Tajamul Khan

50 Python Interview Questions

@Tajamulkhann

1. What is Python and what are its key features?

Python is a high-level, interpreted programming language known for:

- Easy-to-read syntax
- Dynamic typing
- Garbage collection
- Extensive standard library
- Support for multiple programming paradigms (OOP, functional, procedural)
- Portable and platform-independent
- Strong community support and comprehensive documentation

2. Explain the difference between lists and tuples.

Lists

- Mutable (can be modified after creation)
- Defined using square brackets: [1, 2, 3]
- Memory overhead is larger
- Better for data that changes frequently

Tuples

- Immutable (cannot be modified after creation)
- Defined using parentheses: (1, 2, 3)
- More memory efficient
- Faster iteration than lists
- Can be used as dictionary keys (lists cannot)

3. What is PEP 8?

PEP 8 is Python's official style guide that provides coding conventions for writing clean, readable Python code. It includes guidelines for:

- Indentation (4 spaces)
- Maximum line length (79 characters)
- Naming conventions (e.g., snake_case for functions and variables)
- Comments and docstring formatting
- Whitespace usage
- Import organization

4. What is the Global Interpreter Lock (GIL) in Python?

The GIL is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecode simultaneously.

Key implications:

- Only one thread can execute Python code at once.
- CPU-bound Python programs don't benefit from multithreading.
- I/O-bound programs can still benefit from threading
- The GIL is specific to CPython (the standard implementation)

5. How is memory managed in Python?

Python uses automatic memory management with:

- Private heap space that stores all objects and data structures
- Memory manager that handles allocation/deallocation
- Garbage collector that reclaims memory from objects no longer referenced
- Reference counting as primary mechanism for memory management
- Cycle detector to handle reference cycles

Memory management is hidden from the programmer, who doesn't need to allocate or free memory manually.

6. What are decorators in Python?

Decorators are functions that modify the behavior of other functions or methods without changing their source code. They use the @decorator syntax and wrap a function, enabling code reuse and separation of concerns.

Example:

```
@timer
def some_function():
    pass
```


Common use cases:

- Logging and timing functions
- Access control and authentication
- Caching function results
- Input validation and error handling

7. What are generators in Python?

Generators are functions that use the 'yield' keyword to return values one at a time, suspending & resuming their state between calls.

- Allow iteration without creating entire sequence in memory
- Are memory efficient for large data sets
- Are created using functions with 'yield' or generator expressions
- Support the iterator protocol (iter, next)
- Can be consumed only once

Example:

```
def count_up_to(n):
 i = 0
 while i < n:
 yield i
 i += 1
```

8. What is the difference between shallow copy and deep copy?

Shallow Copy

- Duplicates only the top-level container; inner elements remain shared.
- Mutating nested items reflects in both the original and the copy.
- Use via copy.copy() or .copy().

Deep Copy

- Recursively clones the container and all nested objects.
- Modifications in the copy never affect the original.
- Use via copy.deepcopy().

9. What are Python namespaces?

Namespaces in Python are mappings from names to objects. They Provide a way to avoid naming conflicts. They are implemented as dictionaries. They have different lifetimes

Python has four types of namespaces:

- Local: Inside current function
- Enclosing: Names in outer functions
- Global: At the module level
- Built-in: Contains built-in functions and exceptions

Namespaces are accessed in LEGB order:
Local → Enclosing → Global → Built-in

10. Mutable and Immutable objects

Mutable Objects

- Their state can change after creation (e.g., lists, dicts, sets, custom classes).
- Passed into functions, in-place edits modify the original.
- Maintain a variable internal state.

Immutable Objects

- Cannot be altered once created (e.g., numbers, strings, tuples, frozensets).
- “Modifications” yield entirely new objects.
- Hashable—safe to use as dictionary keys or set elements.

11. What are Python iterators?

Iterator Protocol

Defines __iter__() (returns the iterator) and __next__() (yields the next item or raises StopIteration).

Iterators

- Traverse elements one by one without indexing.
- Obtained by calling iter() on an iterable.
- Get “used up” as you loop through them.
- Highly memory-efficient for large or streaming data.
- Power for-loops, generator functions, and comprehensions.

12. __str__ vs __repr__ ?

