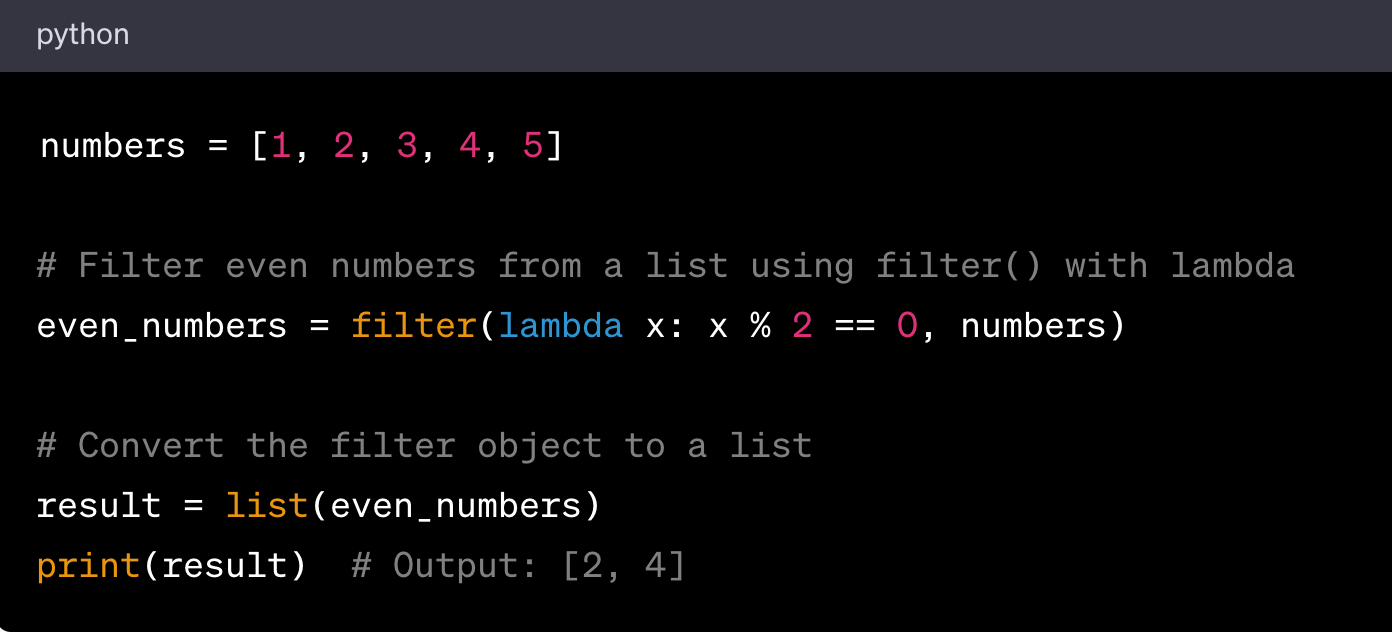
##### 

##### **Filter**





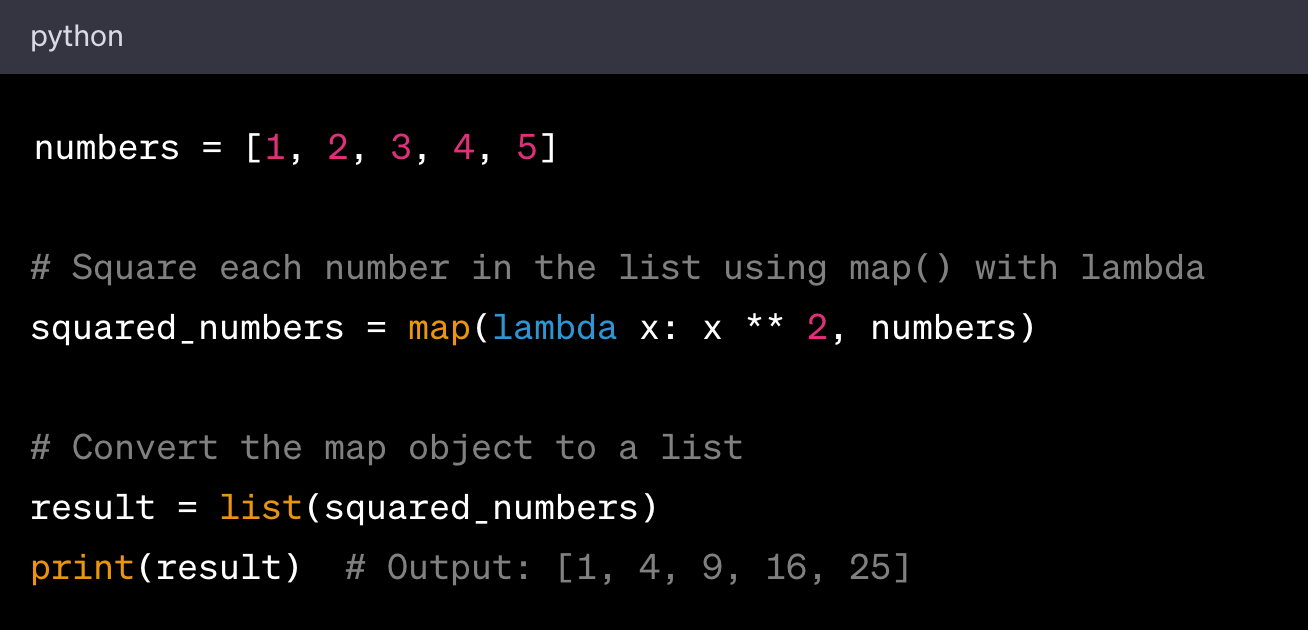
**Using lambda as argument**



##### Functions as Arguments

1. Using lambda as argument
2. Using map lambda

**Using map lambda**



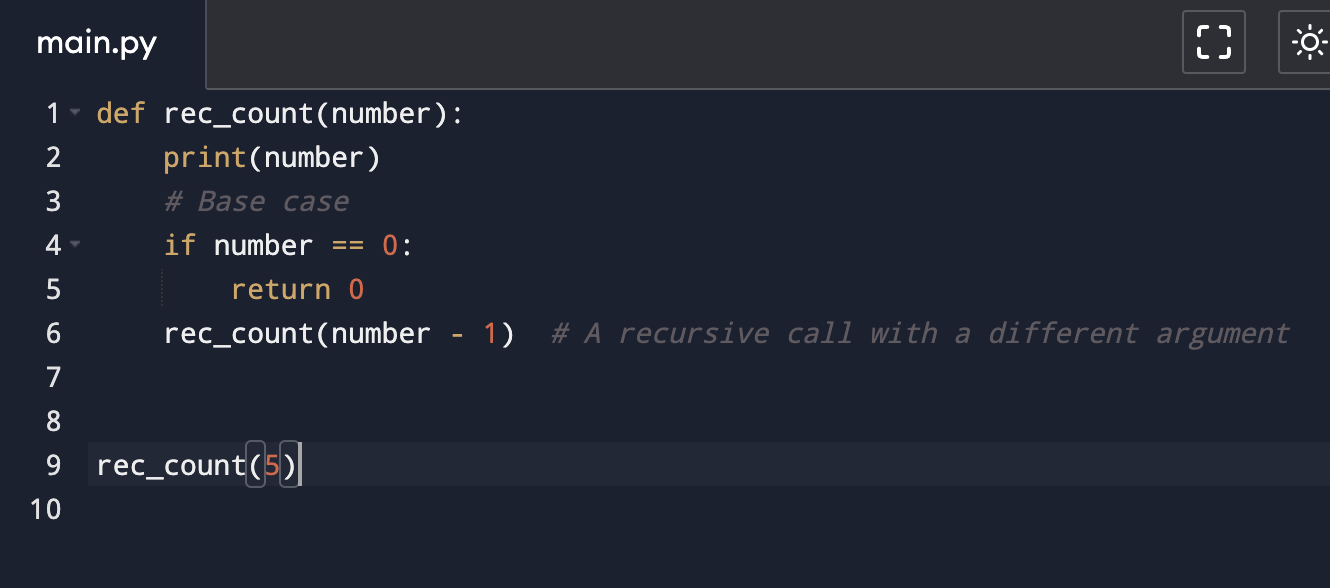
