Python Roadmap

▼ Reddit Wiki

https://www.reddit.com/r/learnpython/wiki/index/

▼ Reference

https://book.pythontips.com/en/latest/

▼ Full stack python

https://www.fullstackpython.com/django.html

▼ Roadmap Part 1

1. MOOC Course - d
2. PDF – Pydon'ts v1.0.2.pdf in python books - r
3. Learn enough python to be dangerous - Upto chapter 9
4.

https://diveintopython3.net/

4. https://automatetheboringstuff.com
5.

Week 0 Functions - CS50's Introduction to Programming with Python

An introduction to programming using Python, a popular language for general-purpose programming, data science, web programming, and more.

https://cs50.harvard.edu/python/2022/weeks/0/

6.

python-deepdive/Part 4 at main · fbaptiste/python-deepdive

Python Deep Dive Course - Accompanying Materials. Contribute to fbaptiste/python-deepdive development by creating an account on GitHub.

https://github.com/fbaptiste/python-deepdive/tree/main/Part%20 4

fbaptiste/python-deepdive

ython Deep Dive Course - Accompanying laterials

12 Contributors · 0 Issues · 2k Stars · 1k Forks

7.

https://www.py4e.com/lessons

8 . Learn OOPS Concept

- https://www.youtube.com/watch?v=ZDa-Z5JzLYM
- https://realpython.com/python3-object-oriented-programming/
- https://python-course.eu/oop/object-oriented-programming.php
- https://www.thepythoncodingstack.com/p/python-object-oriented-programming-mindset-1
- https://matematika-mipa.unsri.ac.id/wp-content/uploads/2022/05/Object-Oriented-Python-Master-OOP-by-Building-Games-and-GUIs-Irv-Kalb-z-lib.org_.pdf

▼ Roadmap Part 2

Beginner

- Data Types - Lists, Strings, Tuples, Sets, Floats, Ints, Booleans, Dictionaries
- Control Flow/Looping - for loops, while loops, if/elif/else
- Arithmetic and expressions
- I/O (Input/Output) - Sys module, Standard input/output, reading/writing files
- Functions
- Exceptions and Error Handling
- Basics of object oriented programming (OOP) (Simple classes).

Intermediate

- Recursion

- More advanced OOP - Inheritance, Polymorphism, Encapsulation, Method overloading.
- Data Structures - Linked lists, Stacks, Queues, Binary Search Trees, AVL Trees, Graphs, Minimum Spanning Trees, Hash Maps
- Algorithms - Linear Search, Binary Search, Hashing, Quicksort, Insertion/Selection Sort, Merge Sort, Radix Sort, Depth First Search, Breathe First Search, Prim's Algorithm, Dijkstra's Algorithm.
- Algorithmic Complexity

Advanced - A.I. / Machine Learning/ Data science

- Statistics
- Probability
- Brute Force search
- Heuristic search (Manhattan Distance, Admissible and Informed Heuristics)
- Hill Climbing
- Simulated Annealing
- A* search
- Adversarial Search (Minimax & Alpha-Beta pruning)
- Greedy Algorithms
- Dynamic Programming
- Genetic Algorithms
- Artificial Neural Networks
- Backpropagation
- Natural Language Processing
- Convolutional Neural Networks
- Recurrent Neural Networks
- Generative Adversarial Networks

Advanced - Full stack web development

- Computer networks (Don't need to go into heavy detail but an understanding is necessary)

- Backend web dev tools (flask, django) (This is for app logic, interfacing with databases etc).
- Front end framework (This is for communicating with the backend) (Angular 6+, React/Redux)
- With frontend you'll also need - HTML, CSS, Javascript (also good to learn typescript which is using in angular. It makes writing javascript nicer).
- Relational database (MySQL, PostgreSQL)
- Non-relational (MongoDB)
- Cloud computing knowledge is good, (AWS have a free tier that lasts a year, so its a nice way to get to know how that works

▼ Interview Questions

▼ Is Python compiled or interpreted language ?

Python is both compiled as well as an interpreted language. This means when we run a python code, it is first compiled and then interpreted line by line. The compilation part is mostly hidden from the user. While running the code, Python generates a byte code internally, this byte code is then converted using a python virtual machine (p.v.m) to generate the output.

▼ What is a dynamically typed language?

Typed languages are the languages in which we define the type of data type and it will be known by the machine at the compile-time or at runtime. Typed languages can be classified
into two categories:

Statically typed languages: In this type of language, the data type of a variable is known at the compile time which means the programmer has to specify the data type of a variable at the time of its declaration.

Dynamically typed languages: These are the languages that do not require any pre-defined data type for any variable as it is interpreted at runtime by the machine itself. In these languages, interpreters assign the data type to a variable at runtime depending on its value.

▼ What is __init__?

`__init__` is a contructor method in Python and is automatically called to allocate memory when a new object/instance is created. All classes have a **__init__** method associated with them. It helps in distinguishing methods and attributes of a class from local variables.

```
# class definition
class Student:
  def __init__(self, fname, lname, age, section):
      self.firstname = fname
      self.lastname = lname
      self.age = age
      self.section = section
# creating a new object
stu1 = Student("Sara", "Ansh", 22, "A2")
```

▼ What is slicing in Python ?

Python Slicing is a string operation for extracting a part of the string, or some part of a list. With this operator, one can specify where to start the slicing, where to end, and specify the step. List slicing returns a new list from the existing list.

▼ Set vs Dict

Parameter	**Set**	**Dictionary**
Definition	An unordered collection of unique elements.	An ordered collection of key-value pairs.
Representation	Curly braces {}.	Curly braces {}, but with key-value pairs.
Order	Unordered.	Ordered (from Python 3.7 onwards).
Duplicates	Does not allow duplicate elements.	Does not allow duplicate keys.
Mutability	Mutable (can add or remove elements).	Mutable (can add, modify, or remove key-value pairs).
Access Method	Elements are accessed directly.	Values are accessed using keys.
Use Case	To store unique elements.	To store related pieces of information.
Example	my_set = {1, 2, 3}	my_dict = {“name”: “Alice”, “age”: 30}

▼ What is List Comprehension? Give an Example.

List comprehension is a syntax construction to ease the creation of a list based on an existing iterable.

Example:-

```
my_list = [i for i in range(1, 10)]
```

▼ What is a lambda function?

A lambda function is an anonymous function. This function can have any number of parameters but can have just one statement.
For example:-

```
a = lambda x, y : x*y
print(a(7, 19))
```

▼ Array vs list

List	**Array**
Can consist of elements belonging to different data types	Only consists of elements belonging to the same data type
No need to explicitly import a module for the declaration	Need to explicitly import the **array** module for declaration
Cannot directly handle arithmetic operations	Can directly handle arithmetic operations
Preferred for a shorter sequence of data items	Preferred for a longer sequence of data items
Greater flexibility allows easy modification (addition, deletion) of data	Less flexibility since addition, and deletion has to be done element-wise
The entire list can be printed without any explicit looping	A loop has to be formed to print or access the components of the array

Can perform direct operations using functions like: count() - for counting a particular element in the list sort() - sort the complete list max() - gives maximum of the list min() - gives minimum of the list sum() - gives sum of all the elements in list for integer list index() - gives first index of the element specified append() - adds the element to the end of the list remove() - removes the element specified No need to import anything to use these functions. and many more...	Need to import proper modules to perform these operations.
Example:my_list = [1, 2, 3, 4]	**Example:**import arrayarr = array.array('i', [1, 2, 3])

▼ What is docstring in python ?

- It is used to document a specific code segment.
- The docstrings are declared using ”’triple single quotes”’ or “””triple double quotes””” just
  below the class, method, or function declaration.
- Accessing Docstrings: The docstrings can be accessed using the **__ doc__** method of the object or using the help function.

▼ What is higher order function & types ?

A higher-order function is a function that either, takes one or more functions as arguments or returns a function as its result

Map

The map () function returns a map **object(which is an iterator)** of the results after applying the given function to each item of a given iterable (**list**, **tuple**, etc.).

The general syntax for the map function is where `function` is the operation we want to execute on each item in the iterable

```
map(<function>, <iterabe>)
```

The `map` function returns an object of type `map`, which is iterable, and can be converted into a list:

```
def capitalize(my_string: str):
    first = my_string[0]
    first = first.upper()
    return first + my_string[1:]

test_list = ["first", "second", "third", "fourth"]

capitalized = map(capitalize, test_list)

capitalized_list = list(capitalized)
print(capitalized_list)

['First', 'Second', 'Third', 'Fourth']
```

Filter

filter() function filters them with a criterion function, which is passed as an argument. If the criterion function returns `True`, the item is selected.

```
filter(<function>, <iter>)
```

```
integers = [1, 2, 3, 5, 6, 4, 9, 10, 14, 15]

even_numbers = list(filter(lambda number: number % 2 == 0, integers))

for number in even_numbers:
    print(number)

2
6
4
10
14
```

Reduce

The **reduce** function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.This function is defined in “functools” module

```
reduce(func, iterable[, initial])
```

```
from functools import reduce

my_list = [2, 3, 1, 5]

sum_of_numbers = reduce(lambda reduced_sum, item: reduced_sum + item, my_list, 0)

print(sum_of_numbers)

11
```

▼ What is pass in Python?

The `pass` keyword represents a null operation in Python. It is generally used for the purpose of filling up empty blocks of code which may execute during runtime but has yet to be written. Without the **pass** statement in the following code, we may run into some errors during code execution.

▼ What is the difference between / and // in Python ?

// represents floor division whereas / represents precise division.

```
5//2 = 2
5/2 = 2.5
```

▼ What is a zip function ?

Python zip() function returns a zip object, which maps a similar index of multiple containers. It takes an iterable, converts it into an iterator and aggregates the elements based on iterables passed. It returns an iterator of tuples.

▼ What are *args and *kwargs?

***args**

- args is a special syntax used in the function definition to pass variable-length arguments.

- It is actually a tuple of the variable names and its value.

****kwargs**

- **kwargs is a special syntax used in the function definition to pass variable-length keyworded arguments.
- Here, also, “kwargs” is used just by convention. You can use any other name.
- It is actually a dictionary of the variable names and its value.

▼ What is break, continue and pass in Python?

Break	The break statement terminates the loop immediately and the control flows to the statement after the body of the loop.
Continue	The continue statement terminates the current iteration of the statement, skips the rest of the code in the current iteration and the control flows to the next iteration of the loop.
Pass	As explained above, the pass keyword in Python is generally used to fill up empty blocks and is similar to an empty statement represented by a semi-colon in languages such as Java, C++, Javascript, etc.

```
pat = [1, 3, 2, 1, 2, 3, 1, 0, 1, 3]
for p in pat:
  pass
  if (p == 0):
     current = p
     break
  elif (p % 2 == 0):
     continue
  print(p)    # output ⇒ 1 3 1 3 1
print(current)    # output ⇒ 0
```

▼ What is the use of self in Python?.

self represents the instances of class . These handy keywords allow you to access the variables, attributes and methods of a defined class in Python. Self parameter doesn't have to be you named ‘self’ as you can call it by any other home however the self
parameter must always be the first parameter of any class function regardless of a name choosen

▼ What are access modifier in python ?

Public Members- Member variables of a class are public by default, which means the data members and methods can be accessed outside or inside of a class.

Private Members- The private data members are accessible only within a class and are denoted with a double underscore prefix before their name.

Protected Members- These data members are accessible within a class and their subclasses and are denoted by a single underscore prefix before the name.

▼ What are modules and packages in Python ?

Modules

They are simply Python files with a .py extension and can have a set of functions, classes, or variables defined and implemented. It allows you to organize related code and reuse it across programs by importing it.

Here's an example of a module named `shapes.py`:

```
# shapes.py
def draw_square():
    print("Drawing a square")
```

You can import this module into another Python file using the `import` statement:

```
# main.py
import shapes

shapes.draw_square()  # Output: Drawing a square
```

Packages

A package in Python is a directory that contains multiple module files and a special `__init__.py` file. It's a way to organize related modules into a hierarchical structure.

Here's a `drawing` package containing `shapes.py`:

```
drawing/
├── __init__.py
└── shapes.py
```

You can import modules from the package:

```
from drawing import shapes

shapes.draw_square()  # Output: Drawing a square
```

▼ What are lists and tuples? What is the key difference between the two?

Lists are represented with **square brackets** `['sara', 6, 0.19]`, while tuples are represented with **parantheses** `('ansh', 5, 0.97)`.

But what is the real difference between the two? The key difference between the two is that while **lists are mutable**, **tuples** on the other hand are **immutable** objects. This means that lists can be modified, appended or sliced on the go but tuples remain constant and cannot be modified in any manner. You can run the following example on Python IDLE to confirm the difference:

▼ What is PEP 8 and why is it important ?

PEP stands for **Python Enhancement Proposal**. It is a set of rules that specify how to write and design Python code for maximum readability.

PEP 8 is especially important since it documents the style guidelines for Python Code. Apparently contributing to the Python open-source community requires you to follow these style guidelines sincerely and strictly.

▼ What is PIP ?

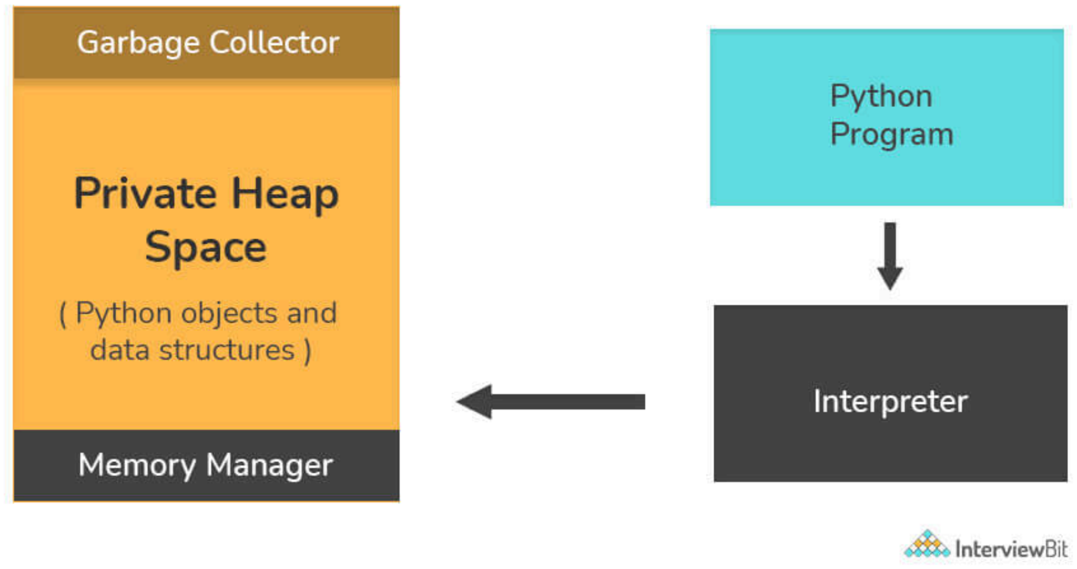
PIP is an acronym for Python Installer Package which provides a seamless interface to install various Python modules. It is a command-line tool that can search for packages over the internet and install them without any user interaction.

▼ What is decorators in python ?

Python decorators allow you to modify or extend the behavior of functions and methods without changing their actual code.

▼ How is memory managed in Python ?

- Memory management in Python is handled by the **Python Memory Manager**. The memory allocated by the manager is in form of a **private heap space** dedicated to Python. All Python objects are stored in this heap and being private, it is inaccessible to the programmer. Though, python does provide some core API functions to work upon the private heap space.
- Additionally, Python has an in-built garbage collection to recycle the unused memory for the private heap space.



▼ How are arguments passed by value or by reference in python?

