



Python Basics

Why should you learn Python?

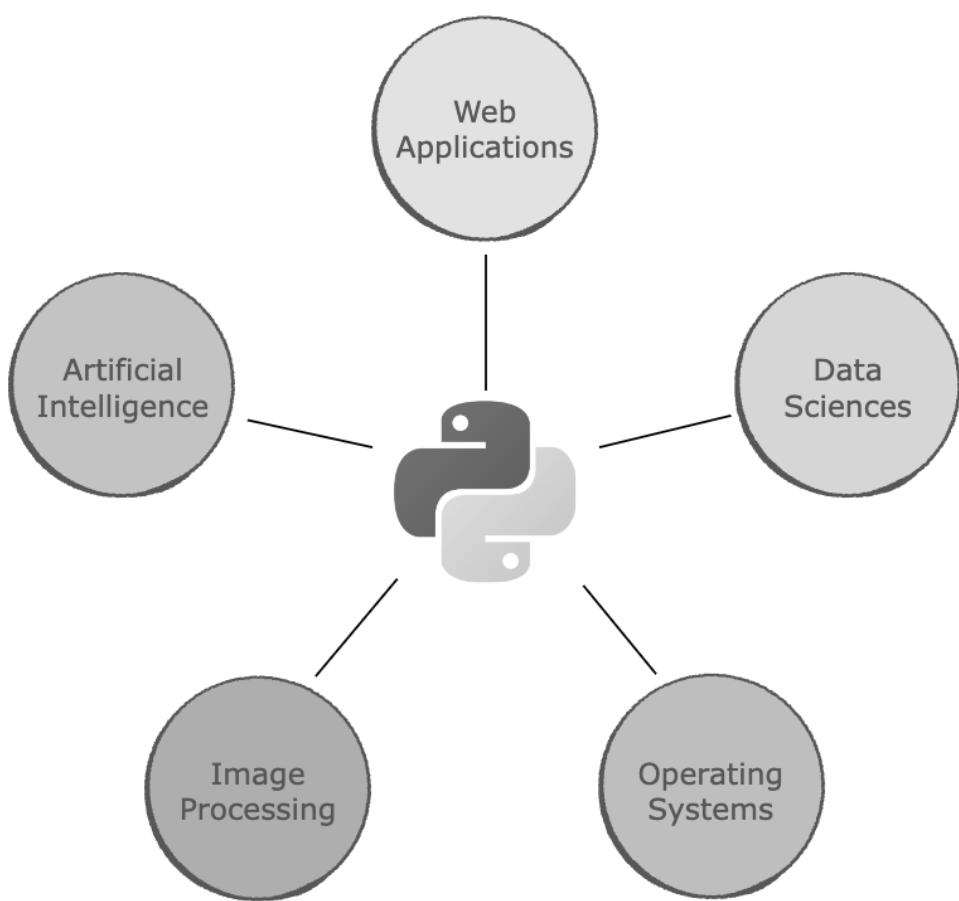
- Easy to learn: Python has a simple and readable syntax
- Large community massive and supportive community of programmers.
- High demand : Python is in high demand in the job market.

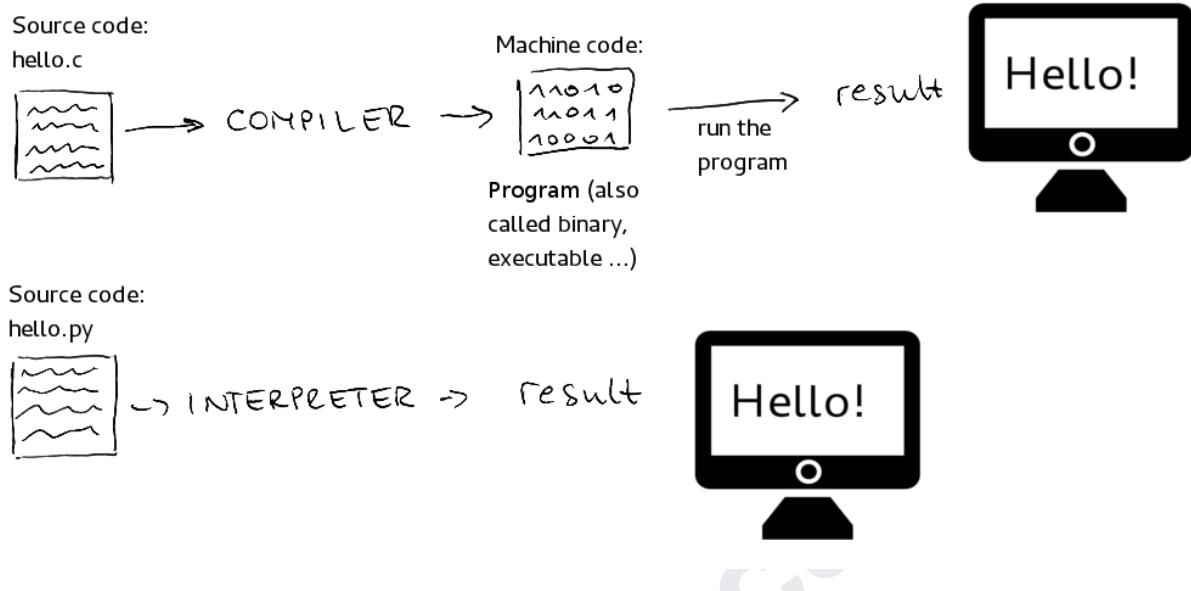
What is Python?

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 Sensitive

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'>
```

TheTestingAcademy

✓ Python 2.7 Vs 3.x ?

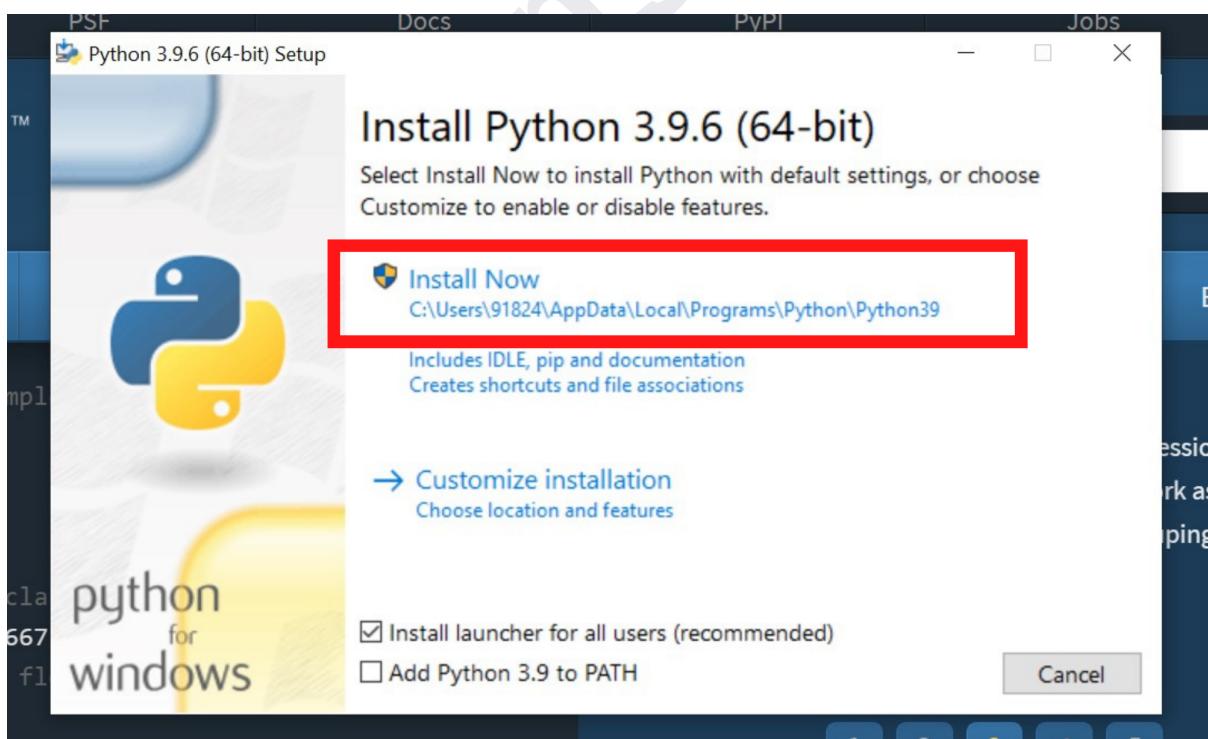
Feature	Python 2.7	Python 3
Release Date	July 3, 2010	December 3, 2008
End of Life	January 1, 2020	Ongoing (latest: 3.10.2)
Print function	<code>print "Hello, World!"`</code>	<code>`print("Hello, World!")`</code>
Integer Division	<code>`5 / 2 = 2`</code>	<code>`5 / 2 = 2.5`</code>
Unicode support	Limited, using <code>`u"..."`</code>	Native, all strings are Unicode
xrange function	<code>`xrange()`</code>	Replaced by <code>`range()`</code>
Exception Handling	<code>`except Exception, e:`</code>	<code>`except Exception as e:`</code>
Syntax for raising errors	<code>`raise ValueError, "error message"`</code>	<code>`raise ValueError("error message")`</code>
Dict comprehensions	Not built-in	Built-in
Ordering Comparisons	Comparisons like <code>`1 < 2`</code> allowed	Not allowed, requires explicit type conversion
Round function	<code>`round(0.5) = 1.0`</code>	<code>`round(0.5) = 0`</code>

👉 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)
```