**Default Parameter**

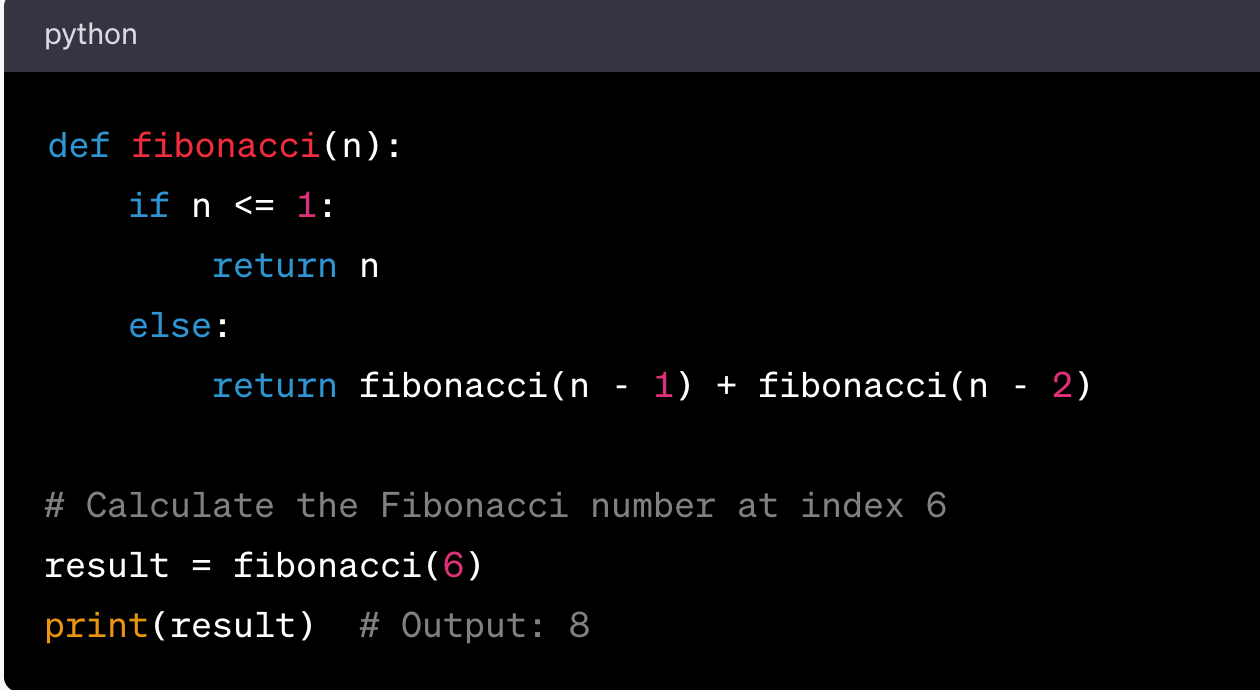
* You can provide default values for parameters, making them optional when calling the function.
* If a caller doesn't provide a value for an optional parameter, the default value is used.