- **Pass by value**: Copy of the actual object is passed. Changing the value of the copy of the object will not change the value of the original object.
- **Pass by reference**: Reference to the actual object is passed. Changing the value of the new object will change the value of the original object.

  Depending on the type of object you pass in the function, the function behaves differently. Immutable objects show "pass by value" whereas mutable objects show "pass by reference".

  ```
  def call_by_value(x):
      x = x * 2
      print("in function value updated to", x)
      return

  def call_by_reference(list):
      list.append("D")
      print("in function list updated to", list)
      return

  my_list = ["E"]
  num = 6
  print("number before=", num)
  call_by_value(num)
  print("after function num value=", num)
  ```

```
print("list before",my_list)
call_by_reference(my_list)
print("after function list is ",my_list)
```

We pass an integer in function call_by_value(). Integers are immutable objects hence Python works according to call by value, and the changes made in the function are not reflected outside the function.

We then pass list to function by reference. In function call_by_reference() we pass a list that is an mutable object. Python works according to call by reference in this function and the changes made inside the function can also be seen outside the function.

▼ What is the difference between .py and .pyc files?

• .py files contain the source code of a program. Whereas, .pyc file contains the bytecode of your program. We get bytecode after compilation of .py file (source code). .pyc files are not created for all the files that you run. It is only created for the files that you import.

▼ What are iterators in Python ?

- An iterator is an object that allows you to traverse through an iterable.
- It keeps track of its current position in the iterable.
- __iter__() method initializes an iterator.
- Iterators have a `__next__()` method, which returns the next element in the sequence.
- When there are no more elements, the `__next__()` method raises a `StopIteration` exception.

```
iter_obj=iter([3,4,5]) # This creates an iterator.

next(iter_obj) # 3
next(iter_obj) # 4
```

▼ What are generators in python ?

A generator is a type of function that returns a generator object, which can return a sequence of values instead of a single result. The def keyword is commonly used to define generators. At least one yield statement is required in a generator

```
def fib(n):
  p, q = 0, 1
  while(p < n):
      yield p
      p, q = q, p + q
x = fib(10)    # create generator object

print(next(x)) // 0
print(next(x)) // 1
print(next(x)) // 1
print(next(x)) // 2
print(next(x)) // 3
print(next(x)) // 5
print(next(x)) // 8
```

▼ Explain split() and join() functions in Python?

- You can use **split()** function to split a string based on a delimiter to a list of strings.
- You can use **join()** function to join a list of strings based on a delimiter to give a single string.

```
string = "This is a string."
string_list = string.split(' ') #delimiter is 'space' character or ' '
print(string_list) #output: ['This', 'is', 'a', 'string.']
print(' '.join(string_list)) #output: This is a strin
```

▼ How do you copy an object in Python ?

In Python, we use `=` operator to create a copy of an object. You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object.

```
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]
new_list = old_list

new_list[2][2] = 9

print('Old List:', old_list)
print('ID of Old List:', id(old_list))
print('New List:', new_list)
print('ID of New List:', id(new_list))

Old List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of Old List: 140673303268168

New List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of New List: 140673303268168
```

Why do we need shallow or deep copy in python ?

Sometimes you may want to have the original values unchanged and only modify the new values or vice versa. In Python, there are two ways to create copies:

1. Shallow Copy
2. Deep Copy

To make these copy work, we use the `copy` module.

Shallow Copy

A **shallow copy** creates a new object, but **does not create copies of nested objects**. Instead, it copies references to those objects. Changes to nested elements in the copy will affect the original.

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)

old_list[1][1] = 'AA'

print("Old list:", old_list) # Old list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
print("New list:", new_list) # New list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
```

In the above program, we made changes to old_list i.e `old_list[1][1] = 'AA'`. Both sublists of old_list and new_list at index `[1][1]` were modified. This is because, both lists share the reference of same nested objects.

Result: Changes to nested objects in the copied object will affect the original object, and vice versa.

Deep Copy

A **deep copy** creates a new object and **recursively copies all nested objects**. Changes to nested elements in the copy do **not** affect the original. This means the original and the copied objects are completely independent.

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)

old_list[1][0] = 'BB'

print("Old list:", old_list) # Old list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
print("New list:", new_list) # New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, when we assign a new value to old_list, we can see only the old_list is modified. This means, both the old_list and the new_list are independent. This is because the old_list was recursively copied, which is true for all its nested objects.

Result: Changes to nested objects in the copied object do **not** affect the original object.

▼ What is pickling and unpickling ?

Python library pickle offers a feature - **serialization** out of the box. Serializing an object refers to transforming it into a format that can be stored, so as to be able to deserialize it, later on, to obtain the original object. Here, the **pickle** module comes into play.

Pickling:

- Pickling is the name of the serialization process in Python. Any object in Python can be serialized into a byte stream and dumped as a file in the memory. The process of pickling is compact but pickle objects can be compressed further. Moreover, pickle keeps track of the objects it has serialized and the serialization is portable across versions.
- The function used for the above process is `pickle.dump()`.

Unpickling:

- Unpickling is the complete inverse of pickling. It deserializes the byte stream to recreate the objects stored in the file and loads the object to memory.
- The function used for the above process is `pickle.load()`.

▼ Differentiate between new and override modifiers.

The new modifier is used to instruct the compiler to use the new implementation and not the base class function. The Override modifier is useful for overriding a base class function inside the child class.

▼ Is it possible to call parent class without its instance creation ?

Yes, it is possible if the base class is instantiated by other child classes or if the base class has a static method.

▼ Is Tuple Comprehension? If yes, how, and if not why?

```
(i for i in (1, 2, 3))
```

Tuple comprehension is not possible in Python because it will end up in a generator, not a tuple comprehension.

▼ How do you do data abstraction in Python?

Data Abstraction is providing only the required details and hides the implementation from the world. It can be achieved in Python by using interfaces and abstract classes.

▼ Points

i. list,dict,set,class are mutable objects while int,float,bool,string,tuple,numbers,unicode are immutable objects.

ii. dir() lists all the attributes and methods available for an object, making it easy to explore what it can do.

The dir() function can be used to explore available methods and attributes of built-in types like str, list, or dict.

```

>>> dir(str)  # Lists all methods and attributes of the str class

#Output
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust','rpar
tition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

```
l = [1, 2, 3]

>>> dir(l) # We are using dir() function to look into the attributes of a user define
d list.

#Output
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'po
p',
'remove', 'reverse', 'sort']
>>>
```

iii. **help()** provides detailed information about an object, including descriptions of its methods and how to use them.

Use help() to get detailed information about built-in classes like str, int, or list.

```
>>> help(str)  # Displays detailed documentation of the str class

#Output
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
```

help() function can explain how a function works, its parameters, and return values.

```
import math
help(math.sqrt)  # Shows details about the sqrt() function in math module

#Output
Help on built-in function sqrt in module math:

sqrt(x, /)
    Return the square root of x.
```

If a class or function has a **docstring**, help() displays it along with method details.

```
class Sample:
    """This is a sample class."""
    def method(self):
        """This is a sample method."""
        pass

help(Sample)  # Displays docstrings and method details of the class

#Output
Help on class Sample in module __main__:

class Sample(builtins.object)
 |  This is a sample class.
 |
 |  Methods defined here:
 |
 |  method(self)
 |      This is a sample method.
 |
 |  -----------...
```

iv. print command normally adds a space character between each argument.

The argument `sep=""` is a *keyword argument*, and its name is short for *separator*. It specifies that the other arguments should now be separated by an empty string. You can set the separator to any string you like. For example, if you wanted each argument on a separate line, you could set the separator to `"\n"`, which is the newline character:

```
print("Hi", name, "your age is", age, "years", sep="\n")

Hi
Mark
your age is
37
years
```

By default, the print command always ends in a newline character, but you can change this as well. The keyword argument `end` specifies what is put at the end of a line. Setting `end` to an empty string means that there is no newline character at the end of the printout:

```
print("Hi ", end="")
print("there!")

Hi there!
```

v. f-string

f-string mainly used for setting the number of decimals that are printed out with a floating point number

The specific format we want the number to be displayed in can be set within the curly brackets of the variable expression. Let's add a colon character and a *format specifier* after the variable name:

```
number = 1/3
print(f"The number is {number:.2f}")

The number is 0.33
```

vi. Find address of variable in computer memory

```
a = [1, 2, 3]
print(id(a))
b = "This is a reference, too"
print(id(b))

4538357072
4537788912
```

vii. Dictionary setdefault()

The `setdefault()` method returns the value of the item with the specified key.

If the key does not exist, insert the key, with the specified value.

```
car = {
        "brand": "Ford",
        "model": "Mustang",
        "year": 1964
}
x = car.setdefault("color", "White")
print(x)
```

```
White
```

viii. **What does if __ name __ == '__ main __' do in Python?**

The "if __ name __ == '__ main __'" statement in Python checks if the current script is being run directly as the main program, or if it's being imported as a module into another program. `__name__` is a variable that exists in every Python module, and is set to the name of the module.

ix. Class & Object

i. Class is a template for creating objects & Object is an instance of class.

ii. Object can be passed as arguments and return values just like any other object.

iii. Any Python object can also be stored in a dictionary or any other data structure.

```
class Student:
    def __init__(self, name: str, cr: int):
        self.name = name
        self.cr = cr

if __name__ == "__main__":
    # The key in this dictionary is the student number,
    # and the value is an object of type Student
    students = {}
    students["12345"] = Student("Saul Student", 10)
    students["54321"] = Student("Sally Student", 67)
```

x. Class variables are accessed through the name of the class, while instance variables are accessed through the name of the object variable. An instance variable naturally only exists when an instance of the class has been created, but a class variable is available everywhere and at any point in time where the class itself is available. Class variables are useful when there is need for values which are shared by all instances of the class.

Instance Variable	**Class Variable**
Instance variables are not shared by objects. Every object has its own copy of the instance attribute.	Class variables are shared by all instances.
Instance variables are declared inside the constructor i.e., the `__init__()` method.	Class variables are declared inside the class definition but outside any of the instance

	methods and constructors.
It gets created when an instance of the class is created.	It is created when the program begins to execute.
Changes made to these variables through one object will not reflect in another object.	Changes made in the class variable will reflect in all objects.

xi. Self vs cls

Instance Method receives first argument as self representing the instance while class method receive cls argument as first argument representing the class.

xii. **str** vs repr method

str returns a string representation of the object while repr returns a *technical* representation of the object. When you use `print()` or `str()` on an object, Python calls this method.

str() displays today's date in a way that the user can understand the date and time. **repr() prints** an "official" representation of a date-time object (means using the "official" string representation we can reconstruct the object). The goal of `__repr__` is to return a string that, when passed to `eval()`, will recreate the object.

repr function should return a string that, ideally, could be used to recreate the object.

```
import datetime
today = datetime.datetime.now()

# Prints readable format for date-time object
print(str(today))   # 2016-02-22 19:32:04.078030

# prints the official format of date-time object
print(repr(today))  # datetime.datetime(2016, 2, 22, 19, 32, 4, 78030)
```

xiii. **iter()** vs next() method

- *The `__iter__()` method in Python is used to make an object iterable. When you define `__iter__()` in a class, it should return an iterator object, which is an object that implements the `__next__()` method. This allows you to use the object in loops, like `for` loops.*
- *The `__next__()` method is used to get the next item from an iterator. When `__next__()` is called, it should return the next item in the sequence. If there are no more items, it should raise a `StopIteration` exception to signal that the iteration is complete.*

```
class MyRange:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.end:
            raise StopIteration
        current = self.current
        self.current += 1
        return current


my_range = MyRange(1, 5)
for num in my_range:
    print(num)  # Output: 1, 2, 3, 4
```

xiv. getattr() vs setattr() vs hasattr() vs delattr()

1. getattr()

   The `getattr()` function returns the value of the specified attribute from the specified object.

Syntax: getattr(*object*, *attribute*, *default*)

Use the "default" parameter to write a message when the attribute does not exist

```
class Person:
  name = "John"
  age = 36
  country = "Norway"

x = getattr(Person, 'name')
y = getattr(Person, 'page', 'my message')
print(x)
print(y)
```

```
#Output
John
my message
```

2. setattr()

   The `setattr()` function sets the value of the specified attribute of the specified object.

   ### Syntax: setattr(*object*, *attribute*, *value*)

   We are changing the value of the "age" property of the "person" object.

```
class Person:
  name = "John"
  age = 36
  country = "Norway"

setattr(Person, 'age', 40)

# The age property will now have the value: 40
x = getattr(Person, 'age')

print(x)

#Output
40
```

3. hasattr()

   The `hasattr()` function returns `True` if the specified object has the specified attribute, otherwise `False`.

   ### Syntax: hasattr(object, attribute)

   Check if the "Person" object has the "age" property:

```
class Person:
  name = "John"
  age = 36
  country = "Norway"
```

```
x = hasattr(Person, 'age')

print(x)

#Output
True
```

4. delattr()

   The `delattr()` function will delete the specified attribute from the specified object.

   ### Syntax: delattr(*object*, *attribute*)

   Delete the "age" property from the "person" object:

```
class Person:
  name = "John"
  age = 36
  country = "Norway"

delattr(Person, 'age')

print(Person.__dict__)

# The Person object will no longer contain an "age" property
{'name': 'John', 'country': 'Norway'}
```

xv. factory method

It is used for creating objects instead of using a direct constructor call (new operator).

```
class User:
    def __init__(self, first_name, last_name, email, role):
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.role = role

    @classmethod
    def create_admin(cls, first_name, last_name):
        """Factory method for creating admin users"""
        email = f"{first_name.lower()}.{last_name.lower()}@admin.com"
        return cls(first_name, last_name, email, "admin")


    def __str__(self):
        return f"{self.first_name} {self.last_name} ({self.role})"

# Creating different types of users
admin = User.create_admin("John", "Doe")

print(f"Admin user: {admin}")

#Output
Admin user: John Doe (admin)
```

xvi. Method vs Function

Method	Function
A piece of code defined inside a class and must be called on an object of that class.	A block of code defined outside any class and can be called on its own.
Called on an object or within class context.	Called independently or pass arguments directly, no object needed.
They are dependent on classes	They are independent of classes
Can access and modify class attributes.	Can’t access or modify class attributes.
Example:- list.clear()	Example:- print()

xvii. M**ethod Resolution Order(MRO)**: MRO is the sequence in which Python looks for a method in a hierarchy of classes.

In Python's MRO, once it finds a method (in this case, in class **First**), it doesn't look any further along the inheritance chain. Therefore, it skips **Second** because **First** already provided the **greet** method.

```
class Base:
    def greet(self):
        return "Hello from Base"

class First(Base):
    def greet(self):
        return "Hello from First"

class Second(Base):
    def greet(self):
        return "Hello from Second"

class Third(First, Second):
    pass

third_obj = Third()
print(third_obj.greet())  # Output: Hello from First
print(Third.__mro__)
# Output: (<class '__main__.Third'>, <class '__main__.First'>, <class '__main__.Seco
```

In the above example the methods that are invoked is from class **First** but not from class **Second**, and this is due to Method Resolution Order(MRO).

The order that follows in the above code is- class **First** – > class **Second** .

xviii. isstance() vs issubclass()

isinstance()

The `isinstance()` function returns `True` if the specified object is of the specified type, otherwise `False`.

If the type parameter is a tuple, this function will return `True` if the object is *one* of the types in the tuple.

Syntax

isinstance(*object*, *type* | *class* | *tuple of types and/or classes*)

```
# Check if "Hello" is one of the types described in the type parameter:
x = isinstance("Hello", (float, int, str, list, dict, tuple))

#Output
True

# Check if y is an instance of myObj:
class myObj:
  name = "John"

y = myObj()

x = isinstance(y, myObj) #True
```

issubclass()

The `issubclass()` function returns `True` if the specified object is a subclass of the specified object, otherwise `False`.