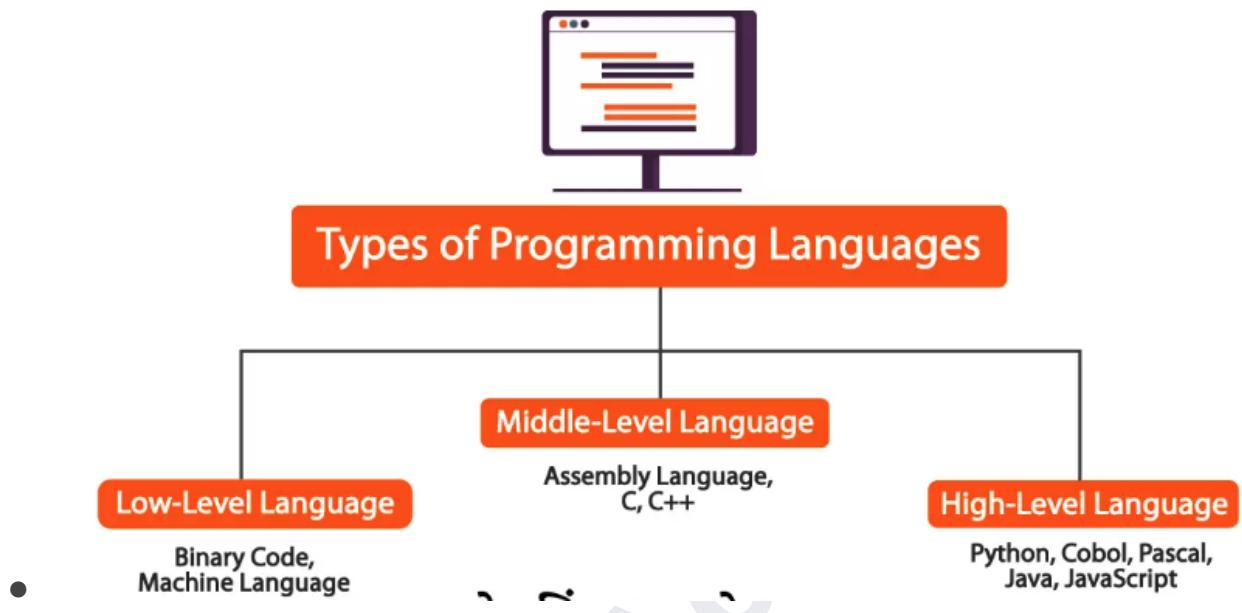
In this example, we print "Hello" and "World" with a dash separator, followed by an exclamation mark at the end.

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.

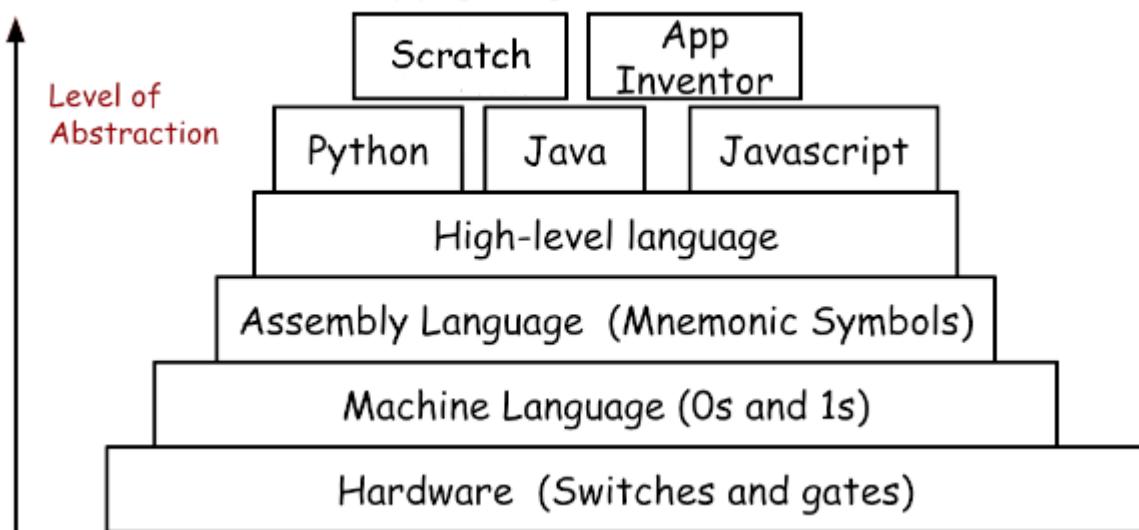
```
print(50, 1000, 3.142, "Hello World")
```

Python is Interpreted Language

Aspect	Interpreted Language	Compiled Language
Execution	Code is executed line-by-line by interpreter	Code is compiled to machine code/executable before running
Example Languages	Python, JavaScript, Ruby	C, C++, Java, Rust
Performance	Generally slower due to runtime interpretation	Generally faster due to pre-compiled machine code



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)

