**Generators -**

There is a function which converts a list into iterators.

>>> nums = [1,2,3,4,5,6,7,8]

>>> it = iter(nums)

>>> print(it)

<list\_iterator object at 0x7f43c17a7100>

>>> print(it.\_\_next\_\_())

1

>>> print(it.\_\_next\_\_())

2

>>> print(next(it))

3

>>> print(next(it))

4

>>> for i in nums:

...     print(i)

...

1

2

3

4

5

6

7

8

>>>

class Topten:

    def \_\_init\_\_(self):

        self.num = 1

    def \_\_iter\_\_(self):

        return self

    def \_\_next\_\_(self):

        if self.num <= 10:

            val = self.num

            self.num += 1

            return val

        else:

            raise StopIteration

values = Topten()

print(next(values))

print(next(values))

for i in values:

    print(f"i==={i}")

def topten():

    yield 5

    yield 6

for i in topten():

    print(i)

def topten():

    n=0

    while n <= 10:

        n +=1

        yield n\*n

for i in topten():

    print(i)

Generators are used when there is a large DB with 1000 of entries and you want to fetch values one by one from that, and if do not want to load the entire data to your memory. Normal list will fetch the complete data and it will consume more memory.

Generator 🡺 yield keyword. It will not stop the execution of the function like return. It is just a function. No need to write class as we did for iterator class.

NamedTuple -🡪 <https://www.geeksforgeeks.org/namedtuple-in-python/>

import collections

student = collections.namedtuple('Student',['name','age'])

s=student('sona',10)

print(s)

In Python, NamedTuple is present inside the collections module. It provides a way to create simple, lightweight data structures similar to a class, but without the overhead of defining a full class. Like dictionaries, they contain keys that are hashed to a particular value. On the contrary, it supports both access from key-value and iteration, the functionality that dictionaries lack.

**Classes –**

class Student:

    def \_\_init\_\_(self, a, b): ## ‘init’ is a reserved method in python. In OOPS, it is called a constructor. When this method is called, it allows the class to initialise the attributes of the class. In an inherited subclass, a parent class can be referred to with the use of the ‘super()’ function. The super() function returns a temporary object of the super class that allows the access to all of its methods to its child class.

        self.a = a

        self.b = b

        print(f"a = {a}, b = {b}")

    def get\_data(self):

        lap = self.Laptop()

        return (self.a,self.b,lap.get\_lap())

    class Laptop:  ##inner class

        com = 'HP'

        def \_\_init\_\_(self):

            print("laptop configs: ", self.com)

        def get\_lap(self):

            return (self.com)

s1 = Student('xxx', '50')

print(s1.get\_data())

lap1 = s1.Laptop()

print(lap1.get\_lap())

a = xxx, b = 50

laptop configs: HP

('xxx', '50', 'HP')

laptop configs: HP

HP

**Inheritance –**

class Parent:

    def \_\_init\_\_(self, a, b):

        self.a = a

        self.b = b

        print(f"a = {a}, b = {b}")

    def get\_data(self):

        return (self.a,self.b)

class A(Parent):  ### single inheritance.

    def \_\_init\_\_(self):

        print("init -class A")

    def get\_A(self):

        print("class A")

class B(A):  ## Multi level inheritance.

    def \_\_init\_\_(self):

        print("init -class B")

    def get\_B(self):

        print("class B")

class E:

    def \_\_init\_\_(self):

        print("init -class E")

    def get\_A(self):

        print("class E")

class D:

    def \_\_init\_\_(self):

        print("init -class D")

    def get\_B(self):

        print("class D")

class C(E,D):  ## Multiple inheritance 🡺 class C will execute its own init by default. If you want to execute init of inherited class, you have to call ‘super()’ function. It will execute init of first class from left. So order will be, first init of ‘C’, then of ‘E’. then ‘D’. This is called MRO(Method Resolution Order).

‘super()’ function is used to refer to parent class or super class. It allows you to call methods defined in the superclass from the subclass, enabling you to extend and customize the functionality inherited from the parent class. It Returns a proxy object which represents the parent’s class.

    def \_\_init\_\_(self): ##pass id,name here.. like \_\_init\_\_(self, id, name):

        super().\_\_init\_\_() ## with variables eg: super().\_\_init\_\_(id, name)

        print("init -class C")

    def get\_C(self):

        print("class C")

p1 = Parent(10,20)

p1.get\_data()

a1 = A()

a1.get\_A()

b1 = B()

b1.get\_B()

c1 = C()

c1.get\_C()

https://www.geeksforgeeks.org/python-super/

a = 10, b = 20

init -class A

class A

init -class B

class B

init -class E

init -class C

class C

class A:

    clas\_var = "class\_var"   ### class variable

    def \_\_init\_\_(self,inst\_var):

        self.inst\_var = "inst\_var"   ### instance variable

        print("Inside class A ==== ", inst\_var)

    def feature1(self):

        print("class A feature1")