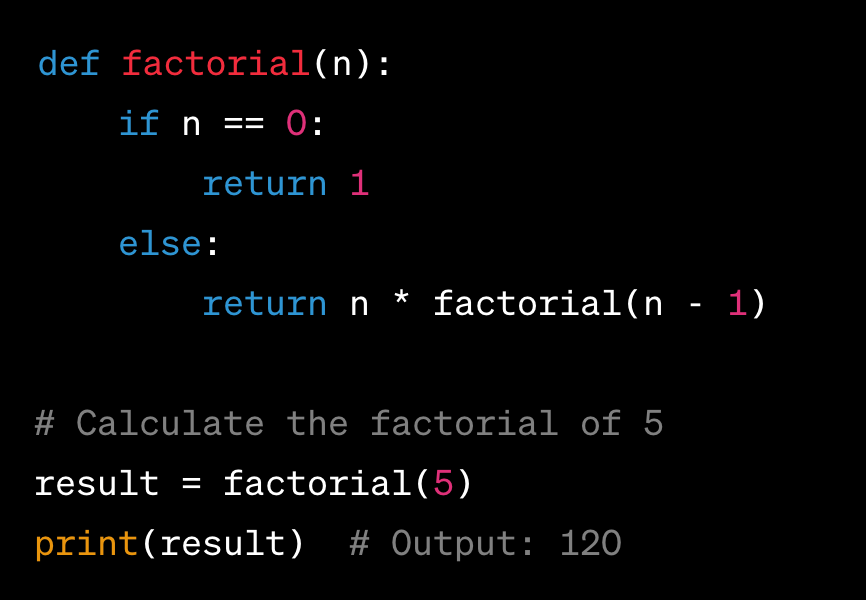


## **✅ Recursion**

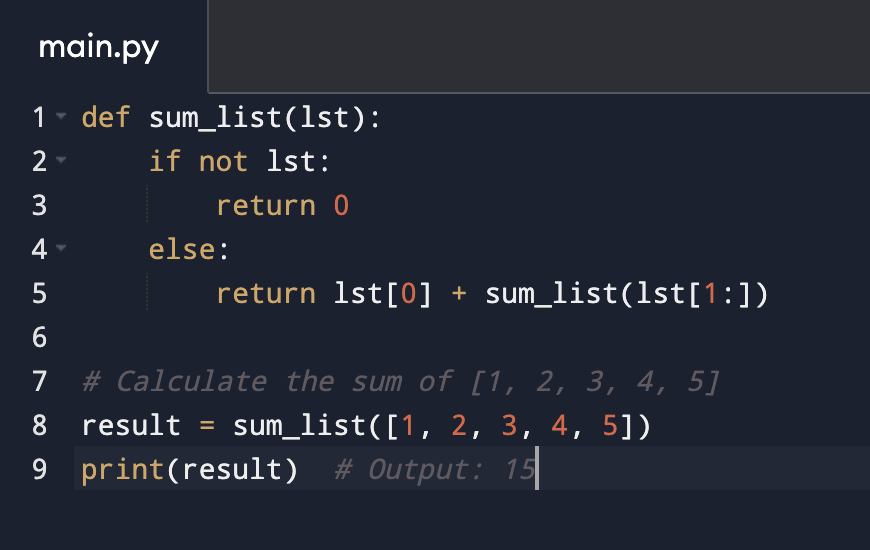
* Recursion is the process in which a function calls itself during its execution.
* Each recursive call takes the program one scope deeper into the function.
* The recursive calls stop at the base case. The base case is a check used to indicate that there should be no further recursion.





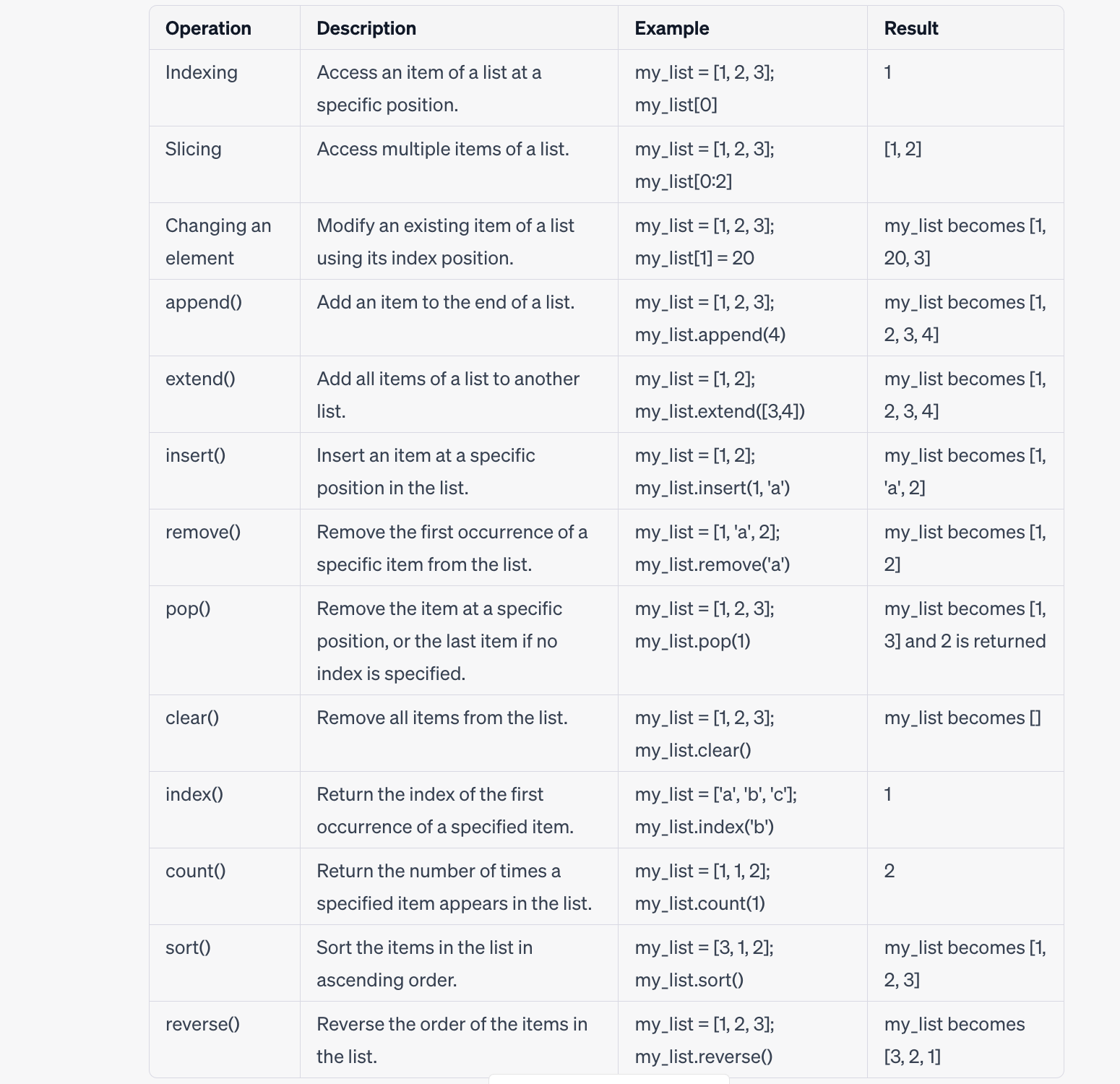


Sum of List: Calculate the sum of all elements in a list using recursion.