Syntax

issubclass(*object*, *subclass | tuple of class objects*)

```
# Check if the class myObj is a subclass of myAge:

class myAge:
  age = 36

class myObj(myAge):
  name = "John"
  age = myAge

x = issubclass(myObj, myAge) #True
```

xix. Duck Typing

Duck typing in Python is a programming concept that focuses on an object's behavior (methods and properties) rather than its actual type. The idea comes from the saying:

"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

In Python, this means you don’t need to check the type of an object. Instead, you just use the object as long as it behaves the way you expect.

```
class Duck:
    def quack(self):
        return "Quack quack!"

class Person:
    def quack(self):
        return "I'm not a duck, but I can quack!"



# Function that expects an object with a quack method
def make_it_quack(thing):
    return thing.quack()

# Testing duck typing
duck = Duck()
person = Person()

print(make_it_quack(duck))    # Output: Quack quack!
print(make_it_quack(person))  # Output: I'm not a duck, but I can quack!
```

This demonstrates duck typing: Python checks for the presence of the quack method at runtime, not the object's type.

xx. Static Typing vs Dynamic Typing

Feature	Static Typing	Dynamic Typing
Type Checking	Data type check at compile-time	Data type check at runtime
Type Declaration	Explicitly declared (e.g., String name)	Implicit, inferred (e.g., var name)
Performance	Faster (types resolved before execution)	Slower (types resolved during execution)
Error Detection	Early (during compilation)	Late (during runtime)
Flexibility	Variable cannot change type	Variable can change the type
Examples	C, Java, TypeScript	Python, JavaScript, Ruby
Code Readability	More verbose (explicit types)	Concise (no type declarations)
Use Case	Large, performance-critical systems	Rapid prototyping, scripting

▼ Remaining Concepts

- [ ] DSA https://medium.com/@devulapellisaikumar/understanding-data-structures-and-algorithms-in-python-a-beginners-guide-70d7f9c65c8b - later
- [ ] Python Interview Questions - later
- [ ] write notes & program for python learning - later

▼ Python Notes

▼ Primitive Data Types

Primitive data types are the basic, immutable, and fundamental building blocks in Python. They store single, simple values and cannot be broken down further.

1. **Integer (int)**
    - Represents whole numbers (positive, negative, or zero).
    - Example: x = 5, y = -10
    - No size limit in Python (limited only by system memory).
2. **Float (float)**
    - Represents decimal or floating-point numbers.
    - Example: x = 3.14, y = -0.001
    - Supports scientific notation, e.g., 1e-10.
3. **Boolean (bool)**
    - Represents truth values: True or False.
    - Example: is_valid = True, has_error = False
    - Used in logical operations and conditionals.
4. **String (str)**
    - Represents a sequence of characters enclosed in quotes (', ", or """).
    - Example: name = "Alice", greeting = 'Hello!'
    - Immutable, but can be indexed and sliced.

Key Characteristics:

- **Immutable**: Values cannot be changed in place (e.g., modifying a string creates a new string).

- Stored as single values in memory.
- Simple and efficient for basic operations.

```
x = 5
print(id(x))  # Memory address of x, e.g., 140735674688784

x = x + 1  # This doesn't modify 5; it creates a new integer object 6
print(x)   # Output: 6
print(id(x))  # New memory address, e.g., 140735674688816
```

▼ Non-Primitive Data Types

Non-primitive data types are complex, mutable (or sometimes immutable), and can store multiple values or more complex data structures. They are often referred to as **compound** or **derived** data types.

1. **List**
    - Ordered, mutable collection of items (can store mixed data types).
    - Example: my_list = [1, "apple", 3.14]
    - Supports indexing, slicing, and methods like append(), remove().
2. **Tuple**
    - Ordered, **immutable** collection of items.
    - Example: my_tuple = (1, "banana", 2.0)
    - Faster than lists due to immutability; used when data shouldn’t change.
3. **Dictionary (dict)**
    - Unordered collection of key-value pairs.
    - Example: my_dict = {"name": "Bob", "age": 25}
    - Keys must be immutable (e.g., strings, numbers); values can be any type.
    - Accessed via keys, e.g., my_dict["name"].
4. **Set**
    - Unordered collection of unique items.
    - Example: my_set = {1, 2, 3, 3} (stores as {1, 2, 3})
    - Mutable, but elements must be immutable.
    - Supports mathematical operations like union (|), intersection (&).

Key Characteristics:

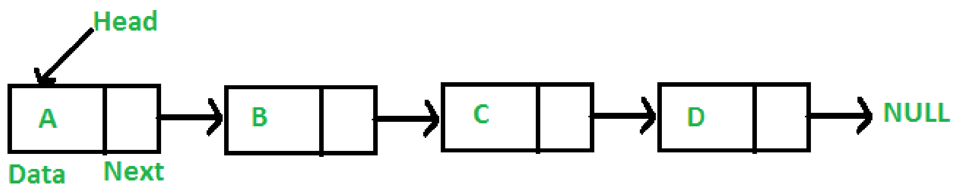
- **Mutable** (except tuples): Contents can be modified (e.g., adding an item to a list).
- Can store multiple values or complex data.
- More flexible but may consume more memory and processing.

▼ User-defined data types

- **Linked list**
- **Stack**
- **Queue**
- **Tree**
- **Graph**
- **Hashmap**

1. **Linked list**

   A **linked list** is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



   A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL. Each node in a list consists of at least two parts:

   - Data
   - Pointer (Or Reference) to the next node

```
# A simple Python program to introduce a linked list

# Node class
class Node:
    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null


# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print (temp.data)
            temp = temp.next


# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    list = LinkedList()

    list.head = Node(1)
    second = Node(2)
    third = Node(3)

    '''
    Three nodes have been created.
    We have references to these three blocks as head,
    second and third

    list.head     second           third
        |            |                 |
        |         |            |
    +----+------+    +----+------+    +----+------+
    | 1 | None |     | 2 | None |     | 3 | None |
    +----+------+    +----+------+    +----+------+
    '''

    list.head.next = second; # Link first node with second

    '''
    Now next of first Node refers to second. So they
    both are linked.

    list.head    second          third
        |         |            |
        |         |            |
    +----+------+    +----+------+    +----+------+
    | 1 | o-------->| 2 | null |     | 3 | null |
    +----+------+    +----+------+    +----+------+
    '''

    second.next = third; # Link second node with the third node

    '''
    Now next of second Node refers to third. So all three
    nodes are linked.

    list.head    second          third
        |         |            |
        |         |            |
    +----+------+    +----+------+    +----+------+
    | 1 | o-------->| 2 | o-------->| 3 | null |
    +----+------+    +----+------+    +----+------+
    '''
```

2. **Stack**

   A **stack** is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.

The functions associated with stack are:

- **empty() –** Returns whether the stack is empty – Time Complexity: O(1)
- **size() –** Returns the size of the stack – Time Complexity: O(1)
- **top() –** Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- **push(a) –** Inserts the element 'a' at the top of the stack – Time Complexity: O(1)
- **pop() –** Deletes the topmost element of the stack – Time Complexity: O(1)

Python Stack Implementation

Stack in Python can be implemented using the following ways:

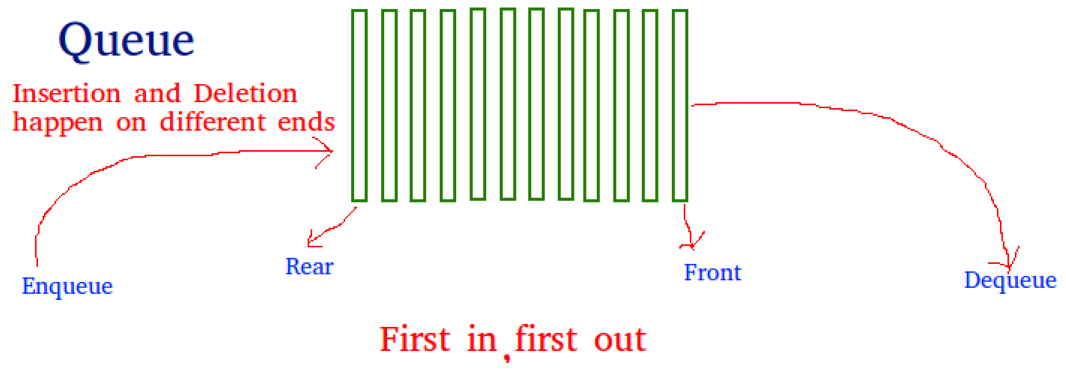
- list
- Collections.deque
- queue.LifoQueue

```
stack = ['first', 'second', 'third']
print(stack)

print()

# pushing elements
stack.append('fourth')
stack.append('fifth')
print(stack)
print()

# printing top
n = len(stack)
print(stack[n-1])
print()

# popping element
```

```
stack.pop()
print(stack)
```

3. **Queue**

As a stack, the **queue** is a linear data structure that stores items in a First In First Out (FIFO) manner. With a queue, the least recently added item is removed first. A good example of the queue is any queue of consumers for a resource where the consumer that came first is served first.



Operations associated with queue are:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity: O(1)
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition – Time Complexity: O(1)
- **Front:** Get the front item from queue – Time Complexity: O(1)
- **Rear:** Get the last item from queue – Time Complexity: O(1)

Python queue Implementation

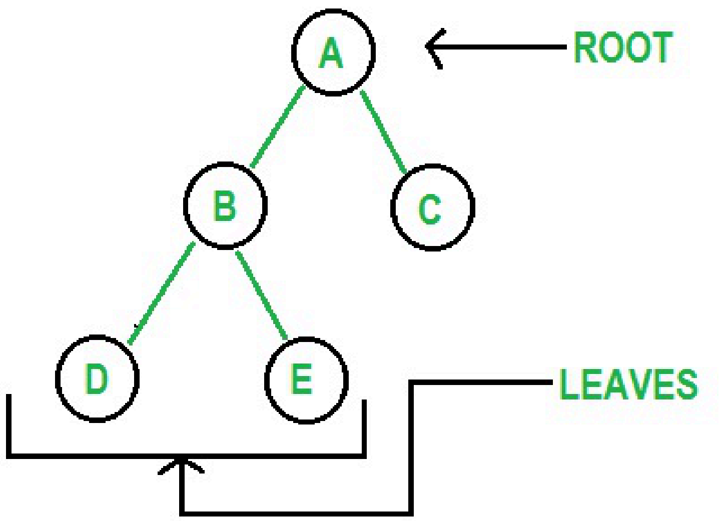
Queue in Python can be implemented in the following ways:

- list
- collections.deque
- queue.Queue

```
queue = ['first', 'second', 'third']
print(queue)
print()

# pushing elements
queue.append('fourth')
queue.append('fifth')
print(queue)
print()

# printing head
print(queue[0])

# printing tail
n = len(queue)
print(queue[n-1])
print()

# popping element
queue.remove(queue[0])
print(queue)
```

4. **Tree**

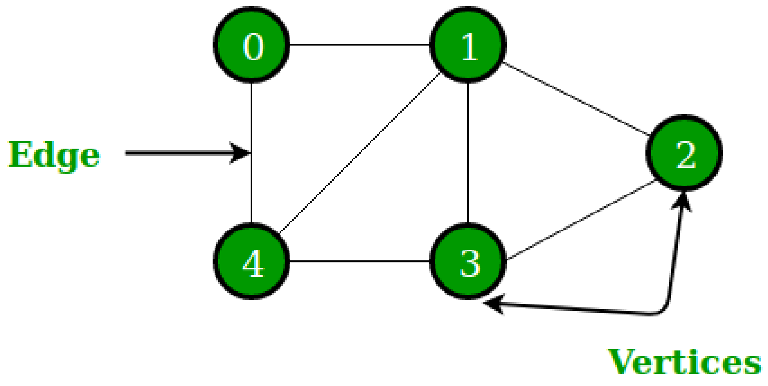
   A **tree** is a non-linear but hierarchical data structure. The topmost element is known as the root of the tree since the tree is believed to start from the root. The elements at the end of the tree are known as its leaves. Trees are appropriate for storing data that aren't linearly connected to each other but form a hierarchy.



```
class node:
    def __init__(self, ele):
        self.ele = ele
        self.left = None
        self.right = None


def preorder(self):
    if self:
        print(self.ele)
        preorder(self.left)
        preorder(self.right)


n = node('first')
n.left = node('second')
n.right = node('third')
preorder(n)
```

5. **Graph**

A **Graph** is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.



```
class adjnode:
    def __init__(self, val):
        self.val = val
        self.next = None


class graph:
    def __init__(self, vertices):
        self.v = vertices
        self.ele = [None]*self.v

    def edge(self, src, dest):
        node = adjnode(dest)
        node.next = self.ele[src]
        self.ele[src] = node

        node = adjnode(src)
        node.next = self.ele[dest]
        self.ele[dest] = node

    def __repr__(self):
        for i in range(self.v):
            print("Adjacency list of vertex {}\n head".format(i), end="")
            temp = self.ele[i]
        while temp:
            print(" → {}".format(temp.val), end="")
            temp = temp.next


g = graph(4)
g.edge(0, 2)
g.edge(1, 3)
g.edge(3, 2)
g.edge(0, 3)
g.__repr__()
```

6. **Hashmap**

   Hash maps are indexed data structures. A hash map makes use of a **hash function** to compute an index with a key into an array of buckets or slots. Its value is mapped to the bucket with the corresponding index. The key is unique and immutable. In Python, dictionaries are examples of hash maps.

   ```
   def printdict(d):
       for key in d:
           print(key, "→", d[key])


   hm = {0: 'first', 1: 'second', 2: 'third'}
   printdict(hm)
   print()

   hm[3] = 'fourth'
   printdict(hm)
   print()

   hm.popitem()
   printdict(hm)
   ```

▼ String(Immutable,Ordered,Allow Duplicates,Indexed)

All elements (characters) inside a string are **immutable**.

Usage:-

- Storing user input (e.g., names, emails).
- Processing text data (e.g., formatting messages, parsing logs).
- Generating dynamic content (e.g., emails, notifications).

1. Use operator + and * for concatenation

```
begin = "ex"
end = "ample"
word = begin+end
print(word)

example
```

```
word = "banana"
print(word*3)


bananabananabanana
```

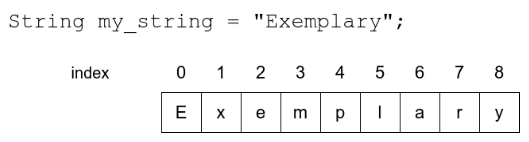
2. Length & index of string

▼ The function `len` returns the number of characters in a string, which is always an integer value.