    @classmethod                 #### class method, for returning class related info

    def get\_classinfo(cls):      ### cls instead of self

        print("class info",cls.clas\_var)  ## cls.var

    @staticmethod                ### static method

    def get\_genericinfo():       ### not related to any variable, no need of self or cls.

        print("static method")

a = A("XXX")

a.get\_classinfo()

a.get\_genericinfo()

s

In Python, an abstract class is a class that cannot be instantiated directly and is intended to be subclassed. Abstract classes are used to define a common interface for a group of related classes while allowing subclasses to implement or extend the abstract methods as needed.

### Key Concepts

1. \*\*Abstract Methods\*\*: An abstract method is a method that is declared in an abstract class but does not provide an implementation. Subclasses inheriting from the abstract class must provide an implementation for these methods. If a subclass does not implement all abstract methods, it too becomes abstract and cannot be instantiated.

2. \*\*Instantiation\*\*: You cannot create an instance of an abstract class directly. This ensures that the abstract class serves as a blueprint for other classes rather than as a concrete implementation.

3. \*\*Purpose\*\*: Abstract classes are used to define a common interface or set of methods that subclasses must implement. They provide a way to ensure that certain methods are present in all subclasses, promoting a consistent interface across different implementations.

### Implementing Abstract Classes

In Python, abstract classes are defined using the `abc` (Abstract Base Class) module, which provides the infrastructure for defining abstract base classes. Here’s how you can define and use an abstract class in Python:

1. \*\*Import the `abc` module\*\*: Use `abc` and the `ABC` class to create an abstract class.

2. \*\*Use the `@abstractmethod` decorator\*\*: This decorator marks methods as abstract, meaning they must be implemented by any non-abstract subclass.

### Example

Here’s a simple example demonstrating how to define and use an abstract class:

```python

from abc import ABC, abstractmethod

# Define an abstract class

class Shape(ABC):

@abstractmethod

def area(self):

"""Method to calculate the area of the shape. Must be implemented by subclasses."""

pass

@abstractmethod

def perimeter(self):

"""Method to calculate the perimeter of the shape. Must be implemented by subclasses."""

pass

# Define a concrete class that inherits from the abstract class

class Rectangle(Shape):

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def area(self):

return self.width \* self.height

def perimeter(self):

return 2 \* (self.width + self.height)

# Define another concrete class that inherits from the abstract class

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

from math import pi

return pi \* self.radius \*\* 2

def perimeter(self):

from math import pi

return 2 \* pi \* self.radius

# Instantiate concrete classes

rect = Rectangle(10, 5)

print(f"Rectangle area: {rect.area()}")

print(f"Rectangle perimeter: {rect.perimeter()}")

circle = Circle(7)

print(f"Circle area: {circle.area()}")

print(f"Circle perimeter: {circle.perimeter()}")

# Attempting to instantiate the abstract class directly will raise an error

# shape = Shape() # This line would raise a TypeError: Can't instantiate abstract class Shape with abstract methods area, perimeter

```

### Explanation

1. \*\*`Shape` Class\*\*:

- This is an abstract class that defines two abstract methods: `area` and `perimeter`.

- Any concrete class that inherits from `Shape` must implement both `area` and `perimeter`.

2. \*\*`Rectangle` and `Circle` Classes\*\*:

- These are concrete implementations of the `Shape` class.

- They provide specific implementations for the `area` and `perimeter` methods.

3. \*\*Instantiation\*\*:

- You can instantiate `Rectangle` and `Circle` because they are concrete classes that provide implementations for all abstract methods.

- You cannot instantiate `Shape` directly because it contains abstract methods.

### Summary

- \*\*Abstract Class\*\*: A class that cannot be instantiated and is intended to be subclassed, providing a template with abstract methods that must be implemented by concrete subclasses.

- \*\*Abstract Methods\*\*: Methods defined in the abstract class that must be implemented by any subclass.

Using abstract classes helps in designing frameworks and libraries where you want to define a common interface for different implementations, ensuring a consistent method structure across all subclasses.

**========**

lexicographical order **is** alphabetical order. The other type is numerical ordering. Consider the following values,

1, 10, 2

Those values are in lexicographical order.

in numerical order: 10 comes after 2,

but 10 comes before 2 in "alphabetical" - aka: lexicographical - order.