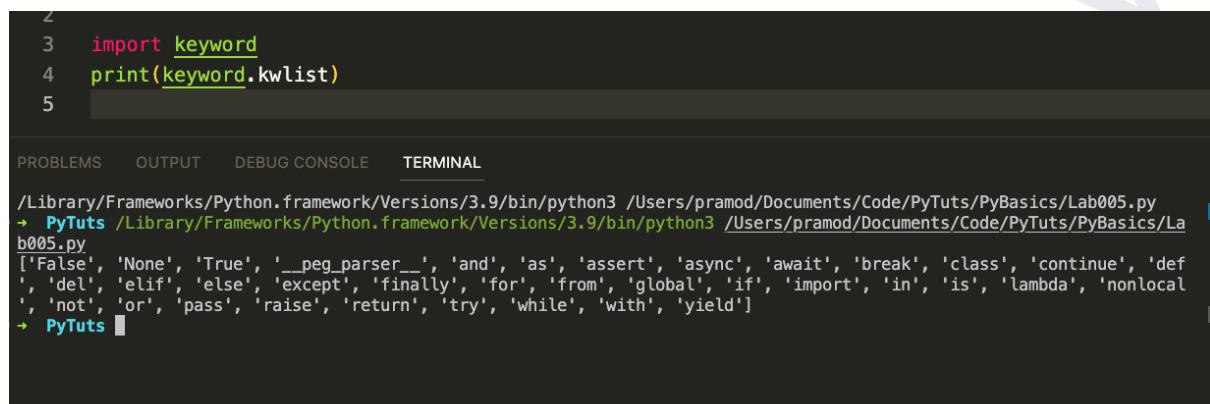
Keywords & Identifiers

- 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.

```
python Copy code  
False    await    else     import   pass  
None     break    except   in       raise  
True     class    finally  is       return  
and      continue for     lambda  try  
as       def      from    nonlocal while  
assert   del     global  not     with  
async    elif    if      or      yield
```

```
import keyword  
print(keyword.kwlist)
```

Lab005.py



```
2  
3 import keyword  
4 print(keyword.kwlist)  
5  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL  
/Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Users/pramod/Documents/Code/PyTuts/PyBasics/Lab005.py  
+ PyTuts /Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Users/pramod/Documents/Code/PyTuts/PyBasics/Lab005.py  
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def',  
'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',  
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']  
+ PyTuts
```

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

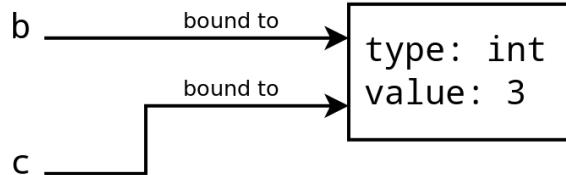
There are two types of variables based on data type used to declare the variable.

1. Numbers
2. Strings
3. Booleans

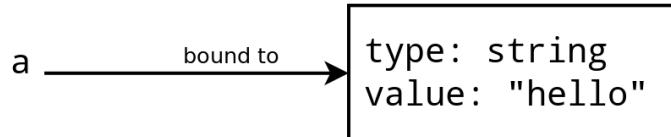
Executed Code: Variable Assignment

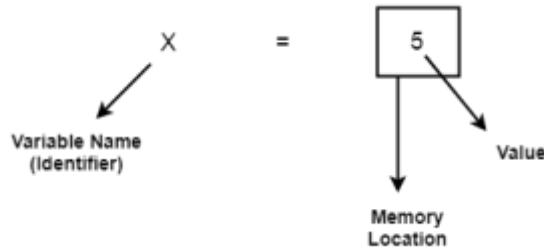
```
a = 3  
b = a  
c = a  
a = "hello"
```

Variables



Values in Memory





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

There are different types of data types in Python. Some built-in Python data types are:

- **Numeric data types:** *int, float, complex*
- **String data types:** *str*
- **Sequence types:** *list, tuple, range*
- **Binary types:** *bytes, bytearray, memoryview*
- **Mapping data type:** *dict*
- **Boolean type:** *bool*
- **Set data types:** *set, frozenset*

Ref

[Python Data Types \(With Complete List\) | DigitalOcean](#)

Integers

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

```
PyBasics > 🐍 Lab02.py > ...
1 # Integers
2 print(99) # A positive integer
3 print(-3001) # A negative integer
4
5 num = 123456789 # Assigning an integer to a variable
6 print(num)
7 num = -16001 # Assigning a new integer
8 print(num)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
/Libra.../Python.framework/Versions/3.9/bin/python3 /Users/pramod
py
● → PyTuts /Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Us
cs/Lab02.py
99
-3001
123456789
-16001
```

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)
```

The screenshot shows a Python code editor interface. On the left is a sidebar with file navigation. In the center is the code editor with the following Python script:

```
1 x = input("Type a number: ")
2 y = input("Type another number: ")
3 sum = int(x) + int(y)
4 print("The sum is: ", sum)
```

Below the code editor are tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active, showing the output of running the script:

```
● → PyTuts /Library/Frameworks/Python.framework/Versions/3.9/bin/pyt
    tegers/Int001.py
    Type a number: 34
    Type another number: 234
    The sum is: 268
○ → PyTuts []
```

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)
```

The screenshot shows a code editor window with a dark theme. At the top, there is a breadcrumb navigation bar: PyBasics > Lab004.py > ... Below this is the code editor area containing the following Python script:

```
1 print(complex(2.5, -18.2)) # Represents the complex number (2.5 - 18.2j)  
2  
3 c = complex(2.5, -18.2)  
4 c = c+1 # Addition  
5 print(c)
```

Below the code editor are several tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is selected, showing the command-line interface output of the script:

```
● → PyTuts /Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Users/pramod/Documents/Code  
cs/Lab004.py  
(2.5-18.2j)  
(3.5-18.2j)  
○ → PyTuts
```

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.

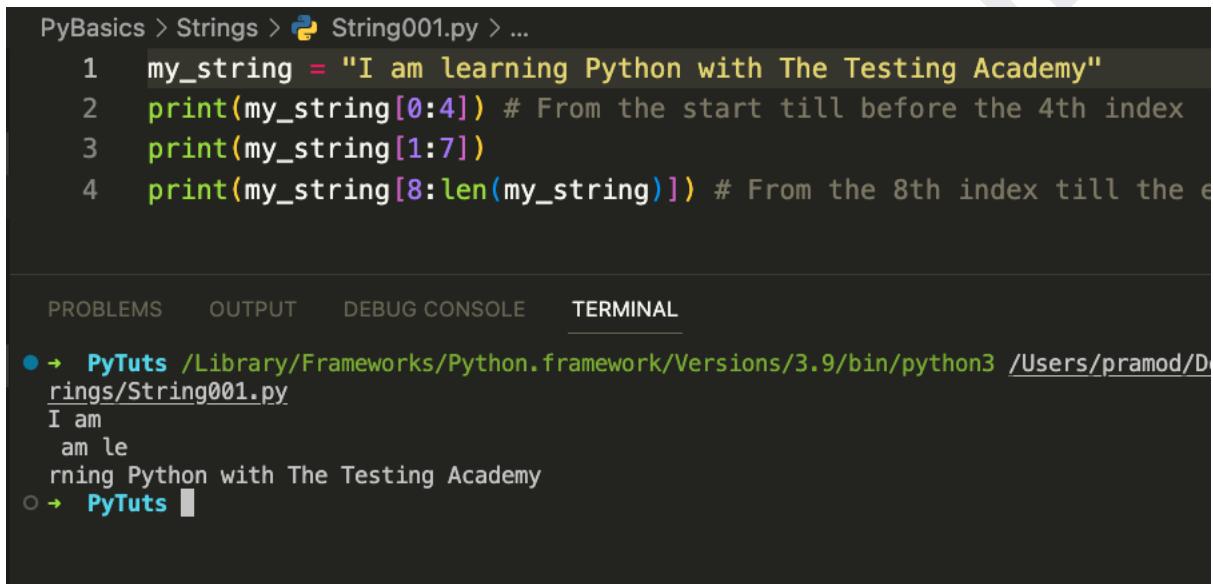
- None is not a default value for the variable that has not yet been assigned a value.
- None is not the same as False.
- None is not an empty string.
- None is not 0.

```
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]



The screenshot shows a code editor interface with a dark theme. At the top, there's a breadcrumb navigation: PyBasics > Strings > String001.py > Below the code area, there are tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is selected and displays the output of the Python script. The code in the editor is:

```
1 my_string = "I am learning Python with The Testing Academy"
2 print(my_string[0:4]) # From the start till before the 4th index
3 print(my_string[1:7])
4 print(my_string[8:len(my_string)]) # From the 8th index till the end
```

The terminal output shows:

```
I am
am le
m learning Python with The Testing Academy
```

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

```
PyBasics > Strings > 📁 String002.py > ...
<
3     my_string = "This is Pramod From The Testing Academy"
4     print(my_string[0:7]) # A step of 1
5     print(my_string[0:7:2]) # A step of 2
6     print(my_string[0:7:5]) # A step of 5

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● → PyTuts /Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Users/pr
rings/String002.py
This is
Ti s
Ti
○ → PyTuts
```

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
```

```
OPEN EDITORS PyBasics > Strings > 🐍 String003.py > ...
Lab06.py Labs
Lab005.py PyBasics
String001.py PyBasics/Str...
String002.py PyBasics/Str...
X 🐍 String003.py PyBasics/Str...
PYTUTS
Labs
PyBasics
  Strings
    String001.py
    String002.py
    String003.py
  Lab000.py

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● ➔ PyTuts /Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Use
rings/String003.py
domarP si s
!oaPs i
○ ➔ PyTuts []
```

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)

The screenshot shows a VS Code interface with several files listed in the sidebar under 'Labs' and 'PyBasics > Strings'. The 'String005.py' file is open, containing the following code:

```
1 String1 = "Hello, Pramod"
2 print("Initial String: ")
3 print(String1)
4
5 # Deleting a character
6 # of the String
7 String2 = String1[0:2] + String1[3:]
8 print("\nDeleting character at 2nd Index: ")
9 print(String2)
```

The terminal below shows the execution of the script:

```
/Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Users/pramod/Documents/String005.py
● → PyTuts /Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Users/pramod/Documents/String005.py
Initial String:
Hello, Pramod
Deleting character at 2nd Index:
Hello, Pramod
○ → PyTuts
```

Delete String

```
del String1
```

Escape Sequence

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

Here's a list of Python's escape sequences:

- `\\` : Backslash
- `\'` : Single quote
- `\"` : Double quote
- `\\a` : ASCII Bell (BEL)
- `\\b` : ASCII Backspace (BS)
- `\\f` : ASCII Formfeed (FF)
- `\\n` : ASCII Linefeed (LF), it's used for a new line
- `\\r` : ASCII Carriage Return (CR)
- `\\t` : ASCII Horizontal Tab (TAB)
- `\\v` : ASCII Vertical Tab (VT)
- `\\ooo` : ASCII character with octal value ooo
- `\\hh` : ASCII character with hex value hh

```
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)

Character	ASCII	OCTAL	UNICODE	Character	ASCII	OCTAL	UNICODE
A	65	101	\u0041	a	97	141	\u0061
B	66	102	\u0042	b	98	142	\u0062
C	67	103	\u0043	c	99	143	\u0063
D	68	104	\u0044	d	100	144	\u0064
E	69	105	\u0045	e	101	145	\u0065
F	70	106	\u0046	f	102	146	\u0066
G	71	107	\u0047	g	103	147	\u0067
H	72	110	\u0048	h	104	150	\u0068
I	73	111	\u0049	i	105	151	\u0069
J	74	112	\u004A	j	106	152	\u006A
K	75	113	\u004B	k	107	153	\u006B
L	76	114	\u004C	l	108	154	\u006C
M	77	115	\u004D	m	109	155	\u006D
N	78	116	\u004E	n	110	156	\u006E
O	79	117	\u004F	o	111	157	\u006F
P	80	120	\u0050	p	112	160	\u0070
Q	81	121	\u0051	q	113	161	\u0071
R	82	122	\u0052	r	114	162	\u0072
S	83	123	\u0053	s	115	163	\u0073
T	84	124	\u0054	t	116	164	\u0074
U	85	125	\u0055	u	117	165	\u0075
V	86	126	\u0056	v	118	166	\u0076
W	87	127	\u0057	w	119	167	\u0077
X	88	130	\u0058	x	120	170	\u0078
Y	89	131	\u0059	y	121	171	\u0079
Z	90	132	\u005A	z	122	172	\u007A

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

Escape Sequence	Description
\t	Tab Space.
\b	Backspace.
\n	Newline.
\r	Carriage return.
\f	Formfeed.
\'	Single quote character.
\"	Double quote character.
\\\	Backslash character.

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

$$061 = (0 \times 8^2) + (6 \times 8^1) + (1 \times 8^0) = 49$$

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

Character	ASCII	OCTAL	UNICODE	Character	ASCII	OCTAL	UNICODE
0	48	060	\u0030	5	53	065	\u0035
1	49	061	\u0031	6	54	066	\u0036
2	50	062	\u0032	7	55	067	\u0037
3	51	063	\u0033	8	56	070	\u0038
4	52	064	\u0034	9	57	071	\u0039

Character	ASCII	OCTAL	UNICODE	Character	ASCII	OCTAL	UNICODE
A	65	101	\u0041	a	97	141	\u0061
B	66	102	\u0042	b	98	142	\u0062
C	67	103	\u0043	c	99	143	\u0063
D	68	104	\u0044	d	100	144	\u0064
E	69	105	\u0045	e	101	145	\u0065
F	70	106	\u0046	f	102	146	\u0066
G	71	107	\u0047	g	103	147	\u0067
H	72	110	\u0048	h	104	150	\u0068
I	73	111	\u0049	i	105	151	\u0069
J	74	112	\u004A	j	106	152	\u006A
K	75	113	\u004B	k	107	153	\u006B
L	76	114	\u004C	l	108	154	\u006C
M	77	115	\u004D	m	109	155	\u006D
N	78	116	\u004E	n	110	156	\u006E
O	79	117	\u004F	o	111	157	\u006F
P	80	120	\u0050	p	112	160	\u0070
Q	81	121	\u0051	q	113	161	\u0071
R	82	122	\u0052	r	114	162	\u0072
S	83	123	\u0053	s	115	163	\u0073
T	84	124	\u0054	t	116	164	\u0074
U	85	125	\u0055	u	117	165	\u0075
V	86	126	\u0056	v	118	166	\u0076
W	87	127	\u0057	w	119	167	\u0077
X	88	130	\u0058	x	120	170	\u0078
Y	89	131	\u0059	y	121	171	\u0079
Z	90	132	\u005A	z	122	172	\u007A

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 Literal
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 |

| | |
|---|---|
| | <p>string literal. It usually starts with a backslash () character.</p> |
| <p>What will be displayed when UNICODE value is found in String Literal?</p> <pre>String str="UNICODE of A is A is \u0041";</pre> | <p>The string "UNICODE of A is A" will be displayed because \u0041 represents the Unicode value of the letter A.</p> |
| <p>What will be displayed when Octal representation is found in String Literal?</p> <pre>String str="Octal of A is \101";</pre> | <p>The string "Octal of A is A" will be displayed because \101 represents the octal value of the letter A.</p> |
| <p>What will happen when Escape Sequence is found in String Literal?</p> <pre>String str="Hello\nGuys";</pre> | <p>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.</p> |

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

Operator	Description
+	Addition (SUM)
-	Subtraction (DIFFERENCE)
*	Multiplication (PRODUCT)
/	Division (QUOTIENT)
%	Modulus (REMAINDER)

```
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
```

AND Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

NOT Truth Table

A	B
0	1
1	0

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!
```

Operator	Description
<code>+=</code>	Add right operand to left operand and assign the result to left operand.
<code>-=</code>	Subtract right operand from left operand and assign the result to left operand.
<code>*=</code>	Multiply right operand to left operand and assign the result to left operand.
<code>/=</code>	Divide right operand to left operand and assign the result to left operand.
<code>%=</code>	Calculate modulus using two operands and assign the result to left operand.

