What are Python decorators, and how do you use them?

Answer:

- A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure.
- It is typically used to wrap another function to extend its behavior.
- They are commonly used for logging, access control, caching, etc.

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Explain the difference between `deepcopy` and `shallow copy` in Python.

Answer:

- **Shallow Copy:** Creates a new object but doesn't create copies of nested objects. Instead, it references them. Changes to nested objects reflect in both copies.
    - Done using `copy.copy()` or object slicing.
- **Deep Copy:** Creates a new object and recursively copies all nested objects. Changes to nested objects don't affect the original.
    - Done using `copy.deepcopy()`.

```python
import copy

list1 = [[1, 2, 3], [4, 5, 6]]
shallow_copy = copy.copy(list1)
deep_copy = copy.deepcopy(list1)

shallow_copy[0][1] = 'X'
deep_copy[1][1] = 'Y'
```

What are Lambda functions in Python? Provide an example.

Answer:

- Lambda functions are anonymous functions defined using the `lambda` keyword.
- They can have any number of arguments but only one expression.
- They are commonly used for short, throwaway functions, like in `map`, `filter`, and `reduce`

```python
# Lambda to add two numbers
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5

# Using lambda with filter to get even numbers from a list
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6]
```

How do you handle exceptions in Python?

Answer:

- Exceptions are handled using `try`, `except`, `finally`, and `else` blocks.
- The `try` block lets you test a block of code for errors.
- The `except` block handles the error.
- The `finally` block executes code regardless of the outcome.
- The `else` block runs if no exception occurs.

```python
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ValueError:
    print("Invalid input. Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Result is:", result)
finally:
    print("Execution completed.")
```

Explain List Comprehensions in Python.

Answer:

- List comprehensions provide a concise way to create lists.
- It consists of brackets containing an expression followed by a `for` clause, and optionally, `if` clauses.

```
# Without List Comprehension
squares = []
for i in range(10):
    squares.append(i ** 2)
print(squares)

# With List Comprehension
squares = [i ** 2 for i in range(10)]
print(squares)
```

How do you manage dependencies and virtual environments in Python?

Answer:

- **Virtual Environments** isolate dependencies for different projects to avoid conflicts.
- **Tools:**
  - `venv`: Built into Python to create virtual environments.
  - `virtualenv`: An older alternative with more features.
  - `pipenv`: Combines package management and virtual environments.
  - `poetry`: For dependency management and packaging.

What are Generators and how are they different from Iterators?

Answer:

- **Iterators** are objects that allow traversal through all elements of a collection, using the `__iter__()` and `__next__()` methods.
- **Generators** are a simpler way to create iterators using the `yield` keyword. They generate values on the fly, hence more memory efficient.

```python
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for num in countdown(5):
    print(num)
```

What are the differences between `list`, `tuple`, and `set` in Python?

- **List**: Ordered, mutable, allows duplicates.
- **Tuple**: Ordered, immutable, allows duplicates.
- **Set**: Unordered, mutable, does not allow duplicates.

```python
lst = [1, 2, 3]
tpl = (1, 2, 3)
st = {1, 2, 3}
```

Explain the difference between `__str__` and `__repr__` in Python.

- `__str__`: Used to define a string representation of the object for the user.
- `__repr__`: Used for debugging and logging, should return a string that could recreate the object.

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

    def __str__(self):
        return f"Person's name is {self.name}"

    def __repr__(self):
        return f"Person({self.name!r})"

p = Person("Alice")
print(str(p))  # Output: Person's name is Alice
print(repr(p))  # Output: Person('Alice')
```

What is polymorphism in Python? Give an example.

Polymorphism allows methods to do different things based on the object it is acting upon. It allows using the same method or attribute on different classes

```python
class Dog:
    def speak(self):
        print("Woof")

class Cat:
    def speak(self):
        print("Meow")

def animal_sound(animal):
    animal.speak()

dog = Dog()
cat = Cat()

animal_sound(dog)  # Output: Woof
animal_sound(cat)  # Output: Meow
```

Explain the difference between `is` and `==` in Python.

- `==` checks if two objects have the same value.
- `is` checks if two objects refer to the same memory location (i.e., the same object).

```python
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(a == c)  # Output: True (same values)
print(a is c)  # Output: False (different objects)
print(a is b)  # Output: True (same object)
```

What is the difference between `assert` and `self.assertEqual` in Python's unittest module?

- `assert`: This is a general Python statement used to check a condition and raise an **AssertionError** if the condition is not true. It's commonly used for quick and simple checks.

