DAY 9: MORNING ASSESSMENT

1. What is the difference between self and cls in Python classes?

Self:

self refers to the current instance of the class.

It is used in instance methods to access or modify instance variables and call other instance methods.

Class:

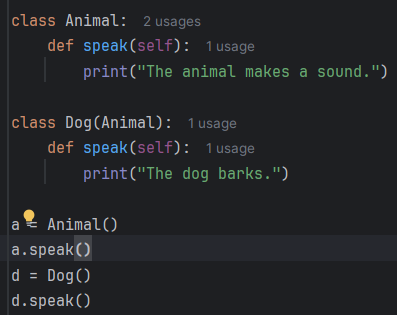
cls refers to the class itself, not an instance.

It is used in class methods, which are defined using the @classmethod decorator.

2. How does inheritance work in Python? Give an example with method overriding.

Inheritance allows a class called a *child* or *subclass* to inherit attributes and methods from another class called a *parent* or *superclass*.

Method Overriding: When a child class defines a method with the same name as a method in the parent class, the child class's method overrides the parent's version.



We can still call the parent method using super()

3. What is method overloading in Python? Is it supported natively?

Method Overloading means defining multiple methods with the same name but different parameters (number, type, or order) in the same class.

No, Python does not support method overloading.

class Example:

def greet(self):

print("Hello!")

def greet(self, name):

print(f"Hello, {name}!")

Only the last greet() method survives — the first one gets overwritten.

4. Define constructor and destructor in Python. When are they called?

A constructor is a special method called automatically when an object is created.

* In Python, the constructor is defined using the \_\_init\_\_() method.
* It is used to initialize the object’s attributes.

A destructor is a special method called when an object is about to be destroyed, typically when it goes out of scope or is deleted.

* In Python, the destructor is defined using the \_\_del\_\_() method.
* It is used for cleanup, like closing files or releasing resources.

5. What is the difference between instance method, class method, and static method?

Instance Method

* Most common type of method.
* Takes self as the first parameter.
* Can access and modify instance variables and class variables.

Class Method

* Takes cls (class) as the first parameter.
* Can access and modify class variables.
* Declared using the @classmethod decorator.

Static Method

* Does not take self or cls.
* Acts like a regular function inside a class.
* Cannot access instance or class variables directly.
* Declared using the @staticmethod decorator.

6. How do you restrict access to class attributes in Python (pseudo-private)?

Python does not have true private access modifiers.  
Instead, it uses naming conventions and name mangling to simulate private access.

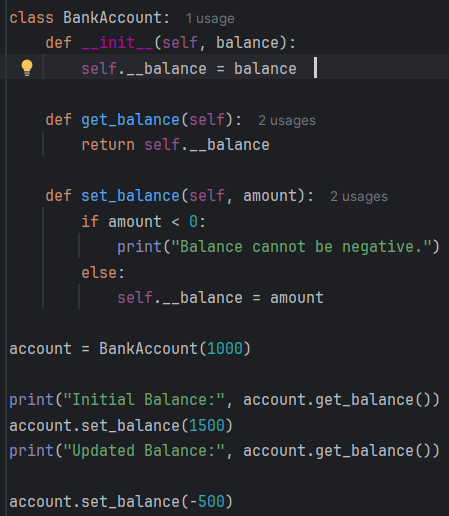
Single Underscore Prefix (\_attr): Marks an attribute as protected. Convention only - still accessible from outside the class.

Double Underscore Prefix (\_\_attr): Triggers name mangling: \_\_attr becomes \_ClassName\_\_attr.

Makes it harder to access accidentally, but still not truly private.

Use @property for Controlled Access: Makes an attribute read-only or validated. Hides the actual data behind a method-like interface.

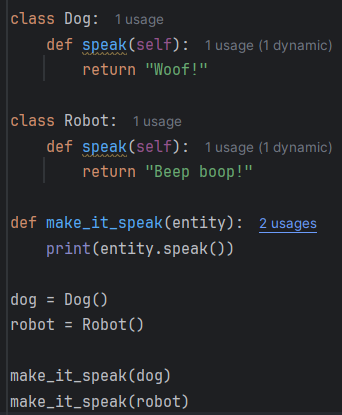
7. Write a Python class to demonstrate encapsulation with getter/setter methods.



8. What is polymorphism in Python? Show it with two unrelated classes using the same method

name.

Polymorphism means "many forms" in Python, it refers to the ability to call the same method name on different types of objects, and each object responds in its own way.