**heap and stack memories in python:**

In Python, memory management is handled by the Python interpreter and the underlying memory manager (typically, `malloc` and `free` in CPython). Python abstracts away much of the details regarding memory management compared to lower-level languages like C or C++, including the management of heap and stack memory. However, conceptually, we can still discuss how these memory concepts relate to Python.

### Stack Memory:

- \*\*Description:\*\* Stack memory is a region of memory used to store local variables and function call information. It operates in a Last-In-First-Out (LIFO) manner, where the most recently pushed item is the first to be popped.

- \*\*Usage in Python:\*\*

- Python uses stack frames to manage function calls and local variables. Each time a function is called, a new stack frame is pushed onto the call stack.

- Local variables, function parameters, and return addresses are stored in the stack frame.

- When a function completes execution, its stack frame is popped from the call stack, freeing up the memory associated with its local variables.

### Heap Memory:

- \*\*Description:\*\* Heap memory is a larger pool of memory used for dynamic memory allocation. It allows for memory allocation and deallocation at runtime and is not bound to a LIFO structure.

- \*\*Usage in Python:\*\*

- Python manages heap memory internally through its memory manager, which handles dynamic memory allocation requests using mechanisms like `malloc` and `free` behind the scenes.

- Objects and data structures in Python (such as lists, dictionaries, and custom objects) are stored in the heap.

- Python's garbage collector manages the allocation and deallocation of heap memory, ensuring that unused objects are properly cleaned up.

### Key Differences and Considerations:

- \*\*Automatic Memory Management:\*\* Python handles memory management automatically, including garbage collection, so developers do not need to manually allocate or deallocate memory as in lower-level languages.

- \*\*Object References:\*\* In Python, variables store references to objects in the heap rather than the objects themselves. This means that memory is managed through reference counting and garbage collection rather than direct manipulation.

- \*\*Performance Considerations:\*\* While Python's automatic memory management simplifies programming, it can lead to performance implications due to garbage collection overhead, especially in applications with high memory churn.

### Example in Python:

Here's a simple example to illustrate how Python manages memory:

```python

def example\_function():

# Local variables stored on the stack

x = 10

y = "hello"

# Object stored on the heap

z = [1, 2, 3]

# Function call creates a new stack frame

print(x, y, z) # Accessing local variables and objects

# z goes out of scope after the function completes,

# and Python's garbage collector manages its deallocation.

example\_function()

```

In this example:

- `x` and `y` are local variables stored on the stack within the function's stack frame.

- `z` is a list object stored on the heap, and its reference is stored in the stack frame's local variable `z`.

- When `example\_function()` is called, a new stack frame is created, and local variables `x`, `y`, and `z` are allocated space on the stack. The list `[1, 2, 3]` is allocated space on the heap.

- When `example\_function()` completes, its stack frame is popped from the call stack, and Python's garbage collector will eventually deallocate the heap memory used by `[1, 2, 3]` if there are no more references to it.

In summary, while Python abstracts the explicit management of stack and heap memory compared to lower-level languages, understanding these concepts helps in writing efficient and effective Python code, especially when dealing with large datasets or performance-critical applications.

**how to write memory efficient code using python :**

Writing memory-efficient code in Python involves understanding how Python manages memory and utilizing best practices to minimize memory usage. Here are some strategies to write memory-efficient Python code:

### 1. Use Generators and Iterators

Generators and iterators are memory-efficient because they produce values on-the-fly rather than storing them all in memory at once.

- \*\*Generator Functions:\*\* Use `yield` instead of `return` to create generator functions. This allows you to generate values one at a time, which is useful for processing large datasets.

```python

def generate\_data():

for i in range(10):

yield i

for value in generate\_data():

print(value)

```

- \*\*Generator Expressions:\*\* Use generator expressions instead of list comprehensions when you only need to iterate over the result once.

```python

# List comprehension (creates a list in memory)

squares = [x\*\*2 for x in range(10)]

# Generator expression (produces values on-the-fly)

squares\_gen = (x\*\*2 for x in range(10))

```

### 2. Avoid Unnecessary Copies of Objects

Python uses references to objects, so assigning one variable to another does not create a new copy of the object unless you explicitly ask for it.

- \*\*Immutable vs. Mutable Objects:\*\* Immutable objects (like integers and strings) are safe to share because they cannot be modified. However, mutable objects (like lists and dictionaries) should be copied only when necessary.

```python

# Safe to share - both variables reference the same immutable object

a = 10

b = a

# Need to copy - modifying one list should not affect the other

list1 = [1, 2, 3]

list2 = list1.copy() # or list(list1)

```

### 3. Use Efficient Data Structures

Choose appropriate data structures and algorithms for your problem to minimize memory usage and optimize performance.