```
python
CopyEdit
assert 1 == 1  # Passes
assert 1 == 2  # Raises AssertionError
```

- `self.assertEqual`: This is used in unit test classes (in `unittest` framework). It compares two values and raises an **AssertionError** if they are not equal.

```
python
CopyEdit
import unittest

class TestCaseExample(unittest.TestCase):
    def test_equal(self):
        self.assertEqual(1, 1)  # Passes
        self.assertEqual(1, 2)  # Fails
```

The main difference is that `assert` is a general-purpose Python statement, while `self.assertEqual` is specific to the `unittest` framework and provides better reporting.

What is the difference between `pytest` and `unittest` in Python?

- **`unittest`**: It is the built-in testing framework in Python and follows the **xUnit** style. Tests are written by creating classes that inherit from `unittest.TestCase`. It requires explicit setup and teardown methods (or `setUp` and `tearDown`).
- **`pytest`**: It is a third-party testing framework that simplifies writing and running tests. It supports features like **fixtures**, **parametrized tests**, and **assertion introspection**, making it easier and more efficient for automation testing. It doesn't require test classes and has a more concise syntax.

What is TestNG, and how is it different from pytest and unittest?

TestNG is a testing framework inspired by **JUnit**, designed for test configuration and parallel execution of tests. It is primarily used in **Java** for automation testing, but its principles and features can be applied to many frameworks, including integration with **Selenium**.

- **TestNG Features**:
  - Supports **parallel test execution**.
  - Provides annotations like `@BeforeTest`, `@AfterTest`, and `@Test`.
  - Supports **grouping** and **data-driven testing**.
  - Flexible test configuration and **test reports**.
- **Differences with `pytest` and `unittest`**:
  - **TestNG** is more feature-rich with regard to **test configuration**, **parallel execution**, and **advanced reporting**.
  - **`pytest`** is more lightweight and Pythonic, while **TestNG** requires a more extensive setup in Java.

What is the difference between a list and a tuple in Python?

- **List**:
  - Mutable (can be modified after creation).
  - Defined using square brackets (`[]`).
  - Supports operations like `append()`, `remove()`, and `pop()`.
- **Tuple**:
  - Immutable (cannot be modified after creation).
  - Defined using parentheses (`()`).
  - Cannot be changed, but you can iterate through them or access elements by index.

What are Python's built-in data structures?

- **Lists**: Ordered collection of items that are mutable. Defined using square brackets `[]`.
- **Tuples**: Ordered collection of items that are immutable. Defined using parentheses `()`.
- **Dictionaries**: Unordered collection of key-value pairs. Defined using curly braces `{}`.
- **Sets**: Unordered collection of unique items. Defined using curly braces `{}` or `set()` constructor.
- **Strings**: Immutable sequence of characters.

Each of these data structures is optimized for different use cases. Lists and tuples are useful for ordered collections, dictionaries are great for key-value lookups, and sets are useful when ensuring unique elements.

Explain the concept of `lambda` functions in Python.

A **lambda function** is an anonymous function defined with the `lambda` keyword. It is used for creating small, throwaway functions without needing to define a full function using `def`.

```
lambda arguments: expression
```

Example:

```python
# Regular function
def add(x, y):
    return x + y

# Lambda function equivalent
add_lambda = lambda x, y: x + y

print(add_lambda(2, 3))  # Output: 5
```

What is the purpose of the `self` keyword in Python?

In Python, the `self` keyword is used to represent the instance of the class. It is the first argument of every method in a class, allowing the method to access the attributes and methods of the class instance.

- **`self`** refers to the current object (or instance) of the class.
- It is not a keyword, but a convention. You can technically use any name instead of `self`, but it is strongly recommended to use it.

What are Python's built-in modules for working with dates and times?

- **`datetime`**: The most commonly used module for working with dates and times. It allows you to get the current date and time, format them, and perform arithmetic.
- **`time`**: Provides functions for working with times, including getting the current time in seconds since the epoch and setting time delays.
- **`calendar`**: Allows you to perform operations related to calendars, such as printing a calendar for a specific month or year.

What is the `with` statement in Python?

The `with` statement is used to simplify exception handling by automatically cleaning up resources after they are no longer needed. It is commonly used with file handling and other resource management tasks.

What is a generator in Python? How is it different from a list?

- A **generator** is a special type of iterator in Python that generates values on the fly and doesn't store them in memory, making it memory-efficient.
- You can create a generator using a function with the `yield` keyword instead of `return`.
- **List** stores all its elements in memory, which can be inefficient if the list is very large.

Example:

```python
# Generator
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

counter = count_up_to(5)
for number in counter:
    print(number)  # Output: 1 2 3 4 5
```

Difference:

- **List:** Stores all values in memory.
- **Generator:** Produces values one at a time, yielding each value only when required, which saves memory.

What are Python's built-in exceptions?

Python has several built-in exceptions that handle different error scenarios, such as:

- **`IndexError`**: Raised when trying to access an element of a list or tuple with an invalid index.
- **`KeyError`**: Raised when accessing a dictionary with a non-existent key.
- **`TypeError`**: Raised when an operation or function is applied to an object of an inappropriate type.
- **`ValueError`**: Raised when a function receives an argument of the correct type but an inappropriate value.
- **`FileNotFoundError`**: Raised when a file or directory is requested but cannot be found.
- **`AttributeError`**: Raised when an invalid attribute reference is made.
- **`ZeroDivisionError`**: Raised when dividing by zero.

How does exception handling work in Python?

Python uses **try-except** blocks for exception handling. Here's the general flow:

1. **`try`**: Code that may cause an exception is placed in the `try` block.
2. **`except`**: If an exception is raised in the `try` block, control is transferred to the `except` block.
3. **`else`**: If no exception occurs, the `else` block will run.
4. **`finally`**: The `finally` block is always executed, regardless of whether an exception was raised or not. It is commonly used for cleanup operations.

What is the `global` keyword in Python?

The `global` keyword allows you to modify the value of a variable defined outside the current scope, in the global scope.

- Without `global`, assignments to variables inside a function would create a local scope.
- With `global`, the assignment modifies the variable in the global scope.

```
x = 10

def modify_global():
    global x
    x = 20

modify_global()
print(x)  # Output: 20
```

What is the `nonlocal` keyword in Python?

The `nonlocal` keyword is used to work with variables in the **enclosing scope** (but not global scope). It allows you to modify a variable defined in a parent function's scope.

Example:

```
def outer():
    x = 10

    def inner():
        nonlocal x
        x = 20

    inner()
    print(x)  # Output: 20

outer()
```

Here, `nonlocal` modifies the `x` variable in the `outer` function's scope, not in the global scope.

What is the `map()` function in Python?

The `map()` function applies a given function to each item in an iterable (e.g., list) and returns an iterator of the results. It is used to perform an operation on all elements of an iterable in a concise way.

Syntax:

```python
map(function, iterable)
```

Example:

```python
numbers = [1, 2, 3, 4]
squared = map(lambda x: x**2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

Here, `lambda x: x**2` is applied to each element of `numbers`.

. What is the `filter()` function in Python?

The `filter()` function constructs an iterator from elements of an iterable for which a function returns true.

Syntax:

```python
filter(function, iterable)
```

Example:

```python
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2, 4]
```

Here, `lambda x: x % 2 == 0` filters out even numbers from the `numbers` list.

What is the difference between `range()` and `xrange()` in Python 2?

In **Python 2**, there was a difference between `range()` and `xrange()`:

- **`range()`**: Generates a list of numbers and consumes memory.
- **`xrange()`**: Generates numbers on demand (lazy evaluation), thus consuming less memory.

In **Python 3**, `range()` behaves like `xrange()` in Python 2 (i.e., it generates values lazily), and `xrange()` no longer exists.

What are Python's built-in functions for file handling?

- **`open()`**: Opens a file and returns a file object.
- **`read()`**: Reads the entire content of a file.
- **`readlines()`**: Reads all lines in a file and returns a list of lines.
- **`write()`**: Writes content to a file.
- **`close()`**: Closes an open file.
- **`with open()`**: Ensures proper handling of resources using context managers

What are Python's built-in sorting functions?

- **`sorted()`**: Returns a new sorted list from the elements of any iterable.
- **`list.sort()`**: Sorts a list in-place (modifies the original list).

Both functions support **custom sorting logic** using the `key` parameter and reverse sorting using the `reverse` parameter.

What are the differences between `@staticmethod`, `@classmethod`, and instance methods?

- **Instance Method**: A regular method that takes `self` as its first argument, referring to the instance of the class.
- **`@staticmethod`**: A method that doesn't require access to the instance (`self`) or class (`cls`). It behaves like a regular function but belongs to the class.
- **`@classmethod`**: A method that takes `cls` as its first argument, referring to the class itself, not an instance.

What is the purpose of `__init__()` in Python?

`__init__()` is a special method (constructor) in Python used for initializing a new object of a class. It is called automatically when an object of the class is created.

It is used to set the initial state of the object, by initializing its attributes.