__str__

- Invoked by str() and print()
- Returns a human-friendly description of the object
- Aimed at end-users

__repr__

- Invoked by repr() and in the REPL
- Returns an unambiguous, ideally evaluable string (e.g., ClassName(args...))
- Aimed at developers for debugging and inspection
- Serves as a fallback for __str__ if the latter isn't defined

13. Garbage collection mechanism.

Reference Counting

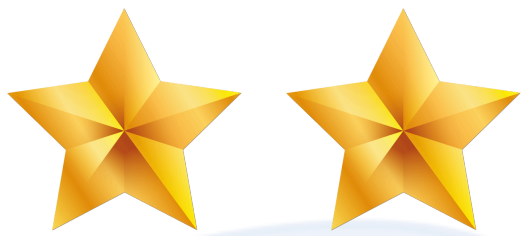
- Tracks active references to each object
- Immediately frees an object’s memory when its count drops to zero

Generational Garbage Collection

- Organizes objects into three “generations” by age
- Frequently scans young objects for cycles; survivors promote to older generations
- Efficiently reclaims cyclic references that reference counting can’t catch

Manual Control (gc module)

- gc.enable() / gc.disable() to toggle collection
- gc.collect() to force an immediate cleanup of all generations



14. Metaclasses in Python?

Metaclasses

- Classes that define how other classes are constructed and behave.
- Every class is an instance of a metaclass—type by default.
- Create a custom metaclass by subclassing type and overriding __new__/__init__.
- Use to inject attributes/methods, enforce patterns, or register classes automatically.

Common Applications

- Auto-adding methods or properties to classes
- Building ORM frameworks that register models
- Enforcing coding standards or design patterns at class creation time

15. What are context managers?

- Context managers control resource acquisition and release using the 'with' statement.
- Implement __enter__ and __exit__ methods
- Ensure proper cleanup of resources
- Handle exceptions raised within the block
- Can be created using 'contextlib.contextmanager' decorator.

Common examples:

- File operations (with open('file.txt') as f:)
- Database connections
- Locks in threading

Benefits: Cleaner code with guaranteed resource cleanup, Exception safety, Encapsulation of setup/teardown logic

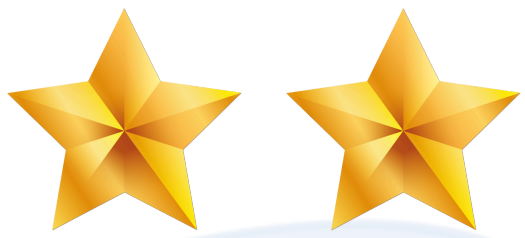
16. What is duck typing in Python?

Duck typing is a programming concept where an object's suitability is determined by its behavior (methods and properties) rather than its class or type.

"If it walks like a duck and quacks like a duck, it's a duck"

Key aspects:

- Focus on what an object can do, not what it is
- No explicit interface requirement
- Emphasizes behavior over inheritance
- Supports polymorphism without formal hierarchies
- Core to Python's dynamic and flexible nature



17. What are lambda functions?

Lambda functions are anonymous, inline functions defined using the 'lambda' **keyword**: lambda arguments: expression

Characteristics:

Single expression only (no statements)
Can take multiple arguments
Return the value of the expression
Can be assigned to variables or passed as arguments
Commonly used with map(), filter(), and sorted()

Example:

```
square = lambda x: x*x
sorted_lst = sorted(items, key=lambda x: len(x))
```

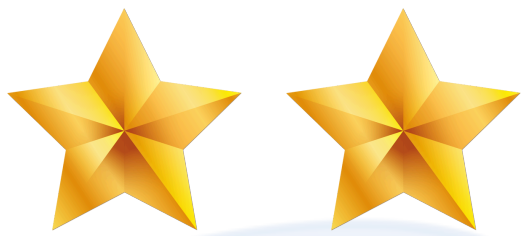
18. *args vs **kwargs.

***args**

- Collects positional arguments into a tuple
- Allows functions to accept variable number of positional arguments
- Convention is to use 'args' but any name with * works
- **Example** def func(*args): print(args)

****kwargs:**

- Collects keyword arguments into a dictionary
- Allows functions to accept variable number of keyword arguments
- Convention is to use 'kwargs' but any name with ** works
- **Example** def func(**kwargs): print(kwargs)



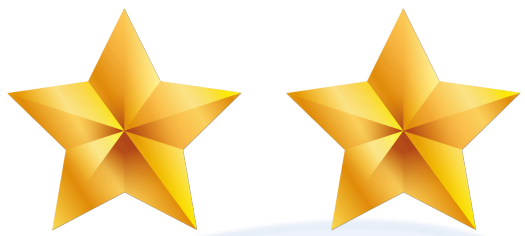
19. Pillars of OOP?