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
```

TheTestingAcademy

Relational Operators

Operator	Description	Example
`>`	Greater than: True if left operand is greater than the right	`x = 10; y = 3; print(x > y) # Prints: True`
`<`	Less than: True if left operand is less than the right	`x = 10; y = 3; print(x < y) # Prints: False`
`==`	Equal to: True if both operands are equal	`x = 10; y = 10; print(x == y) # Prints: True`
`!=`	Not equal to: True if operands are not equal	`x = 10; y = 3; print(x != y) # Prints: True`
`>=`	Greater than or equal to: True if left operand is greater than or equal to the right	`x = 10; y = 10; print(x >= y) # Prints: True`
`<=`	Less than or equal to: True if left operand is less than or equal to the right	`x = 10; y = 20; print(x <= y) # Prints: True`

Logical Operators

Logical OR and Logical AND

Here's a truth table that illustrates these operators:

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Bitwise Operators

Operator	Description
<code>~</code>	Bitwise NOT
<code>&</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Exclusive OR (XOR)
<code><<</code>	Left Shift
<code>>></code>	Right Shift

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);
```

3) If Else-if statement

```
if(<condition1>){
```

```
    // Statements (Block 1)
```

```
}else if(<condition2>){
```

```
    // Statements (Block 2)
```

```
}else if(<condition3>){
```

```
    // Statements (Block 3)
```

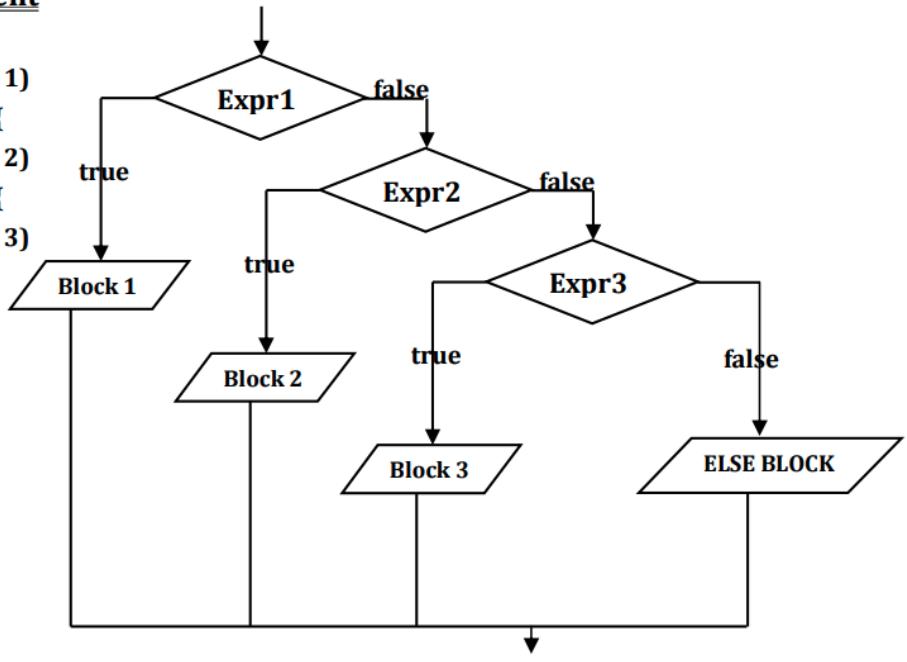
```
}
```

```
...
```

```
else{
```

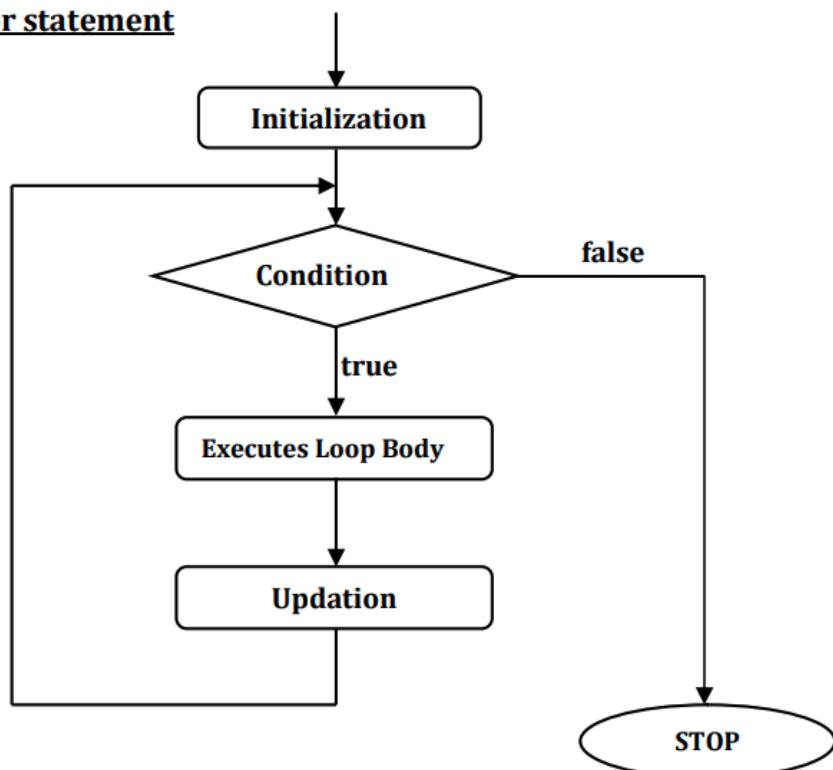
```
    // Statements
```

```
}
```



For

Processing Flow of for statement



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)

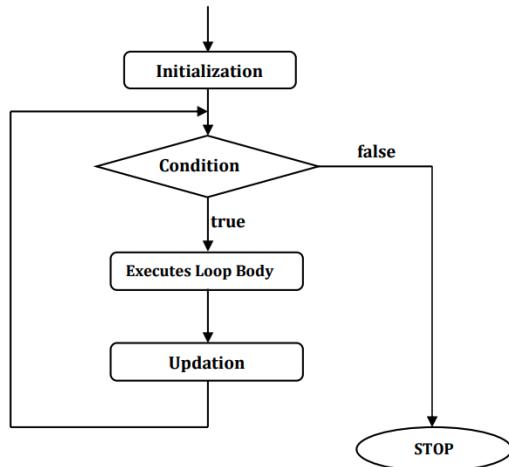
# If you really need a list for some reason:
numbers = list(range(10))
```

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

While

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

Processing Flow of while statement



```
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` 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 Statement
- Multiple -> If else loops

```
parameter = "Pramoda"  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|  
  
match parameter:  
    case "Pramod":  
        print("Hi")  
    case "PramOD":  
        print("2")  
    case _:  
        print("Default")
```

✓ 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.

| Function | Description | Example | Result |
|----------|--|----------------------|---------------------------|
| abs() | Returns the absolute value of a number | abs(-5) | 5 |
| bool() | Returns the boolean value of the specified object | bool(0) | False |
| chr() | Returns a string representing a character at the specified Unicode | chr(97) | 'a' |
| dict() | Creates a dictionary | dict(a=1, b=2) | {'a': 1, 'b': 2} |
| float() | Returns a floating point number | float(5) | 5.0 |
| hex() | Converts a number into a hexadecimal | hex(255) | '0xff' |
| int() | Returns an integer object from a number or a string | int('10') | 10 |
| len() | Returns the number of items in an object | len('Hello') | 5 |
| list() | Creates a list | list('Hello') | ['H', 'e', 'l', 'l', 'o'] |
| max() | Returns the item with the highest value | max(1, 2, 3) | 3 |
| min() | Returns the item with the lowest value | min(1, 2, 3) | 1 |
| pow() | Returns the value of x to the power of y | pow(2, 3) | 8 |
| round() | Rounds a number | round(5.76543, 2) | 5.77 |
| str() | Converts a specified value into a string | str(123) | '123' |
| sum() | Sums the items of an iterable | sum([1, 2, 3, 4, 5]) | 15 |
| type() | Returns the type of an object | type(123) | <class 'int'> |