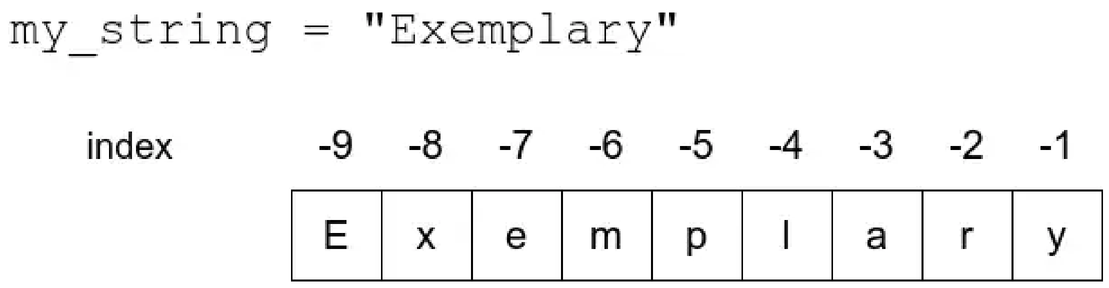
```
string = bye bye
print(len(input_string))
```

```
7
```

▼ The index refers to a position in the string, counting up from zero. The first character in the string has index 0, the second character has index 1, and so forth



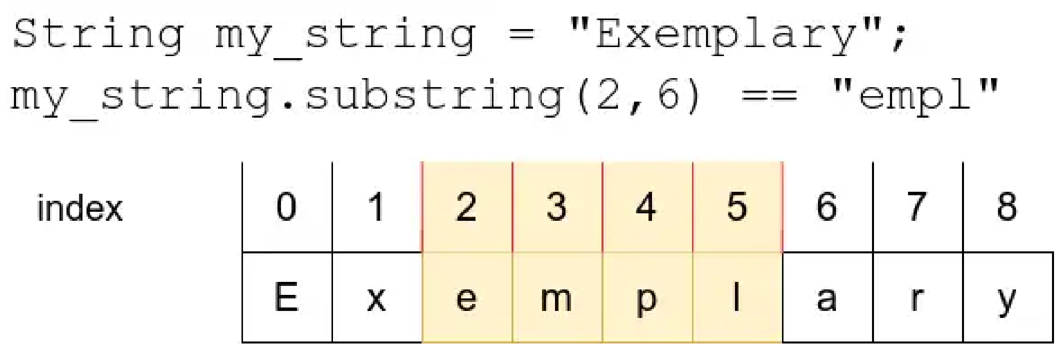
▼ You can also use negative indexing to access characters counting from the end of the string. The last character in a string is at index -1, the second to last character is at index -2, and so forth. You can think of `input_string[-1]` as shorthand for `input_string[len(input_string) - 1]`



3. Slicing

▼ In Python programming, the process of selecting substrings is usually called *slicing.*

  This means the slice begins at the index `a` and ends at the last character before index `b` - that is, including the first, but excluding the last.

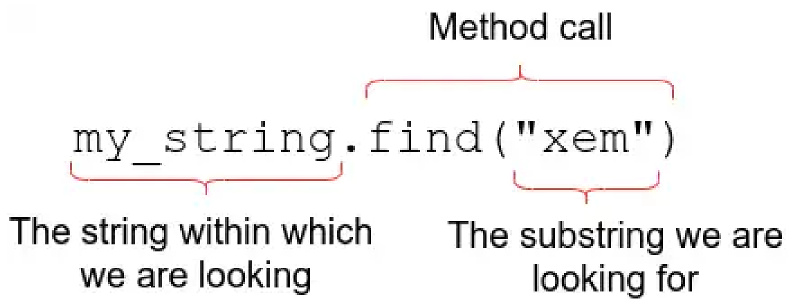

String my_string = "Exemplary";
my_string.substring(2,6) == "empl"
index
0 1 2 3 4 5 6 7 8
E x e m p l a r y

4. Searching for substring

▼ The `in` operator can tell us if a string contains a particular substring. The Boolean expression `a in b` is true, if `b` contains the substring `a`.

```
input_string = "test"

print("t" in input_string)
print("x" in input_string)
print("es" in input_string)
print("ets" in input_string)


True
False
True
False
```

▼ The operator `in` returns a Boolean value, so it will only tell us *if* a substring exists in a string, but it will not be useful in finding out *where* exactly it is. Instead, the Python string method `find` can be used for this purpose. It takes the substring searched for as an argument, and returns either the first index where it is found, or `-1` if the substring is not found within the string.


Method call
my_string.find("xem")
The string within which we are looking
The substring we are looking for

```
5
input_string = "test"

print(input_string.find("t"))
print(input_string.find("x"))
print(input_string.find("es"))
print(input_string.find("ets"))



0
-1
1
-1
```

5. count & replace

   The method `count` counts the number of times the specified item or substring occurs in the target.

   ```
   my_string = "How much wood would a woodchuck chuck if a woodchuck co
   uld chuck wood"
   print(my_string.count("ch"))

   5
   ```

   replace

   The method `replace` creates a new string, where a specified substring is replaced with another string:

   ```
   my_string = "Python is fun"

   # Replaces the substring but doesn't store the result...
   my_string.replace("Python", "Java")
   print(my_string)

   Python is fun
   ```

▼ List(Mutable,Ordered,Allow Duplicates,Indexed)

A list can hold **any type of object**, including:

- Immutable types (e.g., integers, strings, tuples)
- Mutable types (e.g., other lists, dictionaries, sets)

Usage:-

- Managing a collection of items (e.g., shopping cart items, to-do lists).
- Storing and processing real-time data (e.g., sensor readings, user activity logs).
- Iterating over data for analysis or display.

1. Accessing items in list

   A single list item can be accessed just like a single character in a string is accessed, with square brackets

   ```
   my_list = [7, 2, 2, 5, 2]

   print(my_list[0])
   print(my_list[1])
   print(my_list[3])

   print("The sum of the first two items:", my_list[0] + my_list[1])
   ```

   slicing

   In fact, the slice [] syntax works very similarly to the `range` function, which means we can also give it a step:

   ```
   my_string = "exemplary"
   print(my_string[0:7:2])
   my_list = [1,2,3,4,5,6,7,8]
   print(my_list[6:2:-1])

   eepa
   [7, 6, 5, 4]
   ```

2. Adding items to a list

   ```
   numbers = []
   numbers.append(5)
   numbers.append(10)
   numbers.append(3)
   print(numbers)
   ```

```
[5, 10, 3]

a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9, 10]

a.extend(b)
[1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

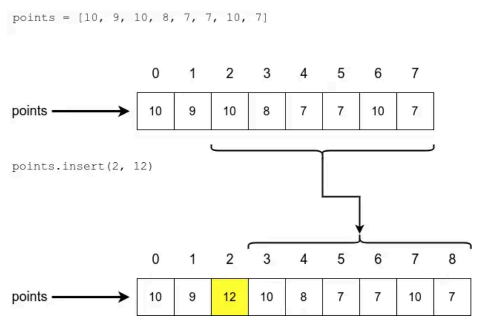
3. Get item index from list

   index(value, [startIndex]) – gets the index of the first occurrence of the input value. If the input value is not in the list a ValueError exception is raised. If a second argument is provided, the search is started at that specified index

   ```
   a = [1, 2, 3, 4, 5, 6, 7, 7]
   a.index(7) # 6

   a.index(7, 7) # 7
   ```

4. Add items to specific location

   If you want to specify a location in the list where an item should be added, you can use the `insert` method. The method adds an item at the specified index. All the items already in the list with an index equal to or higher than the specified index are moved one index further, "to the right"



4. Removing items from list

   There are two different approaches to removing an item from a list:

   - If the *index* of the item is known, you can use the method `pop`.
   - If the *contents* of the item are known, you can use the method `remove`.

So, the method `pop` takes the index of the item you want to remove as its argument. The following program removes items at indexes 2 and 3 from the list. Notice how the indexes of the remaining items change when one is removed.

Pop method returns removed item

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

my_list.pop(2)
print(my_list)
my_list.pop(3)
print(my_list)

[1, 2, 4, 5, 6]
[1, 2, 4, 6]
```

The method `remove`, on the other hand, takes the value of the item to be removed as its argument. For example, this program

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

my_list.remove(2)
print(my_list)
my_list.remove(5)
print(my_list)

[1, 3, 4, 5, 6]
[1, 3, 4, 6]
```

5. Sorting items in a list

   i. sort()

      The items in a list can be *sorted* from smallest to greatest with the method `sort`. `sort` changes the order of the original list in place.

      ```
      my_list = [2,5,1,2,4]

      my_list.sort()
      print(my_list)
      ```

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

ii. sorted function

Sometimes we don't want to change the original list, so we use the function `sorted` instead. It *returns* a sorted list:

Syntax of sorted() function

sorted(iterable, key=None, reverse=False)

- **iterable:** The sequence to be sorted. This can be a list, tuple, set, string, or any other iterable.
- **key (Optional):** A function to execute for deciding the order of elements. By default it is **None**
- **reverse (Optional):** If **True**, sorts in descending order. Defaults value is **False** (ascending order)

Sorting in ascending order

```
my_list = [2,5,1,2,4]
print(sorted(my_list))

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

Sorting in descending order

```
a = [5, 2, 9, 1, 5, 6]

# "reverse= True" helps sorted() to arrange the element
#from largest to smallest elements
res = sorted(a, reverse=True)
print(res)

[9, 6, 5, 5, 2, 1]
```

Sorting using key parameter

The **key** parameter is an optional argument that allows us to customize the sort order based on provided function.

```
a = ["apple", "banana", "cherry", "date"]
res = sorted(a, key=len)
print(res) #['date', 'apple', 'banana', 'cherry']

# sorting a list of dictionaries
a = [
    {"name": "Alice", "score": 85},
    {"name": "Bob", "score": 91},
    {"name": "Eve", "score": 78}
]
b = sorted(a, key=lambda x: x['score'])
print(b) # [{'name': 'Eve', 'score': 78}, {'name': 'Alice', 'score': 85}, {'name':
```

6. Maximum, minimum and sum

   The functions `max` and `min`, short for maximum and minimum, return the greatest and smallest item in a list, respectively. The function `sum` returns the sum of all items in a list.

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

greatest = max(my_list)
smallest = min(my_list)
list_sum = sum(my_list)

print("Smallest:", smallest)
print("Greatest:", greatest)
print("Sum:", list_sum)

Smallest: 1
Greatest: 5
Sum: 15
```

7. count

The method `count` counts the number of times the specified item or substring occurs in the list

```
my_list = [1,2,3,1,4,5,1,6]
print(my_list.count(1))

3
```

8. reverse list

   reverse() – reverses the list in-place and returns None.

```
a.reverse()

[8, 7, 6, 5, 4, 3, 2, 1]

a[::-1] # [8, 7, 6, 5, 4, 3, 2, 1]
```

9. copy()

   returns a copy of the specified list.

```
fruits = ['apple', 'banana', 'cherry', 'orange']
x = fruits.copy()

print(x)

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

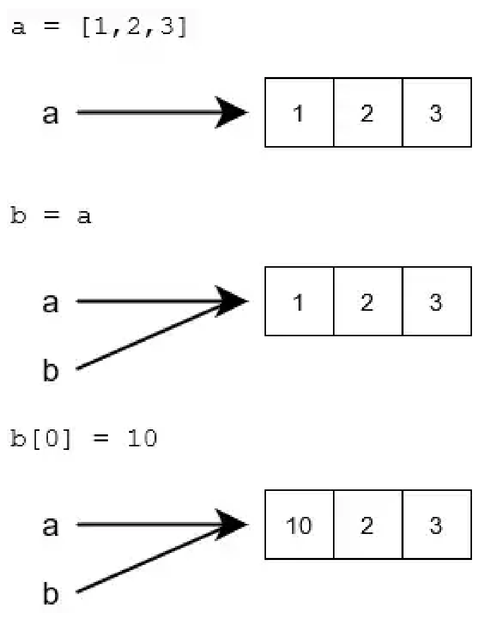
10. clear()

    removes all items from the list

```
a = [1, 2, 3, 4, 5, 6]
a.clear()
print(a)

[]
```

11. Multiple reference to the same list

So, the assignment `b = a` copies the reference. As a result there are now two references to the same memory location containing the list.



```
list1 = [1, 2, 3, 4]
list2 = list1

list1[0] = 10
list2[1] = 20

print(list1)
print(list2)

[10, 20, 3, 4]
[10, 20, 3, 4]
```

12. Copying a list

    An easier way to copy a list is the bracket operator `[]`, which we used for slices previously. The notation `[:]` selects all items in the collection. As a side effect, it creates a copy of the list:

```
my_list = [1,2,3,4]
new_list = my_list[:]
```

```
my_list[0] = 10
new_list[1] = 20

print(my_list)
print(new_list)



[10, 2, 3, 4]
[1, 20, 3, 4]
```

13. Check if list is empty

    ```
    lst = []
    if not lst:
        print("list is empty")

    # Output: list is empty
    ```

14. any() and all()

    all() - if all the values in an iterable evaluate to True

    ```
    nums = [1, 1, 0, 1]
    all(nums)
    # False
    ```

    any() - if one or more values in an iterable evaluate to True

    ```
    nums = [1, 1, 0, 1]
    any(nums)
    # True
    ```

15. Remove duplicates values in list

    ```
    names = ["aixk", "duke", "edik", "tofp", "duke"]
    list(set(names))
    # Out: ['duke', 'tofp', 'aixk', 'edik']
    ```

▼ Dictionary(Mutable,Ordered,Not Allow Duplicates,Unindexed)

- Keys must be **immutable** (e.g., strings, numbers, tuples without mutable elements).
- Values can be **anything**, including mutable types.

Usage:-

Storing user profiles (e.g., name, age, email).Caching data for quick access (e.g., API responses).Grouping data for analysis (e.g., counting occurrences).

- Storing user profiles (e.g., name, age, email).
- Caching data for quick access (e.g., API responses).
- Grouping data for analysis (e.g., counting occurrences).

1. Adding items to dictionary
   i. my_dict['new_key'] = 42
   ii. my_dict.update({'new_key': 42})
2. Get values from dictionary

   my_dict = {}

   i. my_dict[key]
   ii. value = mydict.get(key, default_value)

      returns mydict[key] if it exists, but otherwise returns default_value. Note that this doesn't add key to mydict. So if you want to retain that key value pair, you should use mydict.setdefault(key, default_value), which does store the key value pair.

```
mydict = {}
print(mydict)
{}

print(mydict.get("foo", "bar"))
#bar

print(mydict.setdefault("foo", "bar"))
# bar
```

```
print(mydict)
# {'foo': 'bar'}
```

3. Removing keys and values from a dictionary

    i. pop(key,default return value)

```
staff = {"Alan": "lecturer", "Emily": "professor", "David": "lecturer"}
deleted = staff.pop("Paul", None)
if deleted == None:
  print("This person is not a staff member")
else:
  print(deleted, "deleted")



This person is not a staff member
```

    ii. clear()

    Remove all items from dictionary

4. Iterating over a dictionary

    The items() method can be used to loop over both the key and value simultaneously:

```
d = {'a': 1, 'b': 2, 'c':3}

for key, value in d.items():
    print(key, value)

# c 3
# b 2
# a 1
```

    the methods keys(), values() and items() return lists.

5. Merging dictionaries

```
fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}

fishdog = {**fish, **dog}
```

```
print(fishdog)

{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

6. Create dictionary
   i. Empty dictionary

      mydict = {}

   ii. Populate dictionary values with list

```
mydict = {'a': [1, 2, 3], 'b': ['one', 'two', 'three']}
```

   iii. Populate dictionary values with list of tuples

```
iterable = [('eggs', 5), ('milk', 2)]
mydict = dict(iterable)
print(mydict)

{'eggs': 5, 'milk': 2}
```

   iv. Using keyword argument

```
mydict = dict(eggs=5, milk=2)

{'eggs': 5, 'milk': 2}
```

7. Unpacking dictionaries using ** operator

```
def parrot(voltage, state, action):

    print("This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.", end=' ')
    print("E's", state, "!")

d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
parrot(**d)

This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demis
```

▼ Tuple(Immutable,Ordered,Allow Duplicates,Indexed)

- A **tuple can contain both mutable and immutable elements**.
- Tuples are ideal for when there is a set collection of values which are in some way connected. For example, when there is a need to handle the x and y coordinates of a point, a tuple is a natural choice, because coordinates will always consist of two values:

  ```
  point = (10, 20)
  ```

Usage:-

Storing fixed data (e.g., coordinates, configuration settings).Returning multiple values from a function.Using as dictionary keys (since tuples are immutable and hashable).

- Storing fixed data (e.g., gps coordinates, configuration settings).
- Returning multiple values from a function.
- Using as dictionary keys (since tuples are immutable and hashable).

1. Create tuple using parenthesis

   To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

   ```
   numbers = (1,)
   ```

   The parentheses are not strictly necessary when defining tuples

   ```
   numbers = 1, 2, 3
   ```

▼ Set(Mutable,Unordered,Not Allow Duplicates,Unindexed)

- A set can have any number of items like integer, float, tuple, string, etc. except mutable elements like lists, sets or dictionaries as its elements.
- In Python, a set is an unordered collection of unique elements i.e when we access all items, they are accessed without any specific order and we cannot access items using indexes as we do in lists.
- Unlike lists or tuples, sets do not allow duplicate values i.e. each element in a set must be unique.

- Sets are mutable, meaning you can add or remove items after a set has been created for example, by **adding** or **removing** elements i.e ou **cannot modify an element in-place** (like you could modify an item in a list by index). If you want to "change" an element in a set, you have to remove the old one and add a new one.

Usage:-

  - Removing duplicates from a dataset (e.g., unique visitors to a website).
  - Performing set operations (e.g., finding common interests between users).
  - Fast membership testing (e.g., checking if a user ID is in a banned list).

Creating a Set

You can create a set in Python using curly braces {} or the set() function –

```
my_set = {1, 2, 3, 4, 5} # using culry braces
print (my_set)

my_set = set([1, 2, 3, 4, 5]) # using set function
print (my_set)
```

Adding Elements in a Set

To add an element to a set, you can use the add() function. This is useful when you want to include new elements into an existing set. If the element is already present in the set, the set remains unchanged

```
my_set = {1, 2, 3, 3}
# Adding an element 4 to the set
my_set.add(4)
print (my_set) # {1, 2, 3, 4}
```

Update Elements Set

The update() method is used to update the set with items.