- \*\*Set vs. List:\*\* Use sets when you need unique items and do not care about the order or duplicates.

```python

# List with duplicates

items\_list = [1, 2, 3, 1, 2, 3]

# Set with unique items

items\_set = {1, 2, 3}

```

- \*\*Dictionary vs. List:\*\* Use dictionaries for key-value pairs when you need fast lookups by keys.

```python

# List of tuples vs. dictionary

items\_list = [('a', 1), ('b', 2), ('c', 3)]

items\_dict = {'a': 1, 'b': 2, 'c': 3}

```

### 4. Use Context Managers (e.g., `with` statement)

Context managers help manage resources properly by automatically releasing them when they are no longer needed, which can prevent memory leaks.

- \*\*File Handling:\*\* Use `with` statement for file handling to ensure files are closed properly after use.

```python

with open('file.txt', 'r') as f:

data = f.read()

```

### 5. Careful with Large Data Structures and Loops

Avoid unnecessary large data structures and be mindful of memory usage when iterating over large datasets.

- \*\*Iterating over Files:\*\* Use iterators and `readline()` to process large files line-by-line instead of loading the entire file into memory.

```python

with open('large\_file.txt', 'r') as f:

for line in f:

process\_line(line)

```

### 6. Profile and Optimize

Use profiling tools (`cProfile`, `memory\_profiler`, etc.) to identify memory-intensive parts of your code and optimize them accordingly.

- \*\*Identifying Memory Usage:\*\* Profile your code to identify memory-intensive operations and refactor them to be more memory-efficient.

### Example:

Here’s a basic example demonstrating memory-efficient practices:

```python

# Generator function to yield Fibonacci numbers

def fib(n): # 0,1, 1, 2, 3, 5, 8, 13 (series for n number of elements)

    a, b = 0, 1

    i = 1

    while i <= n:

        yield a

        i = i+1

        a, b = b, a+b

fib\_gen = fib(100)

fib\_sum = 0

for i in fib\_gen :

    if i > 1000:

        break

    fib\_sum = fib\_sum + i

print(fib\_sum)

def fibonacci(): #### with infinite loop

    a, b = 0, 1

    while True:

        yield a

        a, b = b, a + b

# Use generator to calculate sum of Fibonacci numbers less than 1000

fib\_gen = fibonacci()

fib\_sum = 0

for num in fib\_gen:

    if num >= 1000:

        break

    fib\_sum += num

print(f"Sum of Fibonacci numbers less than 1000: {fib\_sum}")

```

In this example:

- `fibonacci()` is a generator function that yields Fibonacci numbers one at a time, saving memory compared to generating and storing all numbers in a list.

- The `for` loop consumes Fibonacci numbers until it reaches the sum of numbers less than 1000, demonstrating efficient memory usage by using a generator.

By following these principles and practices, you can write Python code that is not only more memory-efficient but also clearer and easier to maintain.

ouRequests Module => https://realpython.com/python-requests/

-------------------

$ python -m pip install requests

import requests

response = requests.get("https://api.github.com")

response.status\_code ==> returns status\_code value.

response == > True for values between 200 and 399, False for 400+

response.raise\_for\_status() ==> from requests.exceptions import HTTPError =>

will raise HTTPError exception for status codes between 400 and 600.

response.content ==>(to get payload) -> payload in bytes

response.encoding = "utf-8" (by default, gets encoding scheme from response's headers)

response.text => payload as string

json.loads(response.text) - deserializes

response.json() ==> payload as dict

response.headers ==> dict like object. response.headers["content-type"] gives content type of response payload. - case-insensitive

https://docs.github.com/en/rest/search/search#search-repositories

==================================================

import requests

from requests.exceptions import HTTPError

URLS = ["https://api.github.com", "https://api.github.com/invalid"]

for url in URLS:

try:

response = requests.get(url)

response.raise\_for\_status()

except HTTPError as http\_err:

print(f"HTTP error occurred: {http\_err}")

except Exception as err:

print(f"Other error occurred: {err}")

else:

print("Success!")

CSV - comma separated values

it can read CSV files and convert each row into a python lsit or dict.

two main classes -

csv.reader

csv.writer

then, DictReader and DictWriter that read and write CSV files as dictionaries.

data.csv ==>

name,age,city

John,25,New York

Alice,30,Los Angeles

import csv

with open('data.csv', mode = 'r', newline='') as file:

reader = csv.reader(file)

#skip header row

next(reader)

for row in reader:

print(reader)

output ==>

['John', '25', 'New York']

['Alice', '30', 'Los Angeles']

## read with csv.DictReader >> reads a CSV file into a dict where each row is a dict, with keys as column headers.