More details

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

round_Built-in Functions

| | | | |
|--|--|---|--|
| A
abs()
aiter()
all()
any()
anext()
ascii() | E
enumerate()
eval()
exec() | L
len()
list()
locals() | R
range()
repr()
reversed()
<u>round()</u> |
| B
bin()
bool()
breakpoint()
bytearray()
bytes() | F
filter()
float()
format()
frozenset() | M
map()
max()
memoryview()
min() | S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super() |
| C
callable()
chr()
classmethod()
compile()
complex() | G
getattr()
globals() | N
next() | T
tuple()
type() |
| D
delattr()
dict()
dir()
divmod() | H
hasattr()
hash()
help()
hex() | O
object()
oct()
open()
ord() | V
vars() |
| | I
id()
input()
int()
isinstance()
issubclass()
iter() | P
pow()
print()
property() | Z
zip() |
| | | | _
<u>import</u> () |



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

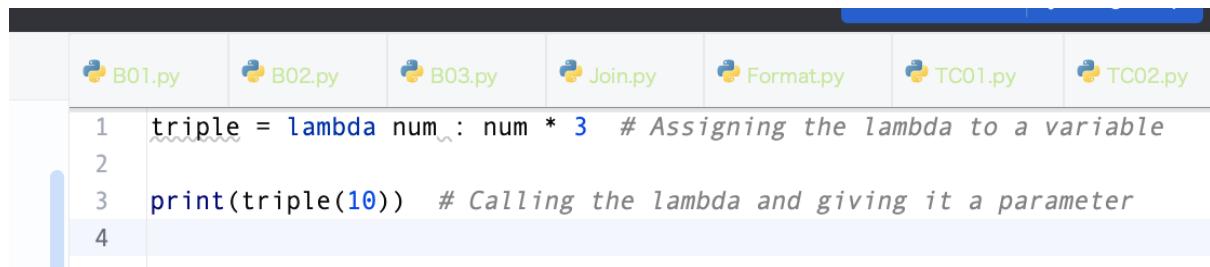
| Function | Description | Example | Result |
|-----------|--|----------------------|---------------------------|
| int() | Converts a value to an integer | int('123') | 123 |
| float() | Converts a value to a floating point number | float('123.45') | 123.45 |
| str() | Converts a value to a string | str(123) | '123' |
| list() | Converts a value to a list | list('hello') | ['h', 'e', 'l', 'l', 'o'] |
| tuple() | Converts a value to a tuple | tuple('hello') | ('h', 'e', 'l', 'l', 'o') |
| set() | Converts a value to a set | set('hello') | {'h', 'e', 'l', 'o'} |
| dict() | Converts a sequence of tuples to a dictionary | dict([(1,2), (3,4)]) | {1: 2, 3: 4} |
| bool() | Converts a value to a boolean | bool(1) | True |
| bytes() | Converts a value to bytes | bytes(5) | b'\x00\x00\x00\x00\x00' |
| complex() | Converts a real number to complex (real+imaginary) | complex(1,2) | (1+2j) |

Lambda

lambda **parameters** : **expression**

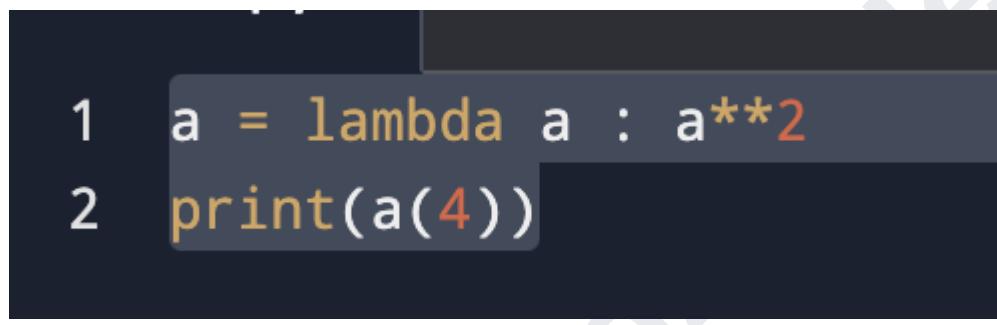
Separated by commas An operation that returns something

- 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.



A screenshot of a code editor window. At the top, there is a horizontal tab bar with several tabs labeled: B01.py, B02.py, B03.py, Join.py, Format.py, TC01.py, and TC02.py. Below the tabs, the main area contains the following Python code:

```
1 triple = lambda num : num * 3 # Assigning the lambda to a variable
2
3 print(triple(10)) # Calling the lambda and giving it a parameter
4
```



A screenshot of a code editor window with a dark background. It displays the following Python code:

```
1 a = lambda a : a**2
2 print(a(4))
```

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)

```
1 # Define a function to square a number
2 def square(x):
3     return x ** 2
4
5 # Apply the square function to a list of numbers using map()
6 numbers = [1, 2, 3, 4, 5]
7 squared_numbers = map(square, numbers)
8
9 # Convert the map object to a list and print it
10 # result = list(squared_numbers)
11 # print(result) # Output: [1, 4, 9, 16, 25]
12
13
14 # Iterate over the map object and print each result
15 for result in squared_numbers:
16     print(result)
17
```

main.py

```
1 # Define a function to square a number
2 def square(x):
3     return x**2
4
5 # Apply the square function to a list of numbers using map
6 numbers = [1, 2, 3, 4, 5]
7 squared_numbers = map(square, numbers)
8
9 # Convert the map object to a list
10 result = list(squared_numbers)
11 print(result) # Output: [1, 4, 9, 16, 25]
12 |
```

Filter

main.py

```
1 numbers = [1, 2, 3, 4, 5]
2
3 # Using lambda as an argument to the filter() function
4 even_numbers = filter(lambda x: x % 2 == 0, numbers)
5 print(list(even_numbers)) # Output: [2, 4]
6 |
```

python

 Copy code

```
# Define a function to check if a number is even
def is_even(x):
    return x % 2 == 0

# Filter even numbers from a list using filter()
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(is_even, numbers)

# Convert the filter object to a list
result = list(even_numbers)
print(result) # Output: [2, 4]
```

Using lambda as argument

python

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

# Filter even numbers from a list using filter() with lambda
even_numbers = filter(lambda x: x % 2 == 0, numbers)

# Convert the filter object to a list
result = list(even_numbers)
print(result) # Output: [2, 4]
```

Functions as Arguments

1. Using lambda as argument
2. Using map lambda

Using map lambda

```
python

numbers = [1, 2, 3, 4, 5]

# Square each number in the list using map() with lambda
squared_numbers = map(lambda x: x ** 2, numbers)

# Convert the map object to a list
result = list(squared_numbers)
print(result) # Output: [1, 4, 9, 16, 25]
```

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.

```
python
```

```
def greet(name="World"):
    print("Hello, " + name + "!")
greet()          # Output: Hello, World!
greet("Alice")   # Output: Hello, Alice!
```

✓ 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.