## ✅ List []

* It allows us to store elements of different data types in one container.
* A list is a collection of items that are ordered and changeable (mutable). It allows duplicate members.
* **Creation**: Lists are created by placing a comma-separated sequence of items inside square brackets [].
  + fruits = ['apple', 'banana', 'cherry']
* **Insert** - aList.insert(index, newElement)
* **Access Items**: You can access items in a list by referring to their index number. Indexes start from 0.
  + print(fruits[0]) # Output: 'apple'
  + fruits[1] = 'blueberry' # change value
  + print(len(fruits)) # Output: 3
  + fruits.append('dragonfruit')
  + fruits.remove('blueberry')
  + mixed\_list = ['apple', 1, True]
* nested\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
* List Creation
  + squares = [i\*\*2 for i in range(10)]
  + print(squares) # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
* num\_seq = range(0, 10) # A sequence from 0 to 9
* num\_list = list(num\_seq)



**Problems**

1. Write a Python program to find the largest number in a list.
2. Write a Python program to find the smallest number in a list.
3. Write a Python program to sum all numbers in a list.
4. Write a Python program to multiply all numbers in a list.
5. Write a Python program to count the number of strings in a list where the string length is 2 or more and the first and last characters are the same.
6. Write a Python program that takes two lists and returns True if they have at least one common member.
7. Write a Python program that prints all the numbers from 0 to 6 except 3 and 6.
8. Write a Python program to get the Fibonacci series between 0 to 50.
9. Write a Python program to find the factorial of a number.
10. Write a Python program to check if a number is a prime number.

### ✅ Multi-dimensional lists in Python

* Multi-dimensional lists are the lists within lists.
* A dictionary will be the better choice rather than a multi-dimensional list in Python.
* Operations with the Multi Dimensional
  + append
  + extend
  + reverse()

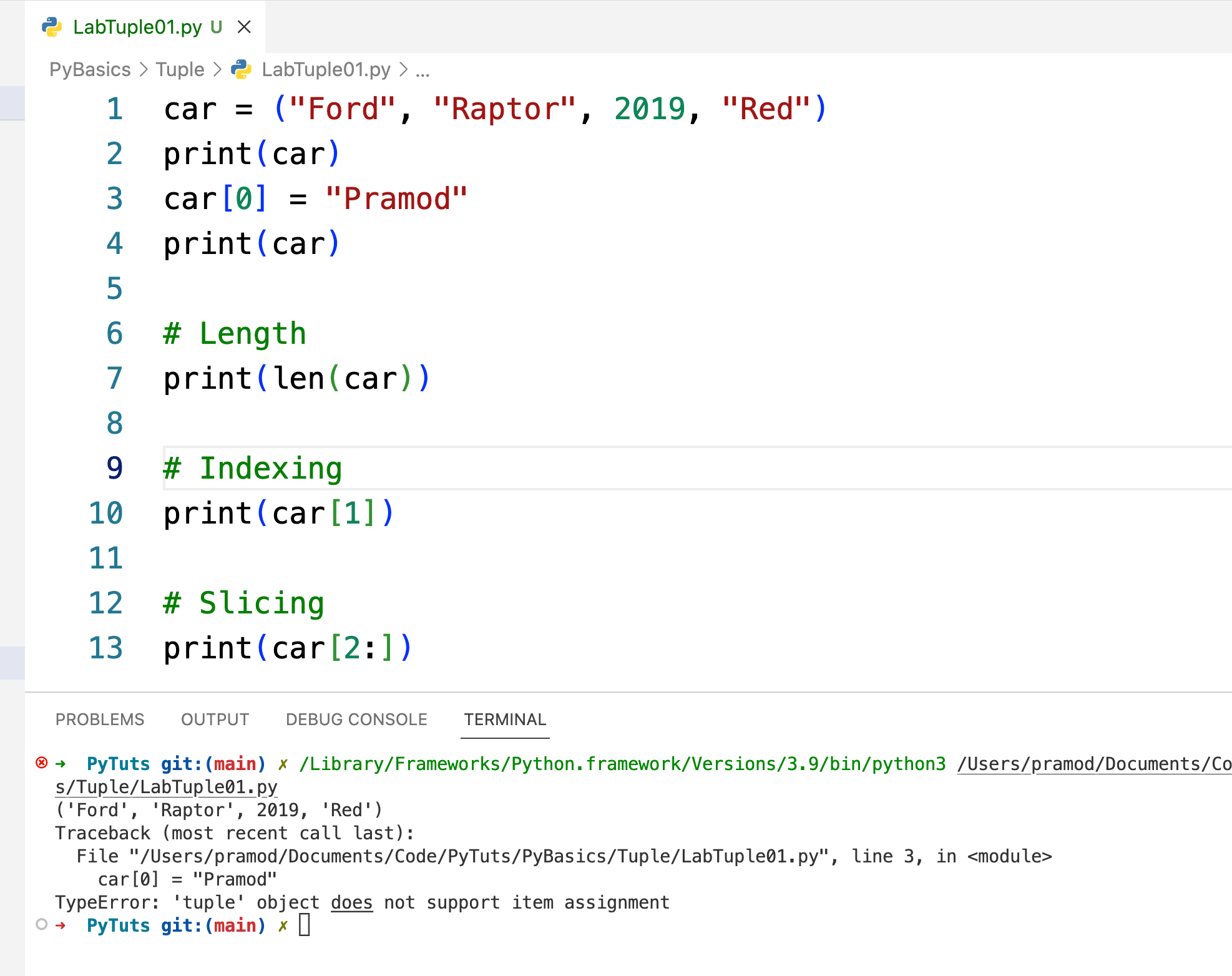
Creating a multidimensional list with all zeros

**Practice Question**

1. How to check if a list is empty in Python. (if len(lis1) == 0:)
2. Remove duplicates from list.
3. Numbers in a list within a given range.
4. Check if two lists are identical.
5. Check for Sublist in List

## ✅ Tuple ()

* A tuple is very similar to a list, except for the fact that its contents cannot be changed.
* A tuple is immutable.
* The contents of a tuple are enclosed in parentheses, (). They are also ordered, and hence, follow the linear index notation.



Merging Tuples, Deleting and More Examples.

Here's a bullet-point list of the functions and methods you can use with tuples:

- **Creation of tuple**: Creating a new tuple with parentheses `()`.

- **Accessing Elements**: Accessing tuple elements using an index.

- **len():** Returns the number of elements in the tuple.

- **min()` and `max()`:** Returns the smallest and largest elements in the tuple, respectively.