Encapsulation:
Groups data and methods inside a class and restricts direct access to some parts, protecting the object's data.

Abstraction:
Hides complex implementation details and exposes only the necessary features to the user.

Inheritance:
Allows a class (child) to inherit properties and behaviors from another class (parent), promoting code reuse.

Polymorphism:
Lets objects of different classes respond differently to the same method call, enabling flexible and interchangeable code.



20. SOLID principle in OOP design?

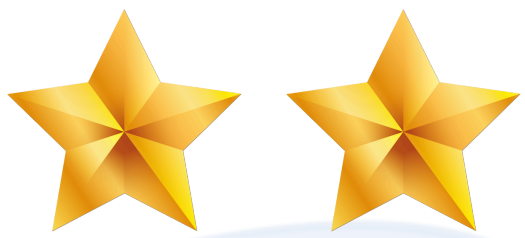
Single Responsibility Principle: A class should have only one job.

Open/Closed Principle: Code should be extendable without modifying existing code.

Liskov Substitution Principle: Subclasses should be usable in place of their parent classes without issues.

Interface Segregation Principle: Use many small, specific interfaces instead of one big one.

Dependency Inversion Principle: Depend on abstractions, not on concrete implementations.



21. What is the use of self in Python?

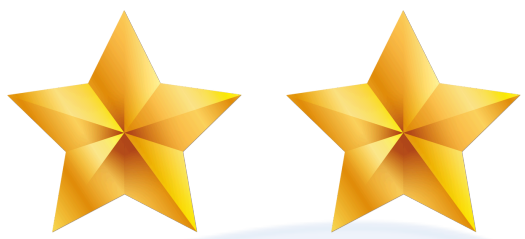
- Represents the instance of a class
- Is the first parameter of instance methods
- Allows access to attributes and methods of the instance
- Creates a way to refer to the specific instance being operated on
- Is a convention (not a keyword), but universally adopted

Example:

```
class MyClass:
    def __init__(self, value):
        self.value = value  #Instance variable
    def my_method(self):
        return self.value  #access instance variable
```

22. Python's magic methods?

"Magic methods (dunder methods) are special methods with double underscores:
__init__(): Object initialization
__str__(), __repr__(): String representation
__len__(), __getitem__(): Container behavior
__add__(), __sub__(): Operator overloading
__enter__(), __exit__(): Context manager protocol
__iter__(), __next__(): Iterator protocol
__call__(): Make objects callable
__eq__(), __lt__(): Comparison operations
They allow Python classes to integrate with built-in functions and operators, enabling natural syntax for custom objects.

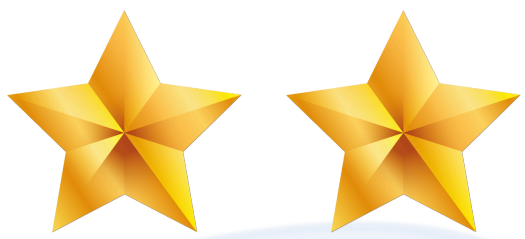


23. Purpose of __init__.py file?

_init__.py is a special Python file used to indicate that a directory is a Python package.

Key Points:

- Executed when the package or its modules are imported
- Can be empty or include initialization logic
- Can define __all__ to control from package import *
- Useful for setting up package-level variables and imports
- Required in Python 2, optional in Python 3
- Helps expose a clean API and hide internal modules



24. @staticmethod vs @classmethod

init.py is a special Python file used to indicate that a directory is a Python package.

Key Points:

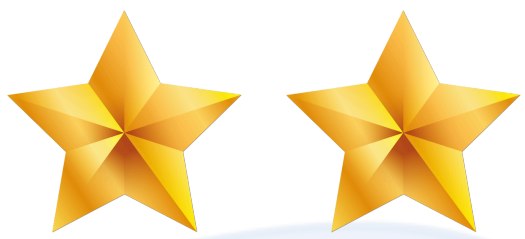
- Executed when the package or its modules are imported
- Can be empty or include initialization logic
- Can define __all__ to control from package import *
- Useful for setting up package-level variables and imports
- Required in Python 2, optional in Python 3
- Helps expose a clean API and hide internal modules