```
main.py
```

```
1 def rec_count(number):
2     print(number)
3     # Base case
4     if number == 0:
5         return 0
6     rec_count(number - 1) # A recursive call with a different argument
7
8
9 rec_count(5)
10
```

```
python
```

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Calculate the Fibonacci number at index 6
result = fibonacci(6)
print(result) # Output: 8
```

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
# Calculate the factorial of 5  
result = factorial(5)  
print(result) # Output: 120
```

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

main.py

```
1 def sum_list(lst):
2     if not lst:
3         return 0
4     else:
5         return lst[0] + sum_list(lst[1:])
6
7 # Calculate the sum of [1, 2, 3, 4, 5]
8 result = sum_list([1, 2, 3, 4, 5])
9 print(result) # Output: 15
```

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)`

| Operation | Description | Example | Result |
|---------------------|--|--|--|
| Indexing | Access an item of a list at a specific position. | my_list = [1, 2, 3];
my_list[0] | 1 |
| Slicing | Access multiple items of a list. | my_list = [1, 2, 3];
my_list[0:2] | [1, 2] |
| Changing an element | Modify an existing item of a list using its index position. | my_list = [1, 2, 3];
my_list[1] = 20 | my_list becomes [1, 20, 3] |
| append() | Add an item to the end of a list. | my_list = [1, 2, 3];
my_list.append(4) | my_list becomes [1, 2, 3, 4] |
| extend() | Add all items of a list to another list. | my_list = [1, 2];
my_list.extend([3,4]) | my_list becomes [1, 2, 3, 4] |
| insert() | Insert an item at a specific position in the list. | my_list = [1, 2];
my_list.insert(1, 'a') | my_list becomes [1, 'a', 2] |
| remove() | Remove the first occurrence of a specific item from the list. | my_list = [1, 'a', 2];
my_list.remove('a') | my_list becomes [1, 2] |
| pop() | Remove the item at a specific position, or the last item if no index is specified. | my_list = [1, 2, 3];
my_list.pop(1) | my_list becomes [1, 3] and 2 is returned |
| clear() | Remove all items from the list. | my_list = [1, 2, 3];
my_list.clear() | my_list becomes [] |
| index() | Return the index of the first occurrence of a specified item. | my_list = ['a', 'b', 'c'];
my_list.index('b') | 1 |
| count() | Return the number of times a specified item appears in the list. | my_list = [1, 1, 2];
my_list.count(1) | 2 |
| sort() | Sort the items in the list in ascending order. | my_list = [3, 1, 2];
my_list.sort() | my_list becomes [1, 2, 3] |
| reverse() | Reverse the order of the items in the list. | my_list = [1, 2, 3];
my_list.reverse() | my_list becomes [3, 2, 1] |

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.