**- index()`:** Returns the first index at which a given element appears in the tuple.

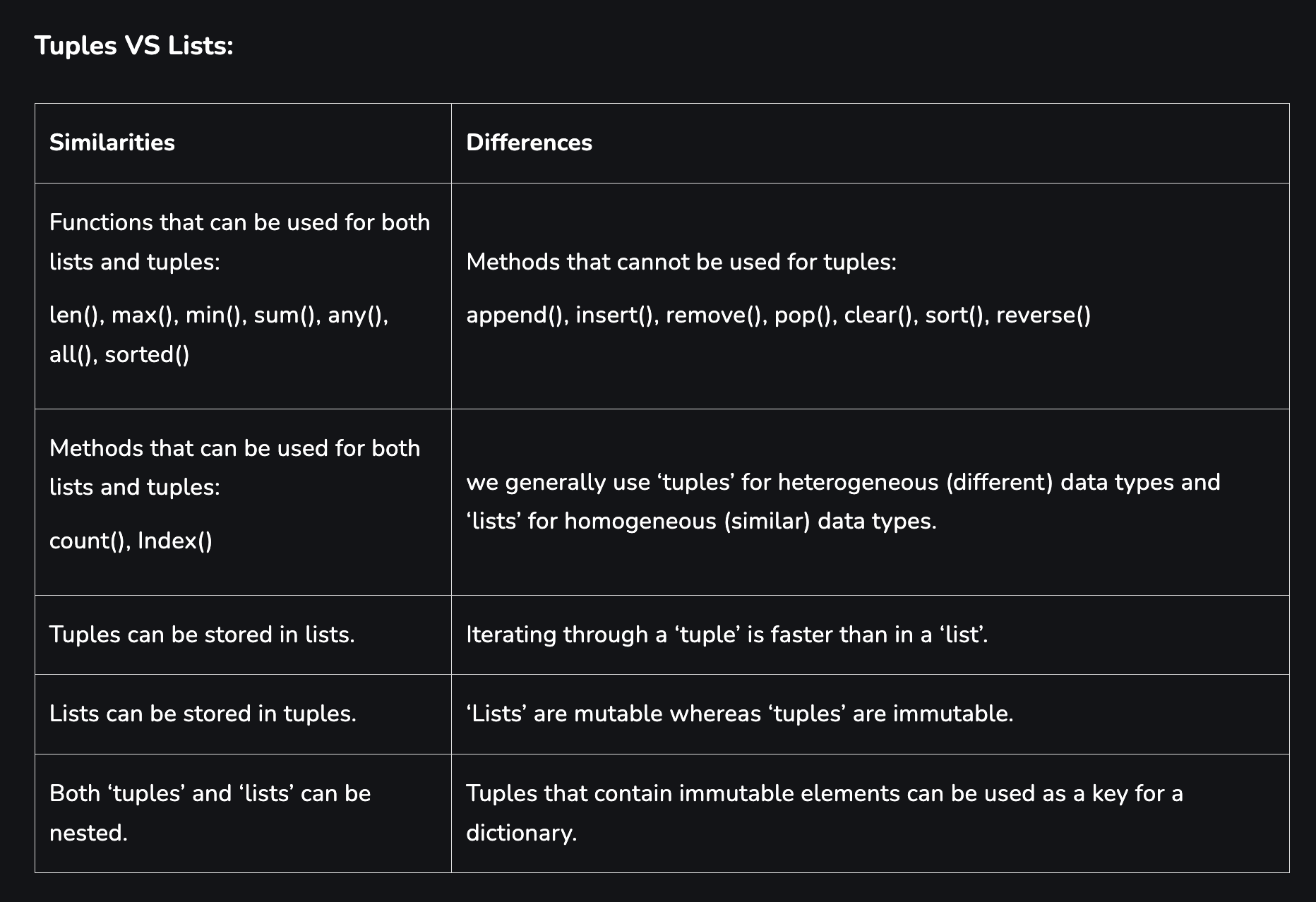
**- count():** Returns the number of times a specified value occurs in a tuple.

- **Concatenation:** Using the `+` operator to add tuples together.

- **Replication**: Using the `\*` operator to repeat the contents of a tuple a given number of times.

- Tuple Unpacking: Extracting the values from a tuple back into variables.

- Membership: Using the `in` keyword to check if an element exists in a tuple.



## ✅ Set ()

a Set is an unordered collection of data types that is iterable, mutable and has no duplicate elements.

* # Creating a Set with a List of Numbers
* # Creating a Set
* # Adding element and tuple to the Set.
* # Access Set
* # Removing elements from Set.
* # Deletion of elements in a Set.
* Frozen sets
  + Python frozen sets are immutable and have methods/operators that don't modify the set.
* Union
* Intersection.
* Diff, SubSet

## ✅ Dictionary (dict{})

## Key and Value Pair.

* A dictionary stores key-value pairs, where each unique key is an index which holds the value associated with it.
* Dictionaries are unordered because the entries are not stored in a linear structure.

1. Creating a Dictionary.
2. Accessing Values.
3. Adding/Updating Entries.
4. Removing Entries.
5. Length of a Dictionary.
6. Checking Key Existence.
7. Copying Contents

## 

## ✅ OOPs : **Object Oriented Programming**

Objects are the main building blocks of OOPS i.e your applications will be divided into multiple objects.

A class with

Data members

Methods -

**OOPs concepts apply now.**

o Abstraction

o Encapsulation

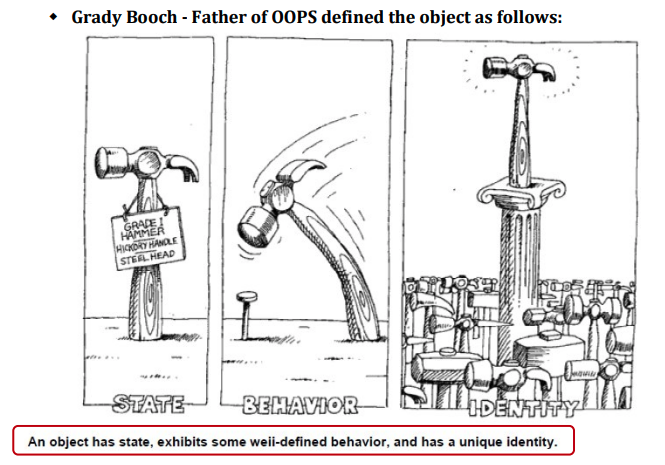
o Inheritance

o Polymorphism

Everything in the world is an object.