```
companies = {'Lacoste', 'Ralph Lauren'}
tech_companies = ['apple', 'google', 'apple']

# using update() method
companies.update(tech_companies)

print(companies)

# Output: {'google', 'apple', 'Lacoste', 'Ralph Lauren'}
```

Removing Elements from a Set

You can remove an element from a set using the remove() function. This is useful when you want to eliminate specific elements from the set.

```
my_set = {1, 2, 3, 4}
# Removes the element 3 from the set
my_set.remove(3)
print (my_set) # {1, 2, 4}
```

Membership Testing in a Set

You can use the **in** keyword to perform this check, which returns **True** if the element is present and **False** otherwise –

```
my_set = {1, 2, 3, 4}

print(2 in my_set) #True.
```

▼ Functions

1. Parameter

A parameter is the variable defined within the parentheses when we declare a function.

```
Here a,b are the parameters
def sum(a,b):
print(a+b)
```

2. Arguments

   An argument is a value that is passed to a function when it is called.

```
def sum(a,b):
print(a+b)

# Here the values 1,2 are arguments
sum(1,2)
```

3. **Types of arguments in python**

▼ ***Default arguments:*** *These have a default value if no argument is provided.*

```
def sum(a,b=2):
print(a+b)

# Calling the function and passing only one argument
sum(1)

3
```

▼ ***Positional arguments:*** *These are passed to functions in the same order as the parameters are defined i.e the first argument is always listed first when the function is called, second argument needs to be called second and so on.*

```
def fun(s1,s2):
  print(s1+s2)

fun("Geeks","forGeeks")
```

▼ *Keyword Arguments is an argument passed to a function or method which is preceded by a keyword and equal to sign (=). The order of keyword argument with respect to another keyword argument does not matter because the values are being explicitly assigned*

```
def fun(s1, s2):
    print(s1 + s2)


# Here we are explicitly assigning the values
fun(s2="Geeks", s1="forGeeks")
```

- ***Variable-length arguments:** These allow passing an arbitrary number of arguments, using `*args` for positional and `**kwargs` for keyword arguments.*

```
def fun(*args):
    # Concatenate all arguments
    result = "".join(args)
    print(result)

# Function calls
fun("Geeks", "forGeeks")
```

Order for defining functional parameters:-

1. **Positional parameters(a,b)**
2. **Default parameters** (parameters with default values)(c=10)
3. **Variable positional parameters** (`args`)
4. **Keyword-only parameters** (d=10)
5. **Variable keyword parameters** (`*kwargs`)

Lambda expressions

Lambda function can have any number of parameters but only one expression.

Lambda function doesn't have a name that's why they are called anonymous function.

```
lambda <parameters> : <expression>
```

For example, the following program sorts a list of strings alphabetically by the *last* character in each string:

```
strings = ["Mickey", "Mack", "Marvin", "Minnie", "Merl"]

for word in sorted(strings, key=lambda word: word[-1]):
    print(word)

Minnie
Mack
Merl
Marvin
Mickey
```

Functions as arguments within your own functions

It is possible to pass a reference to a function as an argument to another function.

```
# the type hint "callable" refers to a function
def perform_operation(operation: callable):
    # Call the function which was passed as an argument
    return operation(10, 5)

def my_sum(a: int, b: int):
    return a + b

def my_product(a: int, b: int):
    return a * b



if __name__ == "__main__":
    print(perform_operation(my_sum))
    print(perform_operation(my_product))
    print(perform_operation(lambda x,y: x - y))



15
```

```
50
5
```

Reading data from file

The variable `new_file` above is a *file handle*. Through it the file can accessed while it is still open. Here we used the method `read`, which returns the contents of the file as a single string.

```
with open("example.txt") as new_file:
    contents = new_file.read()
    print(contents)


Hello there!
This example file contains three lines of text.
This is the last line.
```

Going through contents of file

The `read` method is useful for printing out the contents of the entire file, but more often we will want to go through the file line by line.

Text files can be thought of as lists of strings, each string representing a single line in the file. We can go through the list with a `for` loop.

```
with open("example.txt") as new_file:
    count = 0
    total_length = 0

    for line in new_file:
        line = line.replace("\n", "")
        print("Line")
```

Reading csv file

```
import csv

with open("test.csv") as my_file:
    for line in csv.reader(my_file, delimiter=";"):
        print(line)

['012121212', '5']
['012345678', '2']
['015151515', '4']
```

Reading json file

```
import json

with open("courses.json") as my_file:
    data = my_file.read()

courses = json.loads(data)
print(courses)


[{'name': 'Introduction to Programming', 'abbreviation': 'ItP', 'periods': [1, 3]},
{'name': 'Advanced Course in Programming', 'abbreviation': 'ACiP', 'periods':
[2, 4]}, {'name': 'Database Application', 'abbreviation': 'DbApp', 'periods': [1,
2, 3, 4]}]
```

Python JSON to dict

You can parse a JSON string using `json.loads()` method. The method returns a dictionary.

```
import json

person = '{"name": "Bob", "languages": ["English", "French"]}'
person_dict = json.loads(person)

# Output: {'name': 'Bob', 'languages': ['English', 'French']}
print(person_dict)

# Output: ['English', 'French']
print(person_dict['languages'])
```

Python Convert to JSON string

You can convert a dictionary to JSON string using `json.dumps()` method.

```
import json

person_dict = {'name': 'Bob',
'age': 12,
'children': None
}
person_json = json.dumps(person_dict)

# Output: {"name": "Bob", "age": 12, "children": null}
print(person_json)
```

Python pretty print JSON

To analyze and debug JSON data, we may need to print it in a more readable format. This can be done by passing additional parameters `indent` and `sort_keys` to `json.dumps()` and `json.dump()` method.

```
import json

person_string = '{"name": "Bob", "languages": "English", "numbers": [2, 1.6, null

# Getting dictionary
person_dict = json.loads(person_string)

# Pretty Printing JSON string back
print(json.dumps(person_dict, indent = 4, sort_keys=True))


#Output
{
   "languages": "English",
   "name": "Bob",
   "numbers": [
        2,
        1.6,
        null
    ]
}
```

▼ Writing files

Creating a file

If you want to create a new file, you would call the `open` function with the additional argument `w`, to signify that the file should be opened in write mode. So, the function call could look like this:

```
with open("new_file.txt", "w") as my_file:
    my_file.write("Hello there!")
```

if the file already exists, all the contents will be overwritten. It pays to be very careful when creating new files.

Appending data to an existing file

The following program opens the file `new_file.txt` and appends a couple of lines of text to the end:

```
with open("new_file.txt", "a") as my_file:
    my_file.write("This is the 4th line\n")
    my_file.write("And yet another line.\n")
```

Clearing file contents

Sometimes it is necessary to clear the contents of an existing file. Opening the file in write mode and closing the file immediately will achieve just this:

```
with open("file_to_be_cleared.txt", "w") as my_file:
    pass
```

Deleting files

```
import os

os.remove("unnecessary_file.csv")
```

▼ Handling errors

```
try:
    with open("example.txt") as my_file:
        for line in my_file:
            print(line)
except FileNotFoundError:
    print("The file example.txt was not found")
except PermissionError:
    print("No permission to access the file example.txt")
```

▼ Global variable

If we want to change the value of a global variable inside a function, refer to the variable by using the `global` keyword.

```
x = "awesome"

def myfunc():
        global x
        x = "fantastic"


myfunc()
```

```
Python is fantastic
```

▼ **Times and dates**

The datetime object

The Python datetime module includes the function now, which returns a datetime object containing the current date and time. The default printout of a datetime object looks like this:

```
from datetime import datetime

my_time = datetime.now()
print(my_time)

2021-10-19 08:46:49.311393
```

You can also define the object yourself:

```
from datetime import datetime

my_time = datetime(1952, 12, 24, 18, 45, 10)
print(my_time)

1952-12-24 18:45:10
```

Compare times and calculate differences between them

The difference between two datetime objects can be calculated simply with any comparison & subtraction operator.

```
from datetime import datetime

time_now = datetime.now()
midsummer = datetime(2021, 6, 26)

difference = midsummer - time_now
print("Midsummer is", difference.days, "days away")
```

```
Midsummer is -116 days away
```

```
if time_now < midsummer:
    print("It is not yet Midsummer")
elif time_now == midsummer:
    print("Happy Midsummer!")
elif time_now > midsummer:
    print("It is past Midsummer")


It is past Midsummer
```

Formatting times and dates

strftime - convert datetime object to string format

The `datetime` module contains a handy method strftime for formatting the string representation of a datetime object. For example, the following code will print the current date in the format `dd.mm.yyyy`, and then the date and time in a different format:

```
from datetime import datetime

my_time = datetime.now()
print(my_time.strftime("%d.%m.%Y"))
print(my_time.strftime("%d/%m/%Y %H:%M"))


19.10.2021
19/10/2021 09:31
```

strptime - convert string to datetime object

```
from datetime import datetime

birthday = 5.11.1986
my_time = datetime.strptime(birthday, "%d.%m.%Y")
```

```
if my_time < datetime(2000, 1, 1):
    print("You were born in the previous millennium")
else:
    print("You were born during this millennium")


You were born in the previous millennium
```

Python sleep()

The `sleep()` method suspends the execution of the program for a specified number of seconds.

Syntax:

```
time.sleep(seconds)
```

```
import time

print("Printed immediately.")
time.sleep(2.4)
print("Printed after 2.4 seconds.")


#Output
Printed immediately.
Printed after 2.4 seconds.
```

▼ Shallow vs Deep copy

Copy an Object in Python

In Python, we use `=` operator to create a copy of an object. You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object.

Example 1: Copy using = operator

```
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]
new_list = old_list
```

```
new_list[2][2] = 9

print('Old List:', old_list)
print('ID of Old List:', id(old_list))

print('New List:', new_list)
print('ID of New List:', id(new_list))

Old List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of Old List: 140673303268168

New List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of New List: 140673303268168
```

So, if you want to modify any values in new_list or old_list, the change is visible in both.

Why do we need shallow or deep copy in python ?

Sometimes you may want to have the original values unchanged and only modify the new values or vice versa. In Python, there are two ways to create copies:

1. Shallow Copy
2. Deep Copy

To make these copy work, we use the `copy` module.

```
import copy
copy.copy(x)
copy.deepcopy(x)
```

Here, the `copy()` return a shallow copy of x. Similarly, `deepcopy()` return a deep copy of x.

Shallow Copy

A **shallow copy** creates a new object, but **does not create copies of nested objects**. Instead, it copies references to those objects. Changes to nested elements in the copy will affect the original.

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)

old_list[1][1] = 'AA'

print("Old list:", old_list) # Old list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
print("New list:", new_list) # New list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
```

In the above program, we made changes to old_list i.e `old_list[1][1] = 'AA'`. Both sublists of old_list and new_list at index `[1][1]` were modified. This is because, both lists share the reference of same nested objects.

Result: Changes to nested objects in the copied object will affect the original object, and vice versa.

Deep Copy

A **deep copy** creates a new object and **recursively copies all nested objects**. Changes to nested elements in the copy do **not** affect the original. This means the original and the copied objects are completely independent.

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)

old_list[1][0] = 'BB'

print("Old list:", old_list) # Old list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
print("New list:", new_list) # New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, when we assign a new value to old_list, we can see only the old_list is modified. This means, both the old_list and the new_list are independent. This is because the old_list was recursively copied, which is true for all its nested objects.

Result: Changes to nested objects in the copied object do **not** affect the original object.

▼ Call by Value vs Call by Reference

Call by Value

When you pass a variable to a function, you're essentially handing over the actual value of the variable itself, not the reference to the object it's pointing to. Consequently, any changes made to the variable within the function don't directly affect the original variable outside the function.

"passing by value" is possible only with the **immutable** types such as integers, floats, strings, and tuples. Even though we pass these as a reference, the object itself cannot be changed.

```
# Python code to demonstrate
# call by value
string = "Geeks"

def test(string):
    string = "GeeksforGeeks"
    print("Inside Function:", string)

# Driver's code
test(string)
print("Outside Function:", string)
```

Call by Reference

When passing arguments to functions where the reference to the real object gets passed instead of the object's actual value. This means that if you make changes to the object within the function, those changes directly affect the original object outside the function.

we can only pass the reference of **mutable** types such as lists, dictionaries, and sets, to a function.

```
# Python code to demonstrate
# call by reference

def add_more(list):
    list.append(50)
    print("Inside Function", list)
```

```
# Driver's code
mylist = [10, 20, 30, 40]

add_more(mylist)
print("Outside Function:", mylist)
```

▼ Retrieving a file from the internet

```
import urllib.request

my_request = urllib.request.urlopen("https://helsinki.fi")
print(my_request.read())
```

▼ List Comprehension

```
[<expression> for <item> in <series>]
```

```
numbers = [1, 2, 3, 6, 5, 4, 7]
strings = [str(number) for number in numbers]
```

List Comprehension with if statement

```
[<expression> for <item> in <series> if <Boolean expression>]
```

```
numbers = [1, 1, 2, 3, 4, 6, 4, 5, 7, 10, 12, 3]

even_items = [item for item in numbers if item % 2 == 0]
print(even_items)

[2, 4, 6, 4, 10, 12]
```

List Comprehension with if else statement

```
[<expression 1> if <condition> else <expression 2> for <item> in <series>]
```

```
numbers = [1, -3, 45, -110, 2, 9, -11]
abs_vals = [number if number >= 0 else -number for number in numbers]
print(abs_vals)

[1, 3, 45, 110, 2, 9, 11]
```

▼ Dictionary Comprehension

```
{<key expression> : <value expression> for <item> in <series>}
```

```
numbers = [1, 2, 3, 6, 5, 4, 7]
strings = {number: number **2 for number in numbers}

print(strings)

{1: 1, 2: 4, 3: 9, 6: 36, 5: 25, 4: 16, 7: 49}
```

▼ Generators vs Iterators

Iterator vs Iterable

Iterable:

- An iterable is any object that can be looped over.
- It is capable of returning its members one at a time.
- Examples of iterables include lists, tuples, strings, dictionaries, and sets.
- Iterables have an `__iter__()` method, which returns an iterator object.
- When you use a for loop on an iterable, Python automatically calls the iter() function to create an iterator behind the scenes.