```
LabTuple01.py U
PyBasics > Tuple > LabTuple01.py > ...
1 car = ("Ford", "Raptor", 2019, "Red")
2 print(car)
3 car[0] = "Pramod"
4 print(car)
5
6 # Length
7 print(len(car))
8
9 # Indexing
10 print(car[1])
11
12 # Slicing
13 print(car[2:])
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
④ → PyTuts git:(main) ✘ /Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Users/pramod/Documents/Code/Tuple/LabTuple01.py
('Ford', 'Raptor', 2019, 'Red')
Traceback (most recent call last):
  File "/Users/pramod/Documents/Code/PyTuts/PyBasics/Tuple/LabTuple01.py", line 3, in <module>
    car[0] = "Pramod"
TypeError: 'tuple' object does not support item assignment
○ → PyTuts git:(main) ✘
```

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.

Tuples VS Lists:

| Similarities | Differences |
|--|---|
| Functions that can be used for both lists and tuples:
<code>len()</code> , <code>max()</code> , <code>min()</code> , <code>sum()</code> , <code>any()</code> , <code>all()</code> , <code>sorted()</code> | Methods that cannot be used for tuples:
<code>append()</code> , <code>insert()</code> , <code>remove()</code> , <code>pop()</code> , <code>clear()</code> , <code>sort()</code> , <code>reverse()</code> |
| Methods that can be used for both lists and tuples:
<code>count()</code> , <code>Index()</code> | we generally use 'tuples' for heterogeneous (different) data types and 'lists' for homogeneous (similar) data types. |
| Tuples can be stored in lists. | Iterating through a 'tuple' is faster than in a 'list'. |
| Lists can be stored in tuples. | 'Lists' are mutable whereas 'tuples' are immutable. |
| Both 'tuples' and 'lists' can be nested. | Tuples that contain immutable elements can be used as a key for a dictionary. |

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

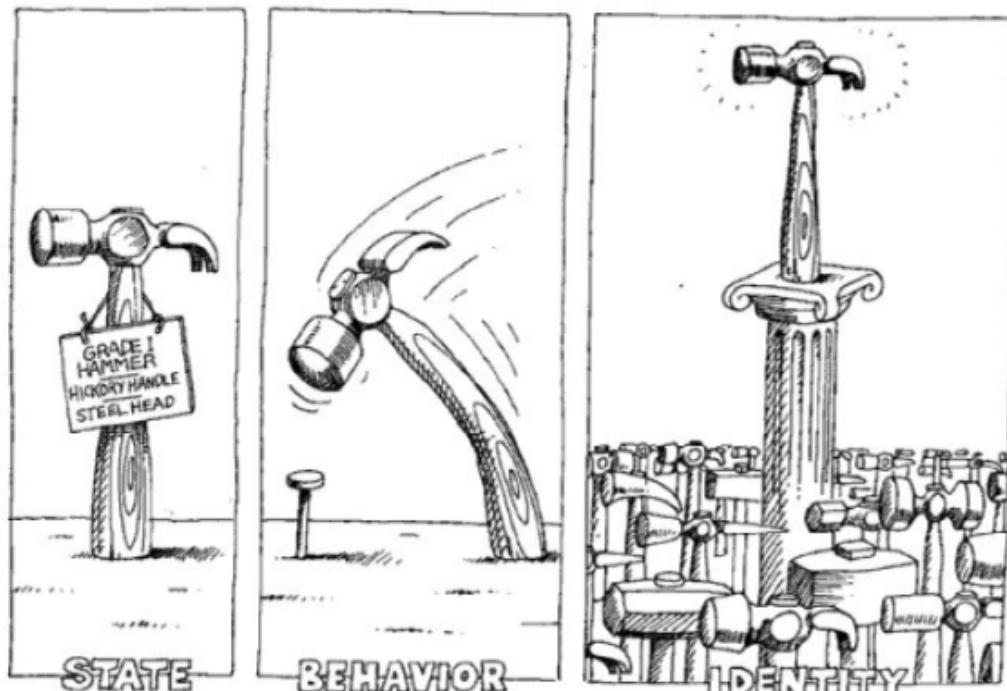
- o Polymorphism

Everything in the world is an object.

OOPs vs POP or PPL (Procedural oriented programming)

Object vs Functional.

- **Grady Booch - Father of OOPS defined the object as follows:**



An object has state, exhibits some well-defined behavior, and has a unique identity.

Objects are a collection of data and their behaviors.

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

| Procedural Programming Language | Object Oriented Programming Language |
|---|--|
| 1. Program is divided into functions. | 1. Program is divide into classes and objects.. |
| 2. The emphasis is on doing things. | 2. The emphasis on data. |
| 3. Poor modeling to real world problems. | 3. Strong modeling to real world problems. |
| 4. It is not easy to maintain project if it is too complex. | 4. It is easy to maintain project even if it is too complex. |
| 5. Provides poor data security. | 5. Provides strong data Security. |
| 6. It is not extensible programming language. | 6. It is highly extensible programming language. |
| 7. Productivity is low. | 7. Productivity is high. |
| 8. Do not provide any support for new data types. | 8. Provide support to new Data types. |
| 9. Unit of programming is function. | 9. Unit of programming is class. |
| 10. Ex. Pascal , C , Basic , Fortran. | 10. Ex. C++ , Java , Oracle. |

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

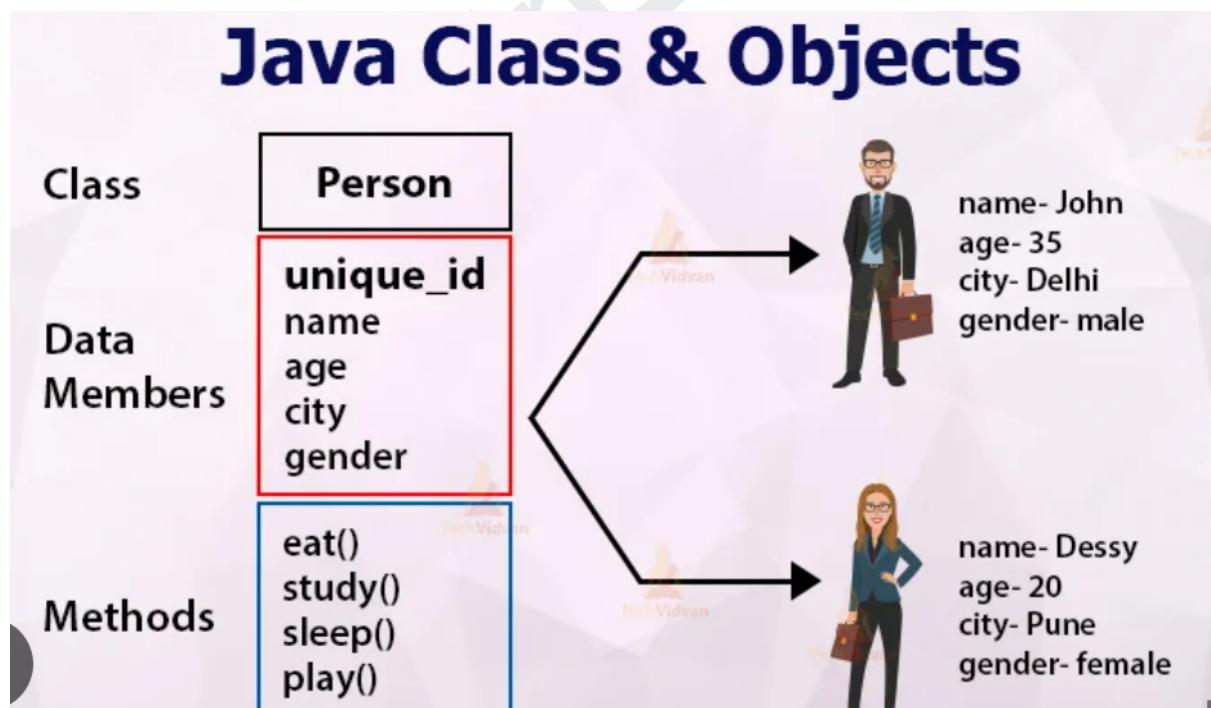
Class and Objects

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

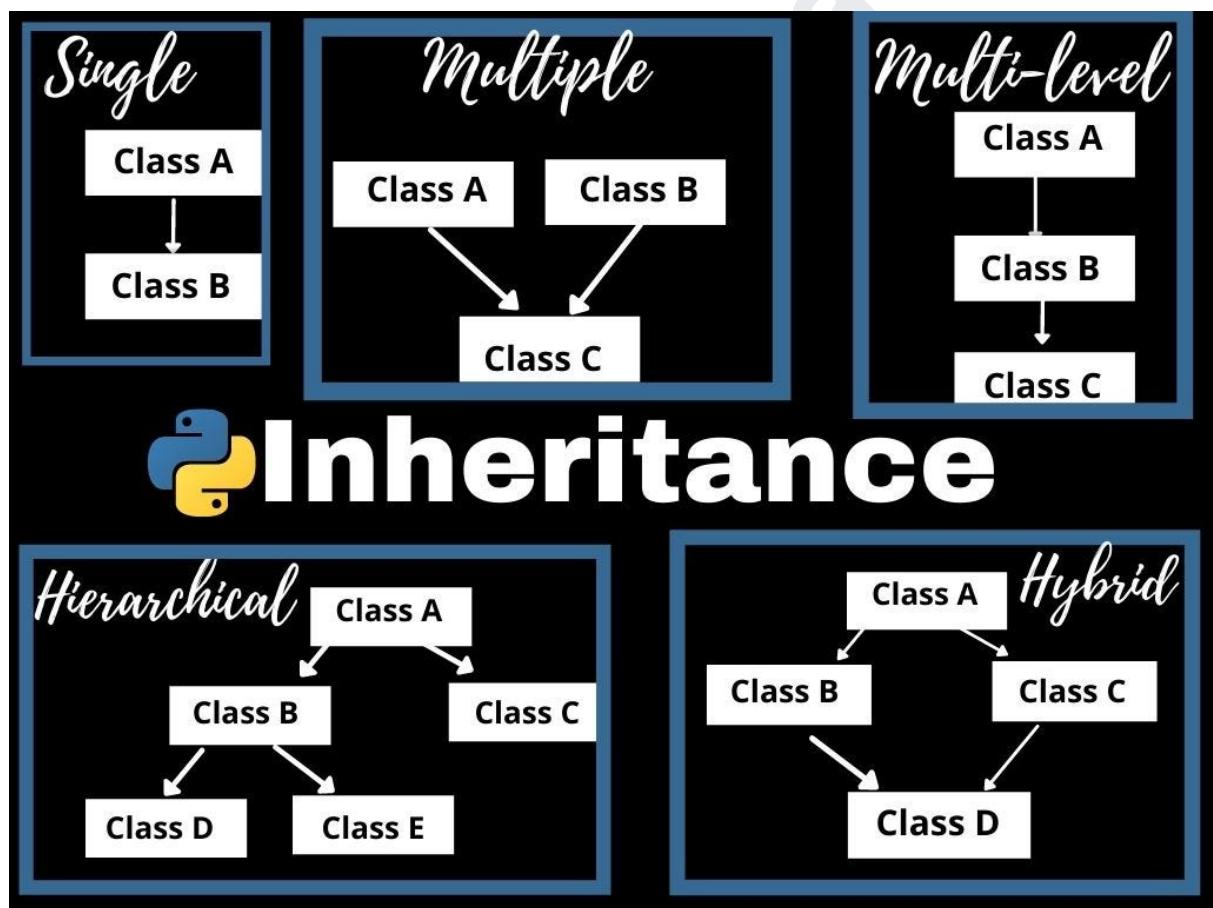
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

shapes = [Rectangle(4, 5), Circle(3)]
for shape in shapes:
    print(shape.area())
```

The /

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.
- **FileNotFoundException:** 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

```
python

import math

x = 16
sqrt_x = math.sqrt(x)
print(sqrt_x) # Output: 4.0
```

```
from math import pi, sin

angle = pi / 2
sin_angle = sin(angle)
print(sin_angle) # Output: 1.0
```

- Custom Modules - my_module.py

```
python
```

```
def greet(name):
    print(f"Hello, {name}!")

class Person:
    def __init__(self, name):
        self.name = name

    def introduce(self):
        print(f"My name is {self.name}.")
```

```
import my_module

my_module.greet("Alice") # Output: Hello, Alice!

person = my_module.Person("Bob")
person.introduce() # Output: My name is Bob.
```

- Aliasing Modules

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr) # Output: [1 2 3 4 5]
```

- 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'
```

markdown

```
my_package/  
    __init__.py  
    module1.py  
    module2.py
```

To import modules from a package, you can use the dot notation:

python

```
import my_package.module1  
from my_package.module2 import some_function  
  
my_package.module1.function1()  
some_function()
```

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.

```
my_package_part1/  
    my_module1.py  
    my_module2.py
```

```
my_package_part2/  
    my_module3.py  
    my_module4.py
```

```
from my_package_part1 import my_module1  
from my_package_part2 import my_module3  
  
my_module1.function1()  
my_module3.function2()
```



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
```