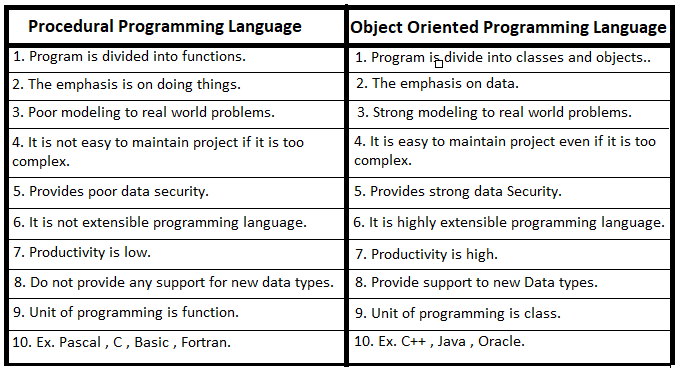
OOPs vs POP or PPL (Procedural oriented programming)

Object vs Functional.



**Objects are a collection of data and their behaviors.**

A class can be thought of as a blueprint for creating objects.



**How to Use Oops to Build Any Modern-Day Software**

Consider this scenario: We are developing an Automation Tester Batch System.

1. Identify the objects
   1. Like Student, Course, Payment.
2. Describe those with details
   1. Data - Student -> name, id, age, address, email, course taken
   2. Operations -> addStudent, deleteStudent
   3. Bind them with Encapsulation (Class)
3. Establish the relationship with each other
   1. Student -> payment, Student -> Course, Course -> Payment
4. Now implement it via Class and Objects.

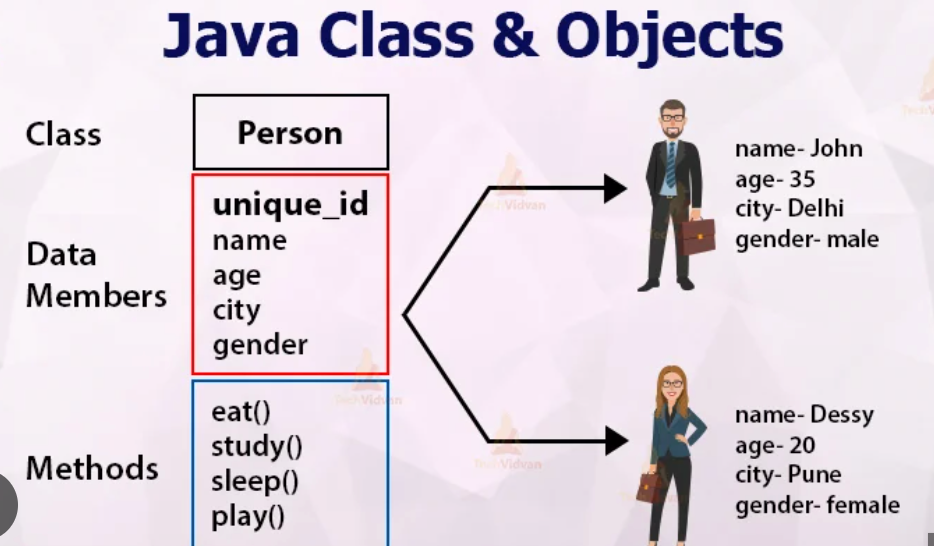
##### **Class and Objects**

a class is a blueprint or template that defines the attributes (variables) and behaviors (methods) that objects of that class will have.

An object is an instance of a class, created using the class as a template.

Objects have their own unique state and can access the attributes and behaviors defined in the class.

Classes provide a way to organize and structure code, enabling code reusability and modular design.



| class Car:  def \_\_init\_\_(self, brand, color):  self.brand = brand  self.color = color    def start\_engine(self):  print("Engine started!")    def drive(self):  print(f"Driving the {self.color} {self.brand} car.")  my\_car = Car("Toyota", "blue") my\_car.start\_engine() my\_car.drive() |
| --- |

##### 

##### **Constructor**

* a constructor is a special method that is automatically called when an object is created from a class.
* It is used to initialize the object's attributes or perform any necessary setup tasks.
* The constructor method in Python is called \_\_init\_\_().

##### 

| class Person:  def \_\_init\_\_(self, name, age):  self.name = name  self.age = age def display(self):  print(f"Name: {self.name}, Age: {self.age}")  # Creating objects and invoking the constructor person1 = Person("Alice", 25) person2 = Person("Bob", 30)  # Accessing object attributes print(person1.name) # Output: Alice print(person2.age) # Output: 30  # Invoking object method person1.display() # Output: Name: Alice, Age: 25 person2.display() # Output: Name: Bob, Age: 30 |
| --- |

##### 

##### 

##### **Encapsulation**

* Refers to the bundling of data and methods that operate on that data within a single unit, or object.
* Encapsulation helps to promote the principle of "data hiding".
* This is achieved by declaring the object's data fields as private and providing public getter and setter methods to access and modify the data.

| class BankAccount:  def \_\_init\_\_(self):  self.balance = 0 # Public attribute    def deposit(self, amount):  self.balance += amount    def \_withdraw(self, amount):  self.balance -= amount # Protected attribute    def \_\_show\_balance(self):  print(f"Account balance: {self.balance}") # Private attribute  account = BankAccount() account.deposit(1000) account.\_withdraw(500) # Protected attribute accessed account.\_\_show\_balance() # Error: private attribute not accessible |
| --- |