9. What is a magic method? Name a few commonly used ones and their purpose.

Magic methods special predefined methods in Python that allow you to define how your objects behave with built-in operations .

\_\_init\_\_(self): Constructor – initializes object

\_\_str\_\_(self): Defines human-readable string for print()

\_\_repr\_\_(self): Developer-friendly string representation

\_\_len\_\_(self): Defines behavior of len(obj)

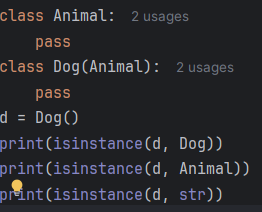
\_\_eq\_\_(self, other): Defines == operator

10. How do you use isinstance() and issubclass() functions?

isinstance():

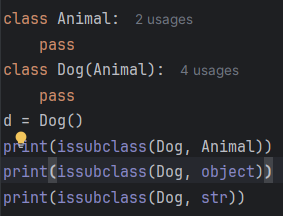
Checks if an object is an instance of a class (or a subclass).

Returns True if it matches, otherwise False.



issubclass():  
 Checks if a class is a subclass of another class (or tuple of classes).

Returns True if it is, otherwise False.



1. What is a decorator in Python and what is its typical use case?

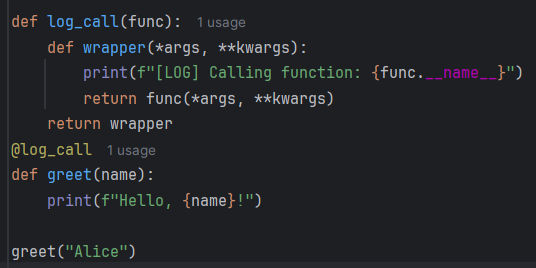
A decorator in Python is a function that takes another function as input, adds some functionality to it, and returns it without changing the original function's code.

Usecase

Logging :Automatically log when functions are called

Authentication: Check user roles/permissions before allowing access

2. Write a simple decorator that logs when a function is called.



3. Can you apply more than one decorator to a function? In what order are they applied?

Yes, you can apply more than one decorator to a function in Python!

You stack them using multiple @decorator\_name lines, one on top of the other.

Decorators are applied from bottom to top

4. What is the use of functools.wraps() in a decorator?

functools.wraps() helps to:

* Copies the original function’s:
  + \_\_name\_\_
  + \_\_doc\_\_
  + \_\_module\_\_
  + \_\_annotations\_\_
  + \_\_dict\_\_
* Keeps help() and IDE tooltips accurate
* Improves debugging and introspection

5. Convert the following decorator to one that accepts arguments (parameterized decorator).

Simple decorator without parameter:  
def log(func):

def wrapper(\*args, \*\*kwargs):

print(f"Calling {func.\_\_name\_\_}")

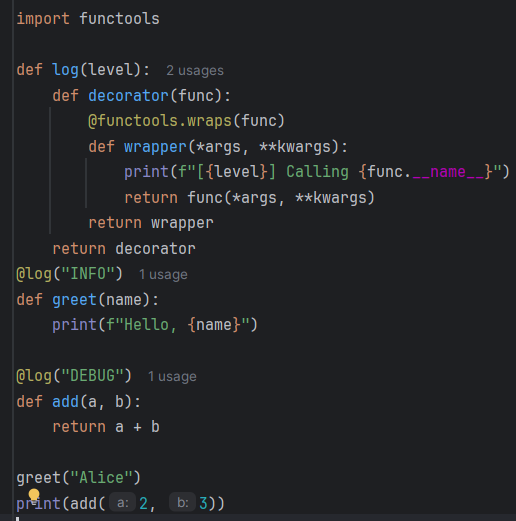
return func(\*args, \*\*kwargs)

return wrapper

@log("DEBUG")

def greet(name):

print(f"Hello, {name}")



6. How can you write a decorator to check if the user is logged in before accessing a function?



Output: 

7. How does the @property decorator work? Give an example.

The @property decorator turns a method into a “read-only” attribute.  
It allows you to access method logic like an attribute, without using parentheses ().