Iterator:

- An iterator is an object that allows you to traverse through an iterable.
- It keeps track of its current position in the iterable.
- Iterators have a `__next__()` method, which returns the next element in the sequence.
- When there are no more elements, the `__next__()` method raises a `StopIteration` exception.

- Iterators are stateful, meaning they remember where they left off.

Example

```
x=[1,2,3,4] # x is an iterable


# On calling iter(x) it returns a iterator only when the x object has iter method oth
erwise it raise an exception.

y=iter(x) # y is a iterartor it support next() method

# On calling next method it returns the individual elements of the list one by one.

>>> y.next()
1
>>> y.next()
2
>>> y.next()
3
>>> y.next()
4
>>> y.next()
StopIteration
```

Iterators

An iterator is an object that is used to iterate over an iterable. Iterators are created by using the iter() function. The function next() is used to get the subsequent value from the iterator.

• All iterators are iterables, but not all iterables are iterators.

```
iter_obj=iter([3,4,5]) # This creates an iterator.
```

```
next(iter_obj) # 3
next(iter_obj) # 4
```

An example of how to create your own iterator which counts till 10 :-

```
class CountToTen:
    def __init__(self):
        self.n = 0
        self.max= 10

    def __iter__(self):
        return self

    def __next__(self):
        if self.n < self.max:
            self.n+=1
            return self.n
        else:
            raise StopIteration


count_obj = CountToTen()
count_iter = iter(count_obj)

for i in count_iter:
    print(i)

# Output
1
2
3
4
5
6
7
8
9
10
```

Generators

A generator is a type of function that returns a generator object, which can return a sequence of values instead of a single result. The def keyword is commonly used to define generators. At least one yield statement is required in a generator.

```
def nums():
  for i in range(1, 5):
      yield i

obj = nums()
print(next(obj))
print(next(obj))
print(next(obj))
print(next(obj))

1
2
3
4
```

Iterator	**Generator**
Class is used to implement an iterator	Function is used to implement a generator.
Local Variables aren't used here.	All the local variables before the yield function are stored.
Iterators are used mostly to iterate or convert other objects to an iterator using iter() function.	Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop
Iterator uses iter() and next() functions	Generator uses yield keyword
Every iterator is not a generator	Every generator is an iterator

▼ Higher Order Function

A higher-order function is a function that either, takes one or more functions as arguments or returns a function as its result

Map

The map () function returns a map **object(which is an iterator)** of the results after applying the given function to each item of a given iterable (**list**, **tuple**, etc.).

The general syntax for the map function is where `function` is the operation we want to execute on each item in the iterable

```
map(<function>, <iterabe>)
```

The `map` function returns an object of type `map`, which is iterable, and can be converted into a list:

```
def capitalize(my_string: str):
    first = my_string[0]
    first = first.upper()
    return first + my_string[1:]

test_list = ["first", "second", "third", "fourth"]

capitalized = map(capitalize, test_list)

capitalized_list = list(capitalized)
print(capitalized_list)

['First', 'Second', 'Third', 'Fourth']
```

Filter

filter() function filters them with a criterion function, which is passed as an argument. If the criterion function returns `True`, the item is selected.

```
filter(<function>, <iter>)
```

```
integers = [1, 2, 3, 5, 6, 4, 9, 10, 14, 15]

even_numbers = list(filter(lambda number: number % 2 == 0, integers))

for number in even_numbers:
    print(number)

2
6
4
```

```
10
14
```

Reduce

The **reduce** function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.This function is defined in “functools” module

```
reduce(func, iterable[, initial])
```

```
from functools import reduce

my_list = [2, 3, 1, 5]

sum_of_numbers = reduce(lambda reduced_sum, item: reduced_sum + item, my_
list, 0)

print(sum_of_numbers)

11
```

▼ PIP & pep8

PIP is a package manager for Python packages, or modules if you like.

PEP(Python Enhancement Proposal)

PEP-8 is basically the official Python style guide. It tells you what your code should ideally look like if following the convention (which is recommended).

▼ Decorators

Python decorators allow you to modify or extend the behavior of functions and methods without changing their actual code.

Why we need decorators ?

Decorators let you execute code before and after a function i.e. we want to modify function and add code , to do something before calling function and also something after function executes.

```
def a_new_decorator(a_func):
    def wrapTheFunction():
        print('I am doing some work before executing function')
        a_func()
        print('I am doing some work after executing function')
    return wrapTheFunction

@a_new_decorator
def a_function_requiring_decoration():
    print('I am the function which needs some decoration')



a_function_requiring_decoration()
#Output
I am doing some work before executing function
I am the function which needs some decoration
I am doing some work after executing function

#the @a_new_decorator is just a short way of saying:
a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoratio
```

▼ Closures

Python closure is a nested function that allows us to access variables of the outer function even after the outer function is closed.

Key Components of a Closure

1. There must be a **nested function** (a function defined inside another function).
2. The **inner function must refer to a variable** defined in the outer function.
3. The **outer function must return the inner function**.

Closures are useful for:

- Maintaining state without global variables.
- Implementing data hiding or encapsulation.
- Creating function factories or decorators.

Example 1: Closure for State Management

```
def make_counter():
    count = 0  # Free variable

    def counter():
        nonlocal count  # Modify the free variable
        count += 1
        return count

    return counter

# Create two counters
counter1 = make_counter()
counter2 = make_counter()

print(counter1())  # Output: 1
print(counter1())  # Output: 2
print(counter2())  # Output: 1
```

Example 2: Closure in a Multiplier Function Factory

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier



# Multiplier of 3
times3 = make_multiplier_of(3)

# Multiplier of 5
times5 = make_multiplier_of(5)
```

```
# Output: 27
print(times3(9))

# Output: 15
print(times5(3))

# Output: 30
print(times5(times3(2)))
```

▼ OOPs

Class & Object

A class is a blueprint or template for creating objects. It defines properties (attributes) and behaviors (methods) that the created objects (instances) can have.

An object is an instance of a class. Each object contains data (variables) and methods to operate on that data.

```
class Car:
    def __init__(self, model, price):
        self.model = model
        self.price = price


audi = Car('R8', 100000)
print(audi.model)
print(audi.price)
```

Methods vs variables

When calling a method we need to use parentheses after method where variable can be accessed with dot operator.

```
my_date = date(2020, 12, 24)

# calling a method
```

```
weekday = my_date.isoweekday()

# accessing a variable
my_month = my_date.month

print("The day of the week:", weekday)
print("The month:", my_month)
```

Any variables attached to an object are called its *attributes*, or more specifically, *data attributes*, or sometimes *instance variables*.

Adding a constructor

```
class BankAccount:

    # The constructor
    def __init__(self, balance: float, owner: str):
        self.balance = balance
        self.owner = owner
```

The name of the constructor method is always `__init__`.The first parameter in a constructor definition is always named `self`. This refers to the object itself, and is necessary for declaring any attributes attached to the object.

Methods in classes

A method is a subprogram or function that is bound to a specific class. Usually a method only affects a single object. A method is defined within the class definition, and it can access the data attributes of the class just like any other variable.

```
class BankAccount:

    def __init__(self, account_number: str, owner: str, balance: float, annual_interes
t: float):
        self.account_number = account_number
        self.owner = owner
        self.balance = balance
        self.annual_interest = annual_interest

    # This method adds the annual interest to the balance of the account
    def add_interest(self):
        self.balance += self.balance * self.annual_interest



peters_account = BankAccount("12345-678", "Peter Python", 1500.0, 0.015)
peters_account.add_interest()
print(peters_account.balance)



1522.5
```

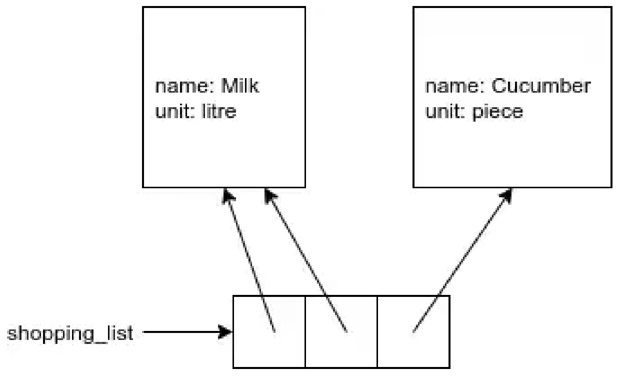
▼ Object & References

Every value in Python is an object. Any object you create based on a class you've defined yourself works exactly the same as any "regular" Python object. For example, objects can be stored in a list:

```
from datetime import date

class CompletedCourse:

    def __init__(self, course_name: str, credits: int, completion_date: date):
        self.name = course_name
        self.credits = credits
        self.completion_date = completion_date



if __name__ == "__main__":
    # Here we create some completed courses and add these to a list
    completed = []

    maths1 = CompletedCourse("Mathematics 1", 5, date(2020, 3, 11))
    prog1 = CompletedCourse("Programming 1", 6, date(2019, 12, 17))

    completed.append(maths1)
    completed.append(prog1)
    # Let's add a couple more straight to the list
    completed.append(CompletedCourse("Physics 2", 4, date(2019, 11, 10)))
    completed.append(CompletedCourse("Programming 2", 5, date(2020, 5, 1
9)))

    # Go through all the completed courses, print out their names
    # and sum up the credits received
    credits = 0
    for course in completed:
        print(course.name)
        credits += course.credits

    print("Total credits received:", credits)



Mathematics 1
Programming 1
Physics 2
Programming 2
Total credits received: 20
```

Object Reference

If we append object to list the list do not contain an object themselves. It contain *references to objects*. The exact same object can appear multiple times in a single list, and it can be referred to multiple times within the list or outside it. So changing contents of any one affects other. Let's have a look at an example:



```
class Product:
    def __init__(self, name: int, unit: str):
        self.name = name
        self.unit = unit


if __name__ == "__main__":
    shopping_list = []
    milk = Product("Milk", "litre")

    shopping_list.append(milk)
    shopping_list.append(milk)
    shopping_list.append(Product("Cucumber", "piece"))
    print(shopping_list)

    shopping_list[0].name = 'Tea'

    for lst in shopping_list:
        print(lst.name)

Tea
Tea
Cucumber
```

Objects as arguments to functions

We can pass object as argument to functions. Let's have a look at a simple example where a function receives a reference to an object of type `Student` as its argument. The function then changes the name of the student. Both the function and the main function calling it access the same object, so the change is apparent in the main function as well.

```
class Student:
    def __init__(self, name: str, student_number: str):
        self.name = name
        self.student_number = student_number

    def __str__(self):
        return f"{self.name} ({self.student_number})"# the type hint here uses th
```

```
e name of the class defined above
def change_name(student: Student):
    student.name = "Saul Student"

# create a Student object
steve = Student("Steve Student", "12345")

print(steve)
change_name(steve)
print(steve)



Steve Student (12345)
Saul Student (12345)
```

It is also possible to create objects within functions. If a function returns a reference to the newly created object, it is also accessible within the main function:

```
from random import randint, choice

class Student:
    def __init__(self, name: str, student_number: str):
        self.name = name
        self.student_number = student_number

    def __str__(self):
        return f"{self.name} ({self.student_number})"

# This function creates and returns a new Student object.
# It randomly selects values for the name and the student number.
def new_student():
    first_names = ["Mark","Mindy","Mary","Mike"]
    last_names = ["Javanese", "Rusty", "Scriptor", "Pythons"]

    # randomly determine the name
    name = choice(first_names) + " " + choice(last_names)

    # randomly determine the student number
    student_number = str(randint(10000,99999))
    # Create and return a Student object
    return Student(name, student_number)

if __name__ == "__main__":
    # Call the function five times and store the results in a list
    students = []
    for i in range(5):
        students.append(new_student())

    # Print out the results
    for student in students :
        print(student)


Mary Rusty (78218)
Mindy Rusty (80068)
Mike Pythons (70396)
Mark Javanese (83307)
Mary Pythons (45149)
```

Objects as arguments to methods

We can pass object as argument to class method. Let's have a look at an example from an amusement park:

```
class Person:
    def __init__(self, name: str, height: int):
        self.name = name
        self.height = height

class Attraction:
    def __init__(self, name: str, min_height: int):
        self.visitors = 0
        self.name = name
        self.min_height = min_height

    def admit_visitor(self, person: Person):
        if person.height >= self.min_height:
            self.visitors += 1
            print(f"{person.name} got on board")
        else:
            print(f"{person.name} was too short :(")

    def __str__(self):
        return f"{self.name} ({self.visitors} visitors)"
```

The Attraction contains a method `admit_visitor`, which takes an object of type `Person` as an argument. If the visitor is tall enough, they are admitted on board and the number of visitors is increased. The classes can be tested as follows:

```
rollercoaster = Attraction("Rollercoaster", 120)
jared = Person("Jared", 172)
alice = Person("Alice", 105)

rollercoaster.admit_visitor(jared)
rollercoaster.admit_visitor(alice)

print(rollercoaster)

Jared got on board
Alice was too short :(
Rollercoaster (1 visitors)
```

An instance of the same class as an argument to a method

We can pass instance of same class as argument to class method.

```
class Person:
    def __init__(self, name: str, year_of_birth: int):
        self.name = name
        self.year_of_birth = year_of_birth

    # NB: type hints must be enclosed in quotation marks if the parameter is of
the same type as the class itself!
```

```
    def older_than(self, another: "Person"):
        return self.year_of_birth < another.year_of_birth:
```

A list of objects as an attribute of an object

We can also pass list of objects as an attribute to another object.

```
class Player:
    def __init__(self, name: str, goals: int):
        self.name = name
        self.goals = goals

    def __str__(self):
        return f"{self.name} ({self.goals} goals)"

class Team:
    def __init__(self, name: str):
        self.name = name
        self.players = []

    def add_player(self, player: Player):
        self.players.append(player)

    def summary(self):
        goals = []
        for player in self.players:
            goals.append(player.goals)
        print("Team:", self.name)
        print("Players:", len(self.players))
        print("Goals scored by each player:", goals)


ca = Team("Campus Allstars")
ca.add_player(Player("Eric", 10))
ca.add_player(Player("Emily", 22))
ca.add_player(Player("Andy", 1))
ca.summary()

Team: Campus Allstars
```

```
Players: 3
Goals scored by each player: [10, 22, 1]
```

None: a reference to nothing

Sometimes we need to refer to something which does not exist, without causing errors. The keyword `None` represents exactly such an "empty" object.

Let's assume we want to add a method for searching for players on the team by the name of the player. If no such player is found, it might make sense to return `None`:

```
class Player:
    def __init__(self, name: str, goals: int):
        self.Object & ReferencesObject & Referencesname = name
        self.goals = goals

    def __str__(self):
        return f"{self.name} ({self.goals} goals)"

class Team:
    def __init__(self, name: str):
        self.name = name
        self.players = []

    def add_player(self, player: Player):
        self.players.append(player)

    def find_player(self, name: str):
        for player in self.players:
            if player.name == name:
                return player
        return None

ca = Team("Campus Allstars")
ca.add_player(Player("Eric", 10))
ca.add_player(Player("Amily", 22))
ca.add_player(Player("Andy", 1))

player1 = ca.find_player("Andy")
```

```
print(player1)
player2 = ca.find_player("Charlie")
print(player2)



Andy (1 goals)
None
```

▼ Private Methods

Private methods are those methods that should neither be accessed outside the class nor by any base class or derived class.

Points:-

i. To define a private method prefix the name with the **double underscore "__"**.

ii. Intended to be used only within the class where they are defined by the public method

iii. Public method and another private method can access private method.