###### 

###### 

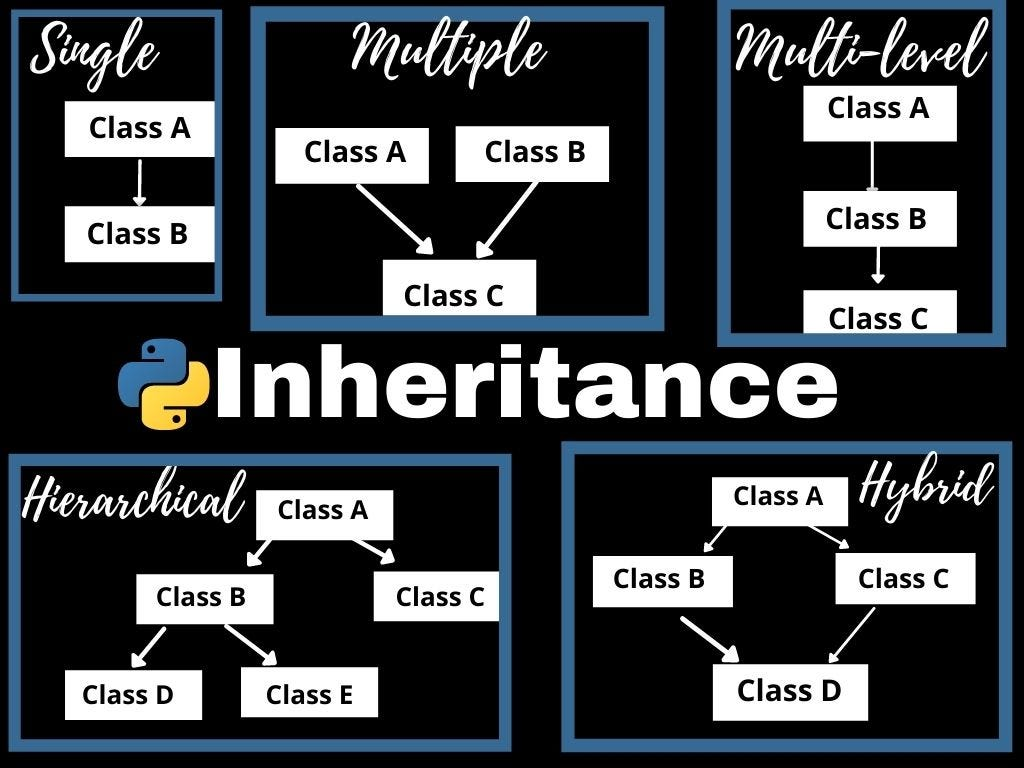
##### Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass.

It enables flexibility and dynamic behavior based on the actual object type. Here's an example:

| class Shape:  def area(self):  pass  class Rectangle(Shape):  def \_\_init\_\_(self, length, width):  self.length = length  self.width = width    def area(self):  return self.length \* self.width  class Circle(Shape):  def \_\_init\_\_(self, radius):  self.radius = radius    def area(self):  return 3.14 \* self.radius \* self.radius  shapes = [Rectangle(4, 5), Circle(3)] for shape in shapes:  print(shape.area()) |
| --- |

##### **Inheritance**



In Python, the different types of inheritance include:

1. **Single Inheritance**: A class inherits from a single base class.

2. **Multiple Inheritance**: A class inherits from multiple base classes.

3. **Multilevel Inheritance**: A class inherits from a derived class, forming a hierarchy of inheritance.

4. **Hierarchical Inheritance**: Multiple derived classes inherit from a single base class.

5. **Hybrid Inheritance**: A combination of multiple inheritance and multilevel inheritance.

6. **Abstract Base Classes (ABCs)**: Classes that define abstract methods and serve as a blueprint for derived classes.

7. **Method Resolution Order (MRO)**: The order in which base classes are searched for a particular attribute or method.

**Single Inheritance:**  
Single inheritance involves a class inheriting from a single base class.

| class Animal:  def speak(self):  print("Animal speaks!")  class Dog(Animal):  def bark(self):  print("Woof!")  my\_dog = Dog() my\_dog.speak() # Inherited from Animal my\_dog.bark() # Specific to Dog |
| --- |

**Multiple Inheritance:**

Multiple inheritance allows a class to inherit from multiple base classes.

| class A:  def method\_a(self):  print("Method A")  class B:  def method\_b(self):  print("Method B")  class C(A, B):  def method\_c(self):  print("Method C")  my\_object = C() my\_object.method\_a() # Inherited from A my\_object.method\_b() # Inherited from B my\_object.method\_c() # Specific to C |
| --- |

MRO - Handle Case -

| class A:  def greet(self):  print("Hello from class A")  class B:  def greet(self):  print("Hello from class B")  class C(A, B):  pass  class D(B, A):  pass  obj1 = C() obj1.greet()  obj2 = D() obj2.greet()  print(C.mro()) print(D.mro()) |
| --- |

**Multilevel Inheritance**

Multilevel inheritance involves a class inheriting from a derived class.

| class Vehicle:  def start\_engine(self):  print("Engine started!")  class Car(Vehicle):  def drive(self):  print("Driving the car!")  class SportsCar(Car):  def race(self):  print("Racing the sports car!")  my\_car = SportsCar() my\_car.start\_engine() # Inherited from Vehicle my\_car.drive() # Inherited from Car my\_car.race() # Specific to SportsCar |
| --- |

**Hierarchical Inheritance**

Hierarchical inheritance involves multiple derived classes inheriting from a single base class.

| class Animal:  def speak(self):  print("Animal speaks!")  class Dog(Animal):  def bark(self):  print("Woof!")  class Cat(Animal):  def meow(self):  print("Meow!")  my\_dog = Dog() my\_dog.speak() # Inherited from Animal my\_dog.bark() # Specific to Dog  my\_cat = Cat() my\_cat.speak() # Inherited from Animal my\_cat.meow() # Specific to Cat |
| --- |

##### Abstraction

Abstraction is a fundamental concept in object-oriented programming (OOP) that allows us to represent complex systems by simplifying and hiding unnecessary details

**Abstract Base Class (ABC):**

* An abstract base class is a class that cannot be instantiated and is meant to be subclassed.
* It serves as a blueprint for derived classes and defines a common interface or set of methods that derived classes must implement.
* In Python, ABCs are created using the ABC metaclass from the abc module.

**Abstract Method**

An abstract method is a method declared within an abstract base class that does not have an implementation.

It serves as a placeholder for the derived classes to provide their own implementation.

Abstract methods are defined using the @abstractmethod decorator.

| from abc import ABC, abstractmethod  class Shape(ABC):  @abstractmethod  def area(self):  pass   @abstractmethod  def perimeter(self):  pass  class Rectangle(Shape):  def \_\_init\_\_(self, length, width):  self.length = length  self.width = width    def area(self):  return self.length \* self.width    def perimeter(self):  return 2 \* (self.length + self.width)  class Circle(Shape):  def \_\_init\_\_(self, radius):  self.radius = radius    def area(self):  return 3.14 \* self.radius \* self.radius    def perimeter(self):  return 2 \* 3.14 \* self.radius  rect = Rectangle(4, 5) print(rect.area()) # Output: 20 print(rect.perimeter()) # Output: 18  circle = Circle(3) print(circle.area()) # Output: 28.26 print(circle.perimeter()) # Output: 18.84 |
| --- |

## 📕 Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

| # Example 1: Division by zero exception try:  result = 10 / 0 # Attempting to divide by zero except ZeroDivisionError as error:  print("Error:", error) |
| --- |

there are several built-in exception classes available to handle different types of errors or exceptional situations that may occur during program execution.