25. Instance Var vs Class Attribute

	Instance Variables	Class Attributes
Scope	*Belong to each object (instance) individually*	*Shared across all instances of the class*
Defined In	*Typically inside the __init__ method*	*Directly inside the class body (outside methods)*
Accessed With	*self.variable_name*	*ClassName.variable_name or self.variable_name*
Storage	*Stored in the instance's __dict__*	*Stored in the class's __dict__*
Overrides?	*Can override class attribute with same name on the instance*	*No override unless redefined inside the instance*
Use Case	*Store data unique to each object*	*Store data or behavior common to all objects*
Example	*self.color = "red"*	*wheels = 4*

26. Method Overridng vs Overloading

	Overriding	Overloading
Definition	*Subclass provides a new version of a method from the parent class*	*Same method name with different number/types of arguments*
Inheritance?	*Yes – used in parent-child class relationships*	*No – done within the same class*
Purpose	*Change or extend base class behavior*	*Allow one method to handle multiple input patterns*
Example	*class Dog(Animal):* *def speak():* *pass*	*def hi(self, name=None):* *pass*



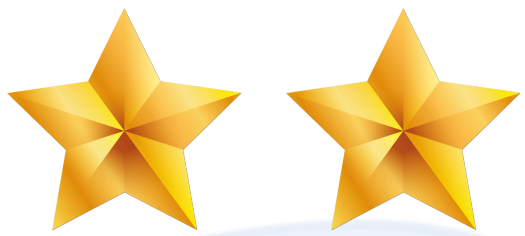
27. Python modules and packages?

Modules:

- Single Python file containing code (functions, classes, variables)
- Imported using 'import module_name'
- Provide a way to organize related code
- Accessed using dot notation: module_name.function()

Packages:

- Collection of related modules in a directory
- Must contain an __init__.py file (in Python 2, optional in Python 3)
- Enable hierarchical organization of modules
- Imported using 'import package.module'
- Can be distributed and installed using pip



28. Multithreading and its limitations.

- Implemented using the 'threading' module
- Creates multiple threads within a single process
- Shares the same memory space

Limitations:

- Global Interpreter Lock (GIL) prevents true parallel execution of Python code
- Only one thread can execute code at once
- Effective for I/O-bound tasks but not CPU-bound tasks
- Alternative solutions for CPU-bound tasks :
  - multiprocessing module (uses multiple processes)
  - concurrent.futures module - Using C extensions that release the GIL

29. Yield keyword in Python?

yield turns a function into a generator, allowing it to return values one at a time while pausing execution and preserving state.

Key Points:

- Pauses and resumes function execution
- Maintains internal state between calls
- Enables lazy evaluation
- Saves memory (ideal for large or infinite data)


```
def gen():
    yield 1
    yield 2
g = gen()
next(g)  # 1
next(g)  # 2
```

30. Python 2 vs Python 3?

	Python 2	Python 3
Print	*print "Hello" (statement)*	*print("Hello") (function)*
Division	*5 / 2 → 2 (integer division by default)*	*5 / 2 → 2.5 (true division)*
Strings	*ASCII by default*	*Unicode by default*
xrange/range	*xrange() for efficiency, range() returns list*	*range() is memory-efficient (like xrange())*
Exceptions	*except Exception, e: syntax*	*except Exception as e: required*
input()	*input() evaluates input as code*	*input() returns a string*
Libraries	*Some libraries named differently*	*Reorganized and renamed for consistency*
End of Life	*Reached EOL on Jan 1, 2020*	*Actively supported and recommended*
Feature	*Python 2*	*Python 3*

31. List Comprehensions in Python.

List comprehensions provide a concise way to create lists based on existing lists or iterables. [expression for item in iterable if condition]

Key features:

- More readable and often faster than equivalent for loops
- Can include conditions with 'if'
- Can use multiple 'for' statements for nested iterations
- Create a new list without modifying the original Similar syntax works for dict/set comprehensions

Example:

```
squares = [x*x for x in range(10) if x % 2 == 0]
```

32. Difference between is and ==

'is' operator:

Checks for identity (if two variables point to same object in memory)

Compares the memory addresses of objects

Returns True if both variables refer to exactly the same object

Used for singletons like None: if x is None

'==' operator:

Check for equality (if 2 objects have same value)