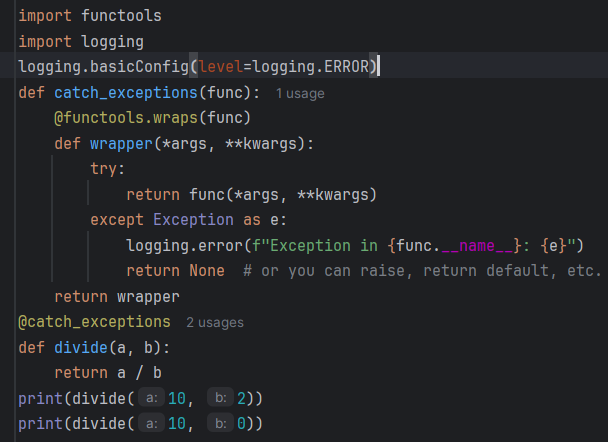


Output: 5

78.53999999999999

314.15999999999997

8. Write a decorator that catches and logs any exceptions in a function.



Output;

5.0

ERROR:root:Exception in divide: division by zero

None

9. What is the difference between function decorator and class decorator?

Function Decorator

* Applies to: Functions or methods
* Purpose: Modify or extend a function's behavior without changing its code
* Input: A function object
* Returns: A modified function

Class Decorator

* Applies to: Classes
* Purpose: Modify class behavior, inject methods or attributes, enforce constraints, register types
* Input: A class object
* Returns: A modified or wrapped class

10. Can decorators be used with class methods or static methods?

Yes, decorators can be used with both class methods and static methods in Python!

You just need to apply them after or with the appropriate method type decorators (@classmethod or @staticmethod).

1. What is a generator function? How is it different from a normal function?

A generator function is a special kind of function that yields values one at a time using the yield keyword, instead of returning them all at once.

Normal function:   
return keyword is used

A single value or data structure

Requires manual loop over results

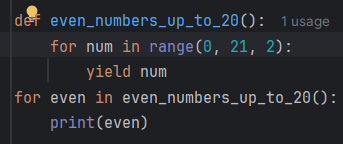
Generator:  
yield keyword is used

A generator object

Naturally works in for loops

|  |
| --- |
|  |

2. Write a generator function to yield even numbers up to 20.



3. What happens if you call next() on an exhausted generator?

If you call next() on a generator that has no more values to yield, Python will raise a:

StopIteration Exception

4. What is the use of yield? How does it help in memory efficiency?

The yield keyword is used in a generator function to pause the function execution and send a value back to the caller — while saving the function’s state for resuming later.

Unlike return, which ends a function, yield allows the function to produce a sequence of values over time.

It helps in memory efficiency as;  
it will return one item at a time

Memory usuage is low

Performace is fast

5. How do you use a generator expression? How is it different from list comprehension?

A generator expression is a concise way to create a generator — similar to list comprehensions, but using () instead of [].

gen = (expression for item in iterable if condition)

example

gen = (x\*x for x in range(5))

print(next(gen))

print(next(gen))

List Comprehension

* Syntax: [x for x in iterable]
* Returns a complete list in memory
* High memory usage for large data sets
* Slightly faster for small datasets (due to full storage in RAM)
* Best for small or fixed-size collections
* Supports indexing and slicing

Generator Expression

* Syntax: (x for x in iterable)
* Returns a generator object (yields one item at a time)
* Low memory usage (lazy evaluation)
* Slightly slower for small data (due to on-demand generation)
* Ideal for large, infinite, or streamed data
* Does not support indexing/slicing directly

6. Convert a normal function that returns a list into a generator.

Normal:  
def get\_even\_numbers(limit):

result = []

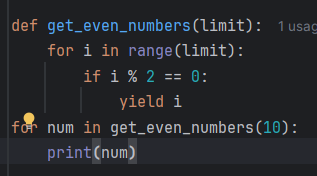
for i in range(limit):

if i % 2 == 0:

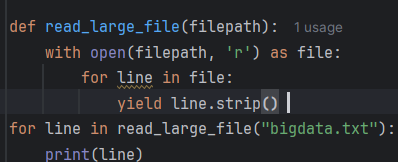
result.append(i)

return result

print(get\_even\_numbers(10))

converted code:  


7. How would you read a large file using a generator to process it line by line?



8. How does the generator maintain its state between calls?

A generator in Python maintains its state automatically between calls to next()

When a generator function yields, Python pauses its execution.

* It remembers:
  + The current position in the code
  + All local variables
  + The internal call stack
* On the next next() call, it resumes from where it left off, with all variable values intact.

9. What is the difference between return and yield inside a function?

1. return: Ends the Function

* Immediately terminates the function
* Sends back a single value
* Subsequent calls require the function to start over

2. yield: Pauses the Function (Generator)

* Pauses the function at that point
* Sends back a value
* Remembers where it left off and resumes on the next call
* Returns a generator object, not a single value

10. What is the output of list(generator\_function()) and how does it differ from a list-returning

function?