* **Exception:** The base class for all built-in exceptions.
* **SyntaxError:** Raised when there is a syntax error in the code.
* **IndentationError:** Raised when there is an indentation-related error, such as incorrect or inconsistent indentation.
* **NameError:** Raised when a local or global name is not found.
* **TypeError:** Raised when an operation or function is performed on an object of an inappropriate type.
* **ValueError:** Raised when a function receives an argument of the correct type but an invalid value.
* **ZeroDivisionError:** Raised when division or modulo by zero is encountered.
* **IndexError:** Raised when a sequence subscript is out of range.
* **KeyError:** Raised when a dictionary key is not found.
* **FileNotFoundError:** Raised when a file or directory is requested but cannot be found.
* **IOError:** Raised when an input/output operation fails.
* **ImportError:** Raised when an import statement fails to find and load a module.
* **AttributeError:** Raised when an attribute reference or assignment fails.
* **RuntimeError:** Raised when an error occurs that doesn't belong to any specific category.

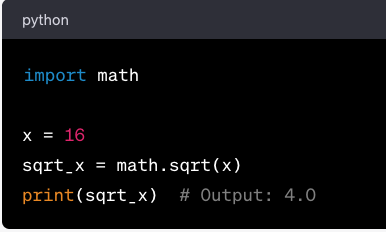
StopIteration: Raised by iterator objects to signal the end of iteration.

KeyboardInterrupt: Raised when the user interrupts the execution of the program by pressing Ctrl+C.

| try:  # Code that may raise an exception  # ... except ExceptionType1:  # Handler for ExceptionType1  # ... except ExceptionType2:  # Handler for ExceptionType2  # ... else:  # Executed if no exceptions are raised  # ... finally:  # Always executed, whether an exception occurs or not  # ... |
| --- |

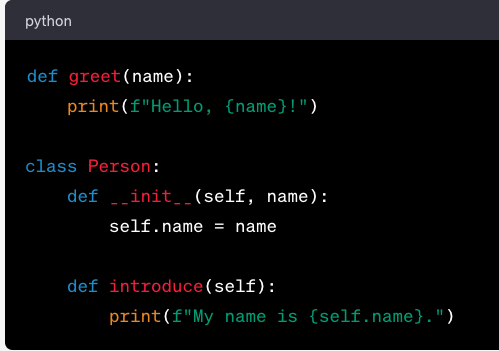
## ⚓ Modules in Python

* A module is a file that contains Python code, usually with a specific functionality or set of related functionalities.
* Modules allow you to organize your code into reusable units, making it easier to maintain, share, and reuse code across different projects.
* You can import modules into your Python scripts or interactive sessions to access the functions, classes, and variables defined in them.
* Built In Modules
* Importing Specific Items from a Module



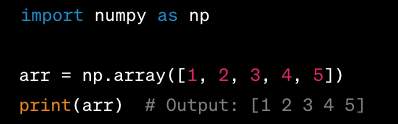
| from math import pi, sin  angle = pi / 2 sin\_angle = sin(angle) print(sin\_angle) # Output: 1.0 |
| --- |

* Custom Modules - my\_module.py



| import my\_module  my\_module.greet("Alice") # Output: Hello, Alice!  person = my\_module.Person("Bob") person.introduce() # Output: My name is Bob. |
| --- |

* Aliasing Modules



* Module Packages

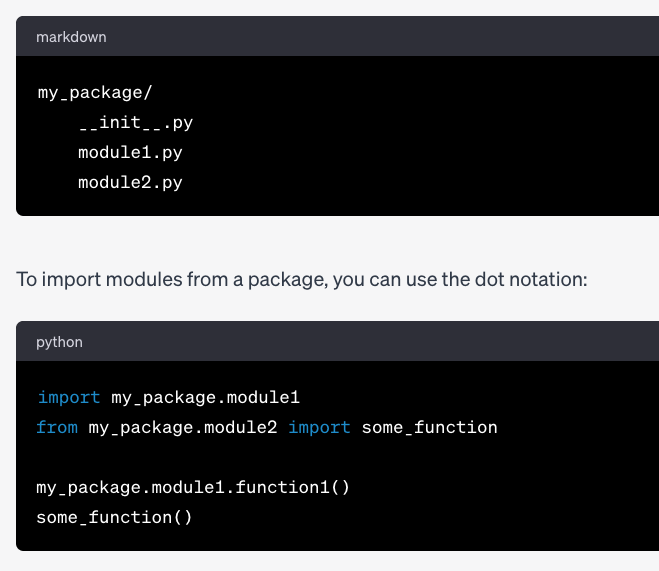
A package is simply a directory containing Python module files and a special \_\_init\_\_.py file.

The \_\_init\_\_.py file can be empty or can contain initialization code for the package. Packages allow for logical grouping and prevent naming conflicts between modules.

By including an \_\_init\_\_.py file in a package, you ensure that any necessary initialization code is executed when the package is imported

It Contains

| VERSION = '1.0.0' AUTHOR = 'John Doe' |
| --- |

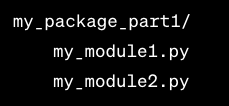


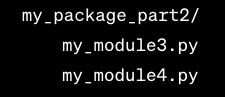
Using Third-Party Modules

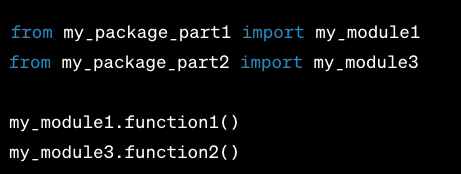
| import requests  response = requests.get('https://www.sdet.live/become') print(response.status\_code) # Output: 200 |
| --- |

Namespace packages

Namespace packages are useful when a package must be distributed to different places or when multiple teams work on different parts of the package.







## ☎️ Collections in Python

Focus on Main Business Logic rather than Low Level Logics

* Counters
* OrderedDict

Counter: A Counter is a dictionary subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values.

| from collections import Counter cnt = Counter() for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:  cnt[word] += 1 print(cnt) # Output: Counter({'blue': 3, 'red': 2, 'green': 1}) |
| --- |

OrderedDict: An OrderedDict is a dictionary subclass that remembers the order that keys were first inserted. The only difference between dict and OrderedDict is that:

| from collections import OrderedDict d = OrderedDict() d['a'] = 'A' d['b'] = 'B' d['c'] = 'C' for key, value in d.items():  print(key, value) # Output: # a A # b B # c C |
| --- |

### 