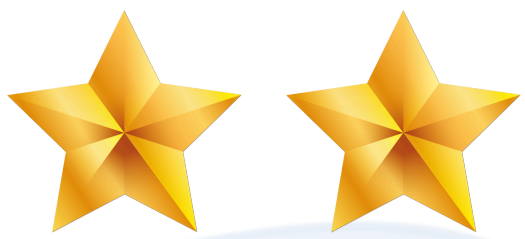
Calls the __eq__() method

Can be overridden in custom classes

Compares the content/value of objects

Example:

[] == [] returns True, but [] is [] returns False



33. Exception Handling

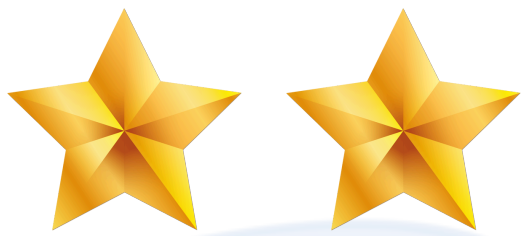
Exceptions in Python are errors that occur during program execution.

Exception handling uses try and except blocks to catch and manage errors without stopping the program.

- try: Code that may cause an error
- except: Code to handle the error
- finally: Code that runs regardless of errors

Example

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```



34. Concept of Virtual environments.

Virtual environments are isolated setups that:

- Let projects use different dependencies
- Prevent version conflicts
- Keep the global Python clean
- Enable reproducible development

Tools:
venv, virtualenv, conda

Basic usage:

```
python -m venv myenv          # Create source
myenv/bin/activate            # Activate (Unix)
myenv\Scripts\activate        # Activate (Windows)
pip install package           # Install packages
deactivate                    # Exit
```

35. Monkey Patching

It means changing code (functions, methods, classes) at runtime without modifying the source.

Used for:

- Extending third-party libraries
- Fixing bugs in external code
- Mocking in tests
- Debugging or profiling

Risks:

Can make code fragile and hard to maintain.

Example:

```
def original():
    return 1

original = lambda: 2  # Patched
```

36. import statement

Import Statement: Loads modules/packages into the current namespace.

How it works:

1. Looks in sys.path
2. Checks sys.modules cache
3. Creates module if not cached
4. Executes module code
5. Stores in sys.modules
6. Binds to local name

Variations:

- import module
- from module import name
- import module as alias
- from module import name as alias
- from module import * (not recommended)

37. Decorator in python.

A decorator is a function that takes another function and extends or modifies its behavior without changing its code. It “wraps” the original function.

Example:

```
def my_decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper
@my_decorator
def say_hello():
    print("Hello!")
say_hello()
```

38. Functional programming Concept

Functional programming in Python focuses on:

- Immutable data (no state changes)
- Pure functions (consistent outputs, no side effects)
- First-class functions (functions as values)
- Higher-order functions (take/return functions)
- Function composition (combine simple functions)

Python tools include:
map(), filter(), reduce(), lambdas, list comprehensions, decorators, and modules like functools and itertools.

39. Python Functional Utilities

map(function, iterable) Applies function to each item in iterable and returns a map object of results.
filter(function, iterable) Returns items from iterable where function(item) is True.
reduce(function, iterable) (from functools) Applies function cumulatively to items of iterable, reducing it to a single value.

Example:

```
from functools import reduce
nums = [1, 2, 3, 4, 5]
map square each number
squares = list(map(lambda x: x**2, nums))
filter keep even numbers
evens = list(filter(lambda x: x % 2 == 0, nums))
reduce sum all numbers
total = reduce(lambda x, y: x + y, nums)
```

40. Generator Expressions

resemble list comprehensions but produce generators instead of lists
(expression for item in iterable if condition)

Key points:

- Use parentheses () instead of square brackets []
- Evaluate lazily, generating items on-demand
- More memory-efficient, ideal for large data sets
- Can be iterated only once
- Perfect for streaming data or saving memory
- Can be passed directly to functions

Example

```
sum(x*x for x in range(1, 10))
```

41. append() and extend() in Python?

"**append():**

- Adds a single element to the end of a list
- The argument becomes one new element regardless of its type
- If you append a list, it becomes a nested list
- Example: lst.append(5) → [1, 2, 3, 4, 5]
- Example: lst.append([6, 7]) → [1, 2, 3, 4, 5, [6, 7]]

extend():

- Adds multiple elements to the end of a list
- Takes an iterable and adds each element individually
- No nested lists are created
- Example: lst.extend([6, 7]) → [1, 2, 3, 4, 5, 6, 7]
- Equivalent to: lst += [6, 7]

42. pickling and unpickling in Python.