```
class Recipient:
    def __init__(self, name: str, email: str):
        self.__name = name
        if self.__check_email(email):
            self.__email = email
        else:
            raise ValueError("The email address is not valid")

    def __check_email(self, email: str):
        # A simple check: the address must be over 5 characters long
        # and contain a dot and an @ character
        return len(email) > 5 and "." in email and "@" in email

    @property
    def email(self):
        return self.__email

    @email.setter
    def email(self, email: str):
        if self.__check_email(email):
            self.__email = email
        else:
            raise ValueError("The email address is not valid")
```

▼ Class variable

A class variable is a variable which is accessed through the class itself, not through the instances created based on the class. A class variable is declared without the `self` prefix, and usually outside any method definition as it should be accessible from anywhere within the class, or even from outside the class.

Here general_rate is class variable & we have add property named total_interest to calculate total interest.

```
class SavingsAccount:
    general_rate = 0.03

    def __init__(self, account_number: str, balance: float, interest_rate: float):
        self.__account_number = account_number
        self.__balance = balance
        self.__interest_rate = interest_rate

    def add_interest(self):
        # The total interest rate equals
        # the general rate + the interest rate of the account
        total_interest = SavingsAccount.general_rate + self.__interest_rate
        self.__balance += self.__balance * total_interest

    @property
    def balance(self):
        return self.__balance

    @property
    def total_interest(self):
        return self.__interest_rate + SavingsAccount.general_rate
```

A class variable is accessed through the name of the class, for example like this:

```
account1 = SavingsAccount("12345", 100, 0.03)
account2 = SavingsAccount("54321", 200, 0.06)

print("General interest rate:", SavingsAccount.general_rate)
print(account1.total_interest)
print(account2.total_interest)

# The general rate of interest is now 10 percent
SavingsAccount.general_rate = 0.10

print("General interest rate:", SavingsAccount.general_rate)
print(account1.total_interest)
print(account2.total_interest)



General interest rate: 0.03
0.06
0.09
General interest rate: 0.1
0.13
0.16
```

Note:-

1) **Accessing Class Variables**

We can access class variable inside init method or instance method or outside class using self & classname but it is recommended to use the class name.

```
class Student:
    # Class variable
    school_name = 'ABC School '

    # constructor
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no
        print('IN INIT METHOD')
        # access using self
        print(self.school_name)

    # Instance method
    def show(self):
        print('\nInside instance method')
        # access using self
        print(self.name, self.roll_no, self.school_name)
        # access using class name
        print(Student.school_name)

# create Object
s1 = Student('Emma', 10)
s1.show()

print('\nOutside class')
# access class variable outside class
# access using object reference
print(s1.school_name)

# access using class name
print(Student.school_name)

#Output
IN INIT METHOD
ABC School

Inside instance method
Emma 10 ABC School
ABC School

Outside class
ABC School
ABC School
```

2) **Modify Class Variables**

Generally, we assign value to a class variable inside the class declaration. However, we can change the value of the class variable either in the class or outside of class.

Note: It is recommended to change the class variable’s value using the class name.

If you modify a class variable using an instance, it doesn’t change the class variable itself. Instead, it creates an instance variable with the same name that shadows (or hides) the class variable for that instance.

```
class Student:
    # Class variable
    school_name = 'ABC School '

    # constructor
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

    # Instance method
    def show(self):
        print(self.name, self.roll_no, Student.school_name)

# create Object
s1 = Student('Emma', 10)
print('Before')
s1.show()

# Modify class variable
Student.school_name = 'XYZ School'
print('After')
s1.show()

Before
Emma 10 ABC School

After
Emma 10 XYZ School
```

Usage of class variable:-

1. Constants and Default Values:

Class variables can be used to define constants or default values shared among instances. For instance, in a configuration class, you might use class variables to store default settings for all objects.

```
class Configuration:
    default_language = "English"
    default_theme = "Light"

config1 = Configuration()
config2 = Configuration()

print(config1.default_language)  # Output: "English"
print(config2.default_theme)     # Output: "Light"
```

2. Counting Instances:

Class variables can be employed to keep track of the number of instances created from a class. This is useful for various applications, such as maintaining object counts or generating unique identifiers.

```
class Task:
    task_count = 0

    def __init__(self, description):
        self.description = description
        Task.task_count += 1

task1 = Task("Write article")
task2 = Task("Complete project")

print(Task.task_count)  # Output: 2
```

3. Shared State:

Class variables can be used to maintain shared state across instances. For example, in a bank account class, you could use a class variable to track the total balance across all accounts.

```
class BankAccount:
    total_balance = 0

    def __init__(self, balance):
        self.balance = balance
        BankAccount.total_balance += balance

acc1 = BankAccount(1000)
acc2 = BankAccount(500)

print(BankAccount.total_balance)  # Output: 1500
```

▼ Static Method

A static method in Python means it is a method that belongs to the class. Static methods are class methods and do not have access to attributes on a class or instance of the class at their own instance or to other instances or class attributes. Unlike variables which are logically grouped inside the class, they work just like any normal functions.

Static methods are used for utility or helper functions i.e logically related to class that don't need access to class data (attributes & method)

```
class BankAccount:
    # class variable
    bank_name = "Python Bank"
    interest_rate = 0.03  # 3% default interest rate

    def __init__(self, owner, balance=0):
        self.owner = owner      # instance variable
        self.balance = balance

    # Instance method: uses self
    def deposit(self, amount):
        self.balance += amount
        return f"{self.owner} deposited ${amount}. New balance: ${self.balance}"

    # Static method: uses neither self nor cls
    @staticmethod
    def is_valid_amount(amount):
        return amount > 0
```

Class Method

A class method is a method which is not attached to any single instance of the class. A class method can be called without creating any instances of the class.

Class methods are usually public, so that they can be called both from outside the class and from within the class, including from within instances of the class.

Class methods are used for factory methods or altering class-level behavior.

The `Registration` class contains a class method for checking whether a license plate is valid. The method is a class method because it is useful to be able to check if a license plate is valid even before a single Registration object is created:

```
class Registration:
    def __init__(self, owner: str, make: str, year: int, license_plate: str):
        self.__owner = owner
        self.__make = make
        self.__year = year

        # Call the license_plate.setter method
        self.license_plate = license_plate

    @property
    def license_plate(self):
        return self.__license_plate

    @license_plate.setter
    def license_plate(self, plate):
        if Registration.license_plate_valid(plate):
            self.__license_plate = plate
        else:
            raise ValueError("The license plate is not valid")

    # A class method for validating the license plate
    @classmethod
    def license_plate_valid(cls, plate: str):
        if len(plate) < 3 or "-" not in plate:
            return False

        # Check the beginning and end sections of the plate separately
        letters, numbers = plate.split("-")

        # the beginning section can have only letters
        for character in letters:
            if character.lower() not in "abcdefghijklmnopqrstuvwxyz":
                return False

        # the end section can have only numbers
        for character in numbers:
            if character not in "1234567890":
                return False

        return True



  registration = Registration("Mary Motorist", "Volvo", "1992", "abc-123")

if Registration.license_plate_valid("xyz-789"):
    print("This is a valid license plate!")



  This is a valid license plate!
```

▼ Class method vs Static Method

Binding	Bound to the class	Not bound to the class or instance
First Parameter	Takes cls (the class itself)	No special first parameter (self or cls)
Access to class/instance data	Can access and modify class-level data	Cannot access class/instance data
Common use cases	Factory methods, class-level behavior	Utility functions, helper methods
Decorator	@classmethod	@staticmethod

```
from datetime import date

class Employee:
    company_name = "Acme Corp"  # Class variable

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def from_birth_year(cls, name, birth_year):
        """Alternative constructor using class method."""
        age = date.today().year - birth_year
        return cls(name, age)

    @staticmethod
    def is_adult(age):
        """Utility method to check if age qualifies as adult."""
        return age >= 18

    def __str__(self):
        return f"{self.name}, Age: {self.age}, Company: {Employee.company_name}

# Using the regular constructor
e1 = Employee("Alice", 30)
print(e1)  # Output: Alice, Age: 30, Company: Acme Corp

# Using the class method as an alternative constructor
e2 = Employee.from_birth_year("Bob", 1990)
print(e2)  # Output: Bob, Age: 35, Company: Acme Corp

# Using the static method for a utility check
print(Employee.is_adult(20))  # Output: True
print(Employee.is_adult(16))  # Output: False
```

▼ Encapsulation

Encapsulation is the process of wrapping data members and methods into a single unit called a class.

It allows you to restrict direct access to to methods and variables from outside the class which helps protect the integrity of the data.

For example, Suppose you have an attribute that is not visible from the outside of an object and bundle it with methods that provide read or write access. In that case, you can hide specific information and control access to the object's internal state. Encapsulation offers a way for us to access the required variable without providing the program full-fledged access to all variables of a class. This mechanism is used to protect the data of an object from other objects

Why do we use encapsulation in Python ?

Encapsulation protects data from unauthorized access and modification, providing a controlled and clear interface to interact with a class.

Example of Encapsulation

For example, imagine a bank account. The balance in your account should only be changed by specific actions, such as deposits or withdrawals, not directly by anyone accessing the balance. By encapsulating the balance inside the class and providing controlled access through methods (deposit, withdraw), you ensure that the balance cannot be changed directly.

- **Private Data (Balance)**: Your balance is stored securely. Direct access from outside is not allowed, ensuring the data is protected from unauthorized changes.
- **Public Methods (Deposit and Withdraw)**: These are the only ways to modify your balance. They check if your requests (like withdrawing money) follow the rules (e.g., you have enough balance) before allowing changes.

```
class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder  # Public attribute
        self.__balance = balance  # Private attribute
    def deposit(self, amount):
        """Method to deposit money to the account."""
        if amount > 0:
            self.__balance += amount  # Modifying the private attribute
        else:
            print("Amount should be positive.")
    def withdraw(self, amount):
        """Method to withdraw money from the account."""
        if 0 < amount <= self.__balance:
            self.__balance -= amount  # Modifying the private attribute
        else:
            print("Invalid withdrawal amount.")
    def get_balance(self):
        """Method to get the balance."""
        return self.__balance
# Creating an instance of the BankAccount class
account = BankAccount("John Doe", 1000)
# Accessing public attribute directly
print(account.account_holder)
# Attempting to access private attribute directly (This will raise an AttributeErr
or)
# print(account.__balance)  # Uncommenting this line will result in an error
# Using public methods to access and modify private attribute
account.deposit(500)
account.withdraw(200)
# Accessing the private attribute via a public method
print("Balance:", account.get_balance())

#Output
John Doe
Balance: 1300
```

How can we implement encapsulation in Python classes?

We can implement encapsulation in Python by using access specifiers to restrict access to class members. In public members, attributes and methods are public by default, so they can be accessed from anywhere outside the class. Private members use double underscores as a prefix, which makes the attribute private and leads to name mangling, so accessing the members outside the class is challenging. In protected members, we use a single underscore as a prefix, which indicates that an attribute is meant for internal use only within the class and its subclass.

▼ Getter & Setter Method

To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members.

In Python, private variables are not hidden fields like in other programming languages. The getters and setters methods are often used when:

- When we want to avoid direct access to private variables
- To add validation logic for setting a value

Understanding Property in Python

Properties are a way to manage attribute access for classes. They allow you to define custom behaviour for accessing and modifying attributes.They are particularly useful when you need to perform additional actions, such as validation or computation, when getting or setting an attribute.

Imagine you're building a class to represent a bank account. You want to ensure that the account balance never goes below zero. Here's how you can use a property to enforce this constraint:

```
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance

    @property
    def balance(self):
        return self._balance

    @balance.setter
    def balance(self, new_balance):
        if new_balance < 0:
            raise ValueError("Balance cannot be negative")
        self._balance = new_balance

# Creating a bank account
account = BankAccount(1000)

# Accessing the balance property
print(account.balance)  # Output: 1000
```

```
# Modifying the balance property
account.balance = 1500
print(account.balance)  # Output: 1500
```

@property decorator

The @***property decorator*** allows you to define getter, setter, and deleter methods for attributes.

We can add getter and setter methods for accessing the private attribute using the `@property` decorator.First, we define a getter method, which returns the amount of money currently in the wallet. Then we define a setter method, which sets a new value for the money attribute while making sure the new value is not negative.

```
class Wallet:
    def __init__(self):
        self.__money = 0

    # A getter method
    @property
    def money(self):
        return self.__money

    # A setter method
    @money.setter
    def money(self, money):
        if money >= 0:
            self.__money = money
        else:
            raise ValueError("The amount must not be below zero")

wallet = Wallet()
print(wallet.money)

wallet.money = 50
print(wallet.money)
```

```
wallet.money = -30
print(wallet.money)

0
50
The amount must not be below zero
```

▼ Access Modifiers

The **Python access modifiers** are used to restrict access to class members (i.e., variables and methods) from outside the class.

Types of access modifiers:-

Public Members- Member variables of a class are public by default, which means the data members and methods can be accessed outside or inside of a class.

Private Members- The private data members are accessible only within a class and are denoted with a double underscore prefix before their name.

Protected Members- These data members are accessible within a class and their subclasses and are denoted by a single underscore prefix before the name.

Access modifier	Example	Visible to client	Visible to derived class
Public	`self.name`	yes	yes
Protected	`self._name`	no	yes
Private	`self.__name`	no	no

Public Access Modifier

Public members are the variables and methods that can be accessed from anywhere within a class, i.e., both outside and inside the class and from any part of a program. They are the default members and are specified without any explicit keyword.

```
class MyClass:
    def __init__(self, value):
        self.value = value  # Public attribute
    def show_value(self):  # Public method
        print(self.value)
```

```
# Usage
obj = MyClass(10)
obj.show_value()  # Accessing public method
print(obj.value)  # Accessing public attribute

#Output
10
10
```

Private Access Modifier

Private members in encapsulation are the variables that can be accessed only within the class. They are similar to the protected members, but what sets private members apart is they can't be accessed outside the class or by any base class. In Python, you won't find Private instance variables that can't be accessed except within a class.

We can define a private access modifier by prefixing a private member with the double underscore '__'.

```
class MyClass:
    def __init__(self, value):
        self.__value = value  # Private attribute
    def __show_value(self):  # Private method
        print(self.__value)
    def public_method(self):  # Public method
        self.__show_value()  # Accessing private method within the class

# Usage
obj = MyClass(10)
# obj.__show_value()  # This would raise an AttributeError
# print(obj.__value)  # This would raise an AttributeError
obj.public_method()  # Accesses private method through a public method

#Output
10
```

Also, private and protected members can be accessed outside a class through name mangling in Python.

```
class MyClass:
    def __init__(self):
        self._protected = "I am protected"
        self.__private = "I am private"



obj = MyClass()

# Access protected member (allowed but discouraged)
print(obj._protected)  # Output: I am protected

# Access private member using name mangling
print(obj._MyClass__private)  # Output: I am private
```

Protected Access Modifier

The members or variables of a class that can be accessed only within the class and its subclass and not outside the class are known as protected members. These members are denoted by prefixing their name with a single underscore '_'.