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

car1 = Car("Toyota", "Corolla")
print(car1.make)  # Output: Toyota
print(car1.model)  # Output: Corolla
```

. What is the difference between `del` and `remove()` in Python?

- **`del`**: Deletes a variable, an item in a list, or even an entire object from memory.
- **`remove()`**: Removes the **first occurrence** of a specified value from a list

```python
# Using del
my_list = [1, 2, 3]
del my_list[0]  # Removes the item at index 0
print(my_list)  # Output: [2, 3]

# Using remove
my_list = [1, 2, 3, 1]
my_list.remove(1)  # Removes the first occurrence of 1
print(my_list)  # Output: [2, 3, 1]
```

What is the `zip()` function in Python?

The **`zip()`** function takes two or more iterables and aggregates them into tuples. It pairs elements from multiple iterables based on their positions.

```
zip(iterable1, iterable2, ...)
```

Example:

```python
names = ["Alice", "Bob", "Charlie"]
ages = [24, 30, 22]
zipped = zip(names, ages)
print(list(zipped))  # Output: [('Alice', 24), ('Bob', 30), ('Charlie', 22)]
```

How do you define a class in Python?

A **class** in Python is a blueprint for creating objects. It defines attributes and methods that the objects of the class will have.

```
class ClassName:
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

    def method(self):
        # Method logic here
```

How does Python manage memory?

Python uses an automatic memory management system, which includes:

- **Garbage Collection (GC)**: Python uses a garbage collector to automatically clean up unused objects from memory. It employs reference counting and cyclic garbage collection.
- **Reference Counting**: Every object in Python has a reference count that tracks how many references exist to that object. When the count reaches zero, the object is deleted.
- **Cyclic Garbage Collector**: Handles situations where reference cycles occur (e.g., objects referring to each other in a loop) that the reference counting mechanism alone cannot manage.

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

The **Global Interpreter Lock (GIL)** is a mutex in Python that protects access to Python objects, ensuring that only one thread can execute Python bytecode at a time. This is important because it prevents race conditions and ensures thread-safety.

However, the GIL can be a limitation for CPU-bound programs that need to take full advantage of multi-core processors, as it prevents true parallel execution of threads. For I/O-bound tasks, the GIL has less impact since threads spend time waiting on I/O operations rather than executing CPU-bound Python code.

How can you optimize memory usage in Python?

Here are some ways to optimize memory usage:

1. **Use Generators**: Generators yield items one at a time, instead of creating large lists in memory.
2. **Use `__slots__`**: For classes, you can use `__slots__` to restrict the attributes an object can have, saving memory by avoiding the overhead of the internal dictionary used to store attributes.
3. **Avoid Global Variables**: Minimize the use of global variables, which can lead to unintended references and memory consumption.
4. **Memory Profiling**: Use libraries like `memory_profiler` to profile your Python programs for memory usage and optimize accordingly.

What is a Python iterator? How do you create one?

An **iterator** is an object that allows you to traverse through a sequence of data, such as a list or a tuple, without having to access the underlying data structure directly. Python provides the built-in `iter()` and `next()` functions to work with iterators.

To create a custom iterator:

1. Define a class.
2. Implement `__iter__()` to return the iterator object.
3. Implement `__next__()` to return the next item in the sequence.

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

rev = Reverse('giraffe')
for char in rev:
    print(char)  # Output: e f f a r i g
```

What is the difference between `sort()` and `sorted()` in Python?

- **`sort()`**: A method that sorts the list **in-place**, modifying the original list.
- **`sorted()`**: A function that returns a **new sorted list**, leaving the original list unchanged.

What are the `__call__` and `__del__` special methods in Python?

- **`__call__()`**: This method allows an instance of a class to be called as a function. It is invoked when an object is followed by parentheses, similar to how functions are invoked.

  Example:

```python
class CallableClass:
    def __call__(self, x):
        print(f"Called with argument: {x}")

obj = CallableClass()
obj(10)  # Output: Called with argument: 10
```

`__del__()`: The `__del__()` method is called when an object is about to be destroyed. It is used for cleanup, similar to a destructor in other languages.

Example:

python Copy Edit

```python
class MyClass:
    def __del__(self):
        print("Object is being destroyed")

obj = MyClass()
del obj  # Output: Object is being destroyed
```

Explain the concept of Python's `enumerate()` function.

The `enumerate()` function adds a counter to an iterable and returns it as an enumerate object, which can be used in a loop. It's useful when you need the index and value from a list or other iterable.

```python
my_list = ['apple', 'banana', 'cherry']

for index, value in enumerate(my_list):
    print(index, value)
# Output:
# 0 apple
# 1 banana
# 2 cherry
```

What is the purpose of `super()` in Python?