Pickling:

Converts Python objects to byte streams for storage or transfer using the pickle module.

- pickle.dump(obj, file) → to file
- pickle.dumps(obj) → to bytes

Unpickling:

Restores objects from byte streams.

- pickle.load(file)
- pickle.loads(bytes)

Use cases: Save state, cache results, inter-process communication.

Note: Avoid unpickling untrusted data (security risk).

43. docstrings and its used?

Docstrings are triple-quoted string literals placed at the start of a module, class, or function to describe its purpose.

- Used for documentation, accessible via .__doc__ or help()
- Follow PEP 257 conventions
- Parsed by tools like Sphinx for auto-generating docs

Example:

```
def add(a, b):
    """Return the sum of a and b."""
    return a + b
```

44. What are Type Hints?

Type hints (PEP 484) are optional annotations that specify expected variable or function types.

- Don’t affect runtime; used for static analysis and IDE support
- Improve readability and catch type errors early (e.g., with mypy)
- Introduced in Python 3.5 via the typing module

Example:

```
from typing import List
def greet(name: str) -> str:
    return f"Hello, {name}"

x: int = 5
values: List[int] = [1, 2, 3]
```

45. range() vs enumerate()?

	range()	enumerate()
Purpose	*Generates a sequence of numbers*	*Adds index to an iterable (like list or string)*
Use Case	*When you need numeric iteration*	*When you need index and value from an iterable*
Returns	*A range object (lazy sequence of numbers)*	*An enumerate object (tuple of index, value)*
Works On	*Integers only*	*Any iterable (string, li st, tuple, etc.)*
Python Version	*Python 3+*	*Python 2.3+*

46. Tests in Python

	Description	Example
assert	*Quick checks for small scripts*	*assert add(2, 3) == 5*
unittest	*Built-in testing framework*	*import unittestclass TestAdd(unittest.TestCase): def test_add(self): self.assertEqual(add(2, 3), 5)*
pytest	*Popular 3rd-party framework (simpler syntax)*	*def test_add(): assert add(2, 3) == 5*
Method	*Description*	*Example*


```
def add(a, b):  # add.py
    return a + b
from add import add. # test_add.py
def test_add():
    assert add(2, 3) == 5
pytest test_add.py # Run with:
```

47. File Handling

Open file: open(filename, mode)
Modes: 'r' (read), 'w' (write), 'a' (append), 'b' (binary), 'x' (create)
Read: read(), readline(), readlines()
Write: write(), writelines()
Close: close() or use with for automatic closing

Example

```
with open('file.txt', 'w') as f:
    f.write("Hello, world!\n")
with open('file.txt', 'r') as f:
    content = f.read()
    print(content)
```


with ensures automatic resource cleanup, simplifies error handling, and keeps code clean.

48. Logging in python

Logging is a built-in module used to track events during program execution. It helps record information, warnings, errors, and debugging messages to files or consoles.

It supports different severity levels

- **DEBUG,**
- **INFO,**
- **WARNING,**
- **ERROR,**
- **CRITICAL**

and flexible configuration, making it essential for monitoring and troubleshooting applications.

49. Pseudocode

Pseudocode is not actual code, but rather a simplified, high-level outline of a program's logic in plain language

It is often written before actual code

Pseudocode

```
START
    Define number1 and number2
    Set number1 to 5
    Set number2 to 3
    Add number1 & number2 & store result in sum
Print sum END
```

Actual Code

```
number1 = 5
number2 = 3
sum = number1 + number2
print(sum)
```

50. Data Structures in Python

- **List** ordered, mutable sequence
- **Tuple** ordered, immutable sequence
- **Dictionary** key-value mapping
- **Set** unordered unique elements
- **Stack** usually via list (append/pop)
- **Queue** usually via collections.deque
- **Linked List** singly or doubly linked nodes
- **Tree** binary trees, BSTs, AVL trees, etc.
- **Graph** nodes & edges representing networks
- **Heap/Priority Queue** specialized tree structure for priority operations
- **Trie (Prefix Tree)** efficient string prefix retrieval
- **Disjoint Set / Union-Find** set partitioning
- **Skip List** probabilistic data structure for ordered data

Found Helpful ?

Repost

Follow for more!