```
class BaseClass:
    def __init__(self, value):
        self._value = value  # Protected attribute
    def _show_value(self):  # Protected method
        print(self._value)

class SubClass(BaseClass):
    def display(self):
        self._show_value()  # Accessing protected method from the subclass

# Usage
obj = SubClass(10)
obj.display()  # Accessing protected method via public method in subclass
print(obj._value)  # Accessing protected attribute directly (not recommended)
```

```
#Output
10
10
```

▼ Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Why do we use inheritance in Python ?

- **Code Reusability:**

  Inheritance allows a new class (subclass or child class) to inherit attributes and methods from an existing class (superclass or parent class). This avoids redundant code and promotes efficient development.

- **Modularity and Organization:**

  By creating a hierarchy of classes, inheritance helps in organizing code into logical modules. Hence making the code easy to understand. Each defined class becomes a separate module that one or many other classes can inherit.

- **Extensibility:**

  New functionalities can be added to subclasses without modifying the parent class. This allows for extending the behavior of existing classes in a controlled and predictable manner.

- **Polymorphism:**

  Inheritance enables treating objects of different classes in a uniform way. This is achieved through method overriding, where a subclass can provide its own implementation of a method inherited from the parent class.

- **Real-world Modeling:**

  Inheritance facilitates modeling real-world relationships between objects. For example, a "Dog" class can inherit from an "Animal" class, reflecting the real-world relationship that a dog is an animal.

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

```
    def eat(self):
        print(f"{self.name} is eating")

class Dog(Animal):
    def bark(self):
        print("Woof!")

my_dog = Dog("Buddy")
my_dog.eat() # Output: Buddy is eating
my_dog.bark() # Output: Woof!
```

In this example, the `Dog` class inherits the `eat` method from the `Animal` class, demonstrating code reusability. It also adds its own method, `bark`, showcasing extensibility.

Parent Class

```
class Person:

def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

def printname(self):
    print(self.firstname, self.lastname)



x = Person("John", "Doe")
x.printname()


John Doe
```

Child Class

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

```
class Student(Person):

  def __init__(self, fname: str, lname:str, email: str, credits: str):
     super().__init__(fname.lname)
     self.email = email
     self.credits = credits
```

▼ Abstraction

Data abstraction is the process of hiding unnecessary and complex implementation details from users while showing only essential information and functionalities to users.

Example of data abstraction is driving a car. We know how to use breaks, clutch, and gear but we are not familiar with what's happening behind the scenes.

We can achieve data abstraction by using abstract classes and abstract classes can be created using abc (abstract base class) module and abstractmethod of abc module.

Abstraction classes in Python

An abstract class is one in which one or more abstract methods or concrete methods are defined. When we declare a method inside a class without its implementation, it is called an abstract method. An abstract class is a blueprint or template for other classes as it defines a list of methods a class must implement. A subclass can inherit an abstract class, and the object of the derived class can be used to access base class features. An abstract class can have abstract and concrete methods

Key components of an abstract class:

- One of the primary features of abstract classes is they can't be created or instantiated directly. They are the base for other classes and are not instantiated themselves.
- They lay down the template or structure to build other classes. They provide a blueprint for other classes by defining methods without implementing them.

Syntax

```
from abc import ABC
class AbstractClass(ABC):
```

Abstract Method: In Python, abstract method feature is not a default feature. To create abstract method and abstract classes we have to import the "**ABC**" and "**abstractmethod**" classes from abc (Abstract Base Class) library. The abstract method of the base class tells the child class to write an implementation of all defined abstract methods. If it’s not done, the code will throw an error.

```
from abc import ABC, abstractmethod

# Abstract class definition
class Animal(ABC):
    # Abstract method (must be implemented by subclasses)
    @abstractmethod
    def make_sound(self):
        pass
```

Concrete Method: Concrete methods are the methods defined in an abstract base class with their complete implementation. Concrete methods are required to avoid replication of code in subclasses. For example, in abstract base class there may be a method that implementation is to be same in all its subclasses, so we write the implementation of that method in abstract base class after which we do not need to write implementation of the concrete method again and again in every subclass.

```
from abc import ABC, abstractmethod

# Abstract class definition
class Animal(ABC):
    # Abstract method (must be implemented by subclasses)
    @abstractmethod
    def make_sound(self):
        pass

    # Concrete method (provides implementation)
    def sleep(self):
        print("Sleeping.")

 # Concrete class inheriting from abstract class
class Dog(Animal):
    # Implementing the abstract method
    def make_sound(self):
        print("Woof! Woof!")
```

Abstract Properties: Abstract properties work like abstract methods but are used for properties. These properties are declared with the `@property` decorator and marked as abstract using `@abstractmethod`. Subclasses must implement these properties.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @property
    @abstractmethod
    def species(self):
        pass  # Abstract property, must be implemented by subclasses

class Dog(Animal):
    @property
    def species(self):
        return "Canine"

# Instantiate the concrete subclass
dog = Dog()
print(dog.species)  # Canine
```

Implementation of Data Abstraction in Python

```
from abc import abstractmethod, ABC
```

```
# Abstract class definition
class Animal(ABC):
    # Abstract method (must be implemented by subclasses)
    @abstractmethod
    def make_sound(self):
        pass

    @property
    @abstractmethod
    def species(self):
        pass  # Abstract property, must be implemented by sub

    # Concrete method (provides implementation)
    def sleep(self):
        print(f"{self.__class__.__name__} is Sleeping")

# Concrete class inheriting from abstract class
class Dog(Animal):
    # Implementing the abstract method
    def make_sound(self):
        print("Woof! Woof!")

    @property
    def species(self):
        print('Canines')

# Concrete class inheriting from abstract class
class Cat(Animal):
    # Implementing the abstract method
    def make_sound(self):
        print("Meow!")

    @property
    def species(self):
        print('Felis Catus')

# Main execution
if __name__ == "__main__":
    # Cannot instantiate abstract class
    try:
```

```
    animal = Animal()  # This will raise TypeError
except TypeError as e:
    print(f"Error: {e}") #

# Creating instances of concrete classes
dog = Dog()
cat = Cat()

# Using abstract and concrete methods
dog.make_sound()  # Output: Woof! Woof!
dog.sleep()       # Output: Dog is sleeping.
dog.species       # Output: Canines
cat.make_sound()  # Output: Meow!
cat.sleep()       # Output: Cat is sleeping.
cat.species       # Output: Felis Catus
```

Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Example 1: Operator Overloading

the `+` operator can be used to add two numbers, concatenate two strings, or merge two lists. This is possible because the `+` operator is overloaded by the `int`, `str`, and `list` classes, respectively.

```
num1 = 1
num2 = 2
print(num1+num2) # used to perform arithmetic addition operation.

str1 = "Python"
str2 = "Programming"
print(str1+" "+str2) # used to perform concatenation.
```

Example 2: Function Polymorphism in Python

Polymorphic Built-in Functions

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Polymorphic User defined Functions

Duck typing enables functions to work with any object regardless of its type.

```
def add(a, b):
    return a + b

print(add(3, 4))          # Integer addition
print(add("Hello, ", "World!"))  # String concatenation
print(add([1, 2], [3, 4])) # List concatenation
```

Example 3: Polymorphism in Class Methods

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

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Meow")


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

    def info(self):
        print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Bark")
```

```
cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

We can pack these two different objects into a tuple and iterate through it using a common animal variable. It is possible due to polymorphism.

▼ Types of Polymorphism

▼ Compile-time Polymorphism

- Occurs when the behavior of a method is determined at compile time based on the type of the object.
- Examples include **method overloading** and operator overloading, where multiple functions or operators can share the same name but perform different tasks based on the context.
- In Python, which is dynamically typed, compile-time polymorphism is not natively supported. Instead, Python uses techniques like dynamic typing and duck typing to achieve similar flexibility.

Method Overloading

Method overloading refers to when two or more methods have similar names but different types or numbers of parameters.python does not support method overloading by default. But there are different ways to achieve method overloading in Python.

The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.

▼ Operator Overloading

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class.

Same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

```
# Python program to show use of# + operator for different purpose
s.
print(1 + 2)

# concatenate two strings
print("Geeks"+"For")

# Product two numbers
print(3 * 4)

# Repeat the String
print("Geeks"*4)
```

How to overload operators in python ?

Consider that we have two objects which are a physical representation of a class (user-defined data type) and we have to add two objects with binary '+' operator it throws an error, because compiler don't know how to add two objects. So we define a method for an operator and that process is called operator overloading. We can overload all existing operators but we can't create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method **add** is automatically invoked in which the operation for + operator is defined.

```
# Python Program illustrate how# to overload an binary + operator#
And how it actually works
class A:
    def __init__(self, a):
            self.a = a

# adding two objects
def __add__(self, o):
        return self.a + o.a
```

```
ob1 = A(1)
ob2 = A(2)
ob3 = A("Geeks")
ob4 = A("For")

print(ob1 + ob2)
print(ob3 + ob4)

#And can also be Understand as :
print(ob1.__add__(ob2))
print(ob3.__add__(ob4))



3
GeeksFor
3
GeeksFor
```

▼ Dynamic Binding

The concept of **dynamic binding** is closely related to polymorphism. In Python, dynamic binding is the process of resolving a method or attribute at runtime, instead of at compile time.

This behavior is achieved through method overriding, where a subclass provides its implementation of a method defined in its superclass.

The Python interpreter determines which is the appropriate method or attribute to invoke based on the object's type or class hierarchy at runtime. This means that the specific method or attribute to be called is determined dynamically, based on the actual type of the object.

```
class shape:
  def draw(self):
    print ("draw method")
    return

class circle(shape):
  def draw(self):
    print ("Draw a circle")
    return
```

```
class rectangle(shape):
   def draw(self):
      print ("Draw a rectangle")
      return

shapes = [circle(), rectangle()]
for shp in shapes:
   shp.draw()

#Output
Draw a circle
Draw a rectangle
```

▼ Run-time Polymorphism

- Occurs when the behavior of a method is determined at runtime based on the type of the object.
- In Python, this is achieved through **method overriding**: a child class can redefine a method from its parent class to provide its own specific implementation.

Method Overriding

Method overriding refers to defining a method in a subclass with the same name as a method in its superclass providing specific implementation.

Here’s a simple example of method overriding where a parent class Shape has a method area(), and the child classes Circle and Rectangle override it to calculate their respective areas:

```
# Parent class
class Shape:
    def area(self):
        pass  # Placeholder for area method

# Child class 1: Circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * (self.radius ** 2)

# Child class 2: Rectangle
class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

# Instantiate objects
circle = Circle(5)
rectangle = Rectangle(4, 6)

# Calling the overridden methods
print(f"Area of Circle: {circle.area()}")     # Output: Area of Circle: 7
8.5
print(f"Area of Rectangle: {rectangle.area()}") # Output: Area of Rec
tangle: 24
```

▼ Magic Methods(Dunder)

***Python Magic methods** are the methods starting and ending with double underscores '__'. They are defined by built-in classes in Python and commonly used for operator overloading.*

When you use certain operations or functions on your objects, Python automatically calls these methods.

Built in classes define many magic methods, **dir()** function can show you magic methods inherited by a class.

Example:

This code displays the magic methods inherited by **int** class.

```
# code
print(dir(int))

['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir_
```

▼ Modules, Packages & Libraries

Modules

A **module** is a single Python file (.py) that contains functions, classes, or variables. It allows you to organize related code and reuse it across programs by importing it.

Here's an example of a module named `shapes.py`:

```
# shapes.py
def draw_square():
    print("Drawing a square")
```

You can import this module into another Python file using the `import` statement:

```
# main.py
import shapes

shapes.draw_square()  # Output: Drawing a square
```

So, you may want to import only one specific function rather than the entire module. For that, you can use the following syntax:

```
from shapes import draw_square
```

Packages

A package in Python is a directory that contains multiple module files and a special `__init__.py` file. It's a way to organize related modules into a hierarchical structure.

Here's a `drawing` package containing `shapes.py`:

```
drawing/
├── __init__.py
└── shapes.py
```

You can import modules from the package:

```
from drawing import shapes

shapes.draw_square()  # Output: Drawing a square
```

Libraries

A **library** is a collection of **modules and/or packages** bundled together to provide specific functionality. Libraries can be built-in (like `math`, `os`, `datetime`) or third-party (like `numpy`, `pandas`, `requests`).

```
import math
print(math.sqrt(16))  # Output: 4.0

import requests
response = requests.get('https://api.github.com')
print(response.status_code)  # Output: 200
```

▼ Multithreading & Syncronization

Multithreading

Multithreading allows concurrent execution of two or more threads (smaller units of a process). In Python, multithreading is often used to perform I/O-bound tasks (like reading/writing files, making network requests) in parallel.

However, due to the **Global Interpreter Lock (GIL)** in CPython, only one thread executes Python bytecode at a time. So it's less effective for CPU-bound tasks but still very useful for I/O-bound operations.

Synchronization

When multiple threads access shared resources (like a variable or data structure), synchronization ensures that only one thread can modify the resource at a time. This prevents race conditions and data corruption.

Python provides synchronization primitives in the `threading` module, such as:

- `Lock`
- `RLock`
- `Semaphore`
- `Event`
- `Condition`

```
import threading
import time

# Shared resource
counter = 0

# Lock for synchronization
lock = threading.Lock()

# Function to increment the counter
def increment_counter(thread_name, increments):
    global counter
    for _ in range(increments):
        # Acquire the lock before modifying the shared resource
        with lock:  # Automatically releases lock when block is exited
            current = counter
            time.sleep(0.001)  # Simulate some work
            counter = current + 1
            print(f"{thread_name} incremented counter to {counter}")
        # Lock is released here automatically due to 'with' context

# Function to decrement the counter
def decrement_counter(thread_name, decrements):
    global counter
    for _ in range(decrements):
        # Acquire the lock before modifying the shared resource
        with lock:
            current = counter
            time.sleep(0.001)  # Simulate some work
            counter = current - 1
            print(f"{thread_name} decremented counter to {counter}")
```

```
# Create threads
t1 = threading.Thread(target=increment_counter, args=("Thread-1", 5))
t2 = threading.Thread(target=decrement_counter, args=("Thread-2", 5))

# Start threads
t1.start()
t2.start()

# Wait for both threads to finish
t1.join()
t2.join()

print(f"Final counter value: {counter}")
```

▼ DSA

https://medium.com/@devulapellisaikumar/understanding-data-structures-and-algorithms-in-python-a-beginners-guide-70d7f9c65c8b

▼ Algorithms

▼ Searching Algorithms

1) Binary Search

The idea of a binary search is to always look at the item at the very centre of the list. We then have three possible scenarios. If the item at the centre is

- the one we are looking for: we can return an indication that we found the item
- smaller than the one we are looking for: we can re-do the search in the greater half of the list
- greater than the one we are looking for: we can re-do the search in the smaller half of the list.

```
def binary_search(target: list, item: int, left : int, right : int):
    """ The function returns True if the item is contained in the target list, Fa
lse otherwise """
    # If the search area is empty, item was not found
    if left > right:
        return False
    # Calculate the centre of the search area, integer result
    centre = (left+right)//2

    # If the item is found at the centre, return
    if target[centre] == item:
        return True

    # If the item is greater, search the greater half
    if target[centre] < item:
        return binary_search(target, item, centre+1, right)
    # Else the item is smaller, search the smaller half
    else:
        return binary_search(target, item, left, centre-1)



if __name__ == "__main__":
    # Test your function
    target = [1, 2, 4, 5, 7, 8, 11, 13, 14, 18]
    print(binary_search(target, 2, 0, len(target)-1))
    print(binary_search(target, 13, 0, len(target)-1))
    print(binary_search(target, 6, 0, len(target)-1))
    print(binary_search(target, 15, 0, len(target)-1))



True
True
False
False
```

```
def binary_search(target: list, item: int, left : int, right : int):
    """ The function returns True if the item is contained in the target list, False
otherwise """
    # If the search area is empty, item was not found
    if left > right:
        return False

    # Calculate the centre of the search area, integer result
    centre = (left+right)//2
    # If the item is found at the centre, return
    if target[centre] == item:
        return True

    # If the item is greater, search the greater half
    if target[centre] < item:
        return binary_search(target, item, centre+1, right)
    # Else the item is smaller, search the smaller half
    else:
        return binary_search(target, item, left, centre-1)


if __name__ == "__main__":
    # Test your function
    target = [1, 2, 4, 5, 7, 8, 11, 13, 14, 18]
    print(binary_search(target, 2, 0, len(target)-1))
    print(binary_search(target, 13, 0, len(target)-1))
    print(binary_search(target, 6, 0, len(target)-1))
    print(binary_search(target, 15, 0, len(target)-1))
```

```
True
True
False
False
```