When you pass a generator function to list(), it:

* Exhausts the generator, collecting all yielded values
* Returns a list of those values

Difference from a List-Returning Function

* Evaluation:
  + list(generator\_function()): Lazy (on demand)
  + List-returning function: Eager (all at once)
* Memory Usage:
  + list(generator\_function()): Low (one item at a time)
  + List-returning function: High (entire list in memory)
* Output:
  + Both produce a list

1. What is the difference between an iterable and an iterator?

Iterable:

An object that can be looped over (used in a for loop).

Examples: list, tuple, str, set, dict, and custom classes with \_\_iter\_\_().

Does not produce values one at a time on its own.

Returns an iterator when passed to iter().

Iterator:

An object that remembers its state and produces items one at a time.

Has two methods:

\_\_iter\_\_() → returns itself

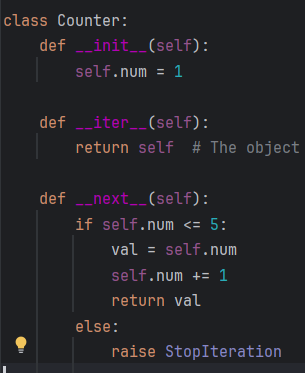
\_\_next\_\_() → returns next item or raises StopIteration

Created from an iterable using iter() or a generator.

2. How do you make a class iterable using \_\_iter\_\_() and \_\_next\_\_()?

\_\_iter\_\_(self) – returns the iterator object (usually self)

\_\_next\_\_(self) – returns the next value or raises StopIteration when done



3. Explain what happens when StopIteration is raised.

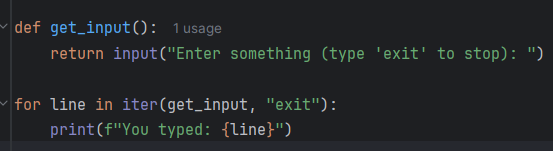
StopIteration is a built-in exception that signals the end of an iterator.

When an iterator has no more items to return, it raises StopIteration.

4. Give an example of using iter() with a sentinel value.

Python’s built-in iter() function has a two-argument form

This creates an iterator that repeatedly calls the callable until it returns the sentinel value, which ends the iteration.



5. How does a for loop work internally with iterators?

Call iter() : Converts an iterable (like list) into an iterator

Repeatedly call next() : Fetches the next item from the iterator

StopIteration : Signals the end of iteration

6. What built-in functions rely on iterators (e.g., map, zip, filter)?

built-in functions and tools are iterator-based, meaning they return iterator

map()

* Applies a function to each item of an iterable.
* Returns a map object

filter()

* Filters elements based on a condition.
* Returns a filter object

zip()

* Combines elements from multiple iterables into tuples.
* Stops at the shortest iterable.
* Returns a zip object

enumerate()

* Adds an index to each item in an iterable.
* Returns an enumerate object

reversed()

* Returns a reverse iterator over a sequence

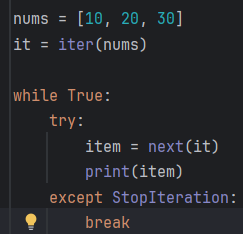
iter()

* Converts an iterable into an iterator.

next()

* Retrieves the next item from an iterator.

7. How to manually loop over an iterator using next()?



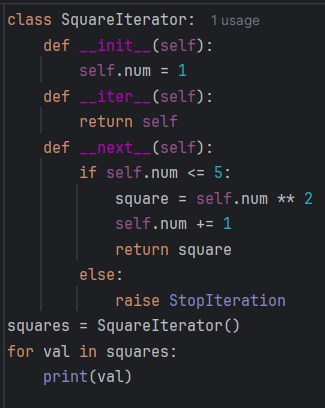
output:

10

20

30

8. Write a custom iterator that returns square of numbers from 1 to 5.



Output:  
1

4

9

16

25

9. What happens when you try to iterate over an already exhausted iterator?

Once an iterator is exhausted, meaning it has raised a StopIteration exception, any further calls to next() will always raise StopIteration

An iterator cannot be reused once exhausted.

10. What is the use of the itertools module in python with itertools.

The itertools module is a standard library in Python that provides a set of fast, memory-efficient tools for working with iterators and combinatorics.

Efficient looping with minimal memory usage

Supports combinatorics (like permutations, combinations)

Enables infinite iterators and filtering

Great for stream processing, data analysis, and functional programming