`super()` is used to call methods from a parent class. It is commonly used in the context of inheritance to invoke methods of the base class. This is especially useful when you need to extend or override methods from the parent class.

How would you implement a singleton pattern in Python?

A **singleton pattern** ensures that a class has only one instance, and provides a global point of access to that instance. In Python, this can be achieved using various techniques, including overriding the `__new__` method.

```python
class Singleton:
    _instance = None

    def __new__(cls):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

# Test Singleton pattern
obj1 = Singleton()
obj2 = Singleton()

print(obj1 is obj2)  # Output: True, both references point to the same object
```

What are Python's `yield` and generators?

A **generator** is a special type of iterator that is defined using a function with the `yield` keyword. It allows you to generate items one at a time, on demand, without storing them in memory.

`yield` allows a function to return an item, and resume where it left off when `next()` is called.

```
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

How does Python handle multiple inheritance?

Python supports **multiple inheritance**, where a class can inherit from more than one class. In the case of a method conflict (i.e., two parent classes define a method with the same name), Python uses the **Method Resolution Order (MRO)** to determine the order in which classes are called.

You can check the MRO using the `mro()` method.

What is the difference between `filter()`, `map()`, and `reduce()` functions in Python?

1. **`filter()`**: Filters elements from an iterable based on a function that returns a boolean value. It returns an iterator of the elements for which the function returned `True`.
    - Example: `filter(lambda x: x > 2, [1, 2, 3, 4])`
2. **`map()`**: Applies a function to all items in an iterable and returns a new iterator with the results.
    - Example: `map(lambda x: x * 2, [1, 2, 3])`
3. **`reduce()`**: Cumulatively applies a function to items of an iterable to reduce them to a single value. It is found in the `functools` module.
    - Example: `reduce(lambda x, y: x + y, [1, 2, 3])`

How does Python handle function arguments?

Python supports several ways to handle function arguments:

1. **Positional arguments**: These arguments are assigned based on the position when the function is called.
2. **Keyword arguments**: These arguments are assigned based on the name of the argument when calling the function.
3. **Default arguments**: Arguments that are given default values.
4. **Variable-length arguments (`*args`, `**kwargs`)**: These allow you to pass a variable number of arguments to a function.

What is the difference between `args` and `kwargs` in Python functions?

- **`*args`**: Allows a function to accept a variable number of positional arguments (non-keyword arguments).
- **`**kwargs`**: Allows a function to accept a variable number of keyword arguments (name-value pairs).

```python
def my_func(arg1, *args, kwarg1="default", **kwargs):
    print(arg1)
    print(args)
    print(kwarg1)
    print(kwargs)

my_func(1, 2, 3, 4, kwarg1="changed", kwarg2="new")
# Output:
# 1
# (2, 3, 4)
# changed
# {'kwarg2': 'new'}
```

How does Python handle type hinting and static typing?

Python is a dynamically typed language, meaning types are determined at runtime. However, Python supports **type hinting** (introduced in Python 3.5) using the `typing` module to provide optional static typing, helping developers improve code clarity, readability, and tooling support.

```python
from typing import List, Dict

def process_data(data: List[int], info: Dict[str, str]) -> int:
    total = sum(data)
    print(info)
    return total

result = process_data([1, 2, 3], {"key": "value"})
```

What are the advantages of using Python in automation testing?

Python is widely used for automation testing due to several advantages:

1. **Readability**: Python's clean and readable syntax makes it easy for testers to write and maintain test scripts.
2. **Large ecosystem**: Python has a rich ecosystem of libraries, such as `Selenium`, `PyTest`, `unittest`, and `Robot Framework`, for automation tasks.
3. **Cross-platform compatibility**: Python works well across different operating systems, making it suitable for cross-platform testing.
4. **Integration with other tools**: Python easily integrates with other testing tools and frameworks (e.g., Jenkins, Git).
5. **Fast prototyping**: Python allows rapid development and execution of test cases.
6. **Community support**: Python has a large community that contributes to open-source automation testing frameworks.

What is the importance of using `assert` in test cases?

The `assert` keyword is used in Python to check whether a condition is true during the execution of a test. If the condition evaluates to `False`, an `AssertionError` is raised, causing the test to fail. This is useful for verifying that the expected behavior of the system matches the actual results.

How can you optimize Python code for performance?

To optimize Python code for performance, you can focus on several areas:

1. **Avoid global variables**: Local variables are faster than global ones.
2. **Use list comprehensions**: They are more efficient than using `for` loops for creating lists.
3. **Use built-in functions and libraries**: Python's standard libraries (e.g., `itertools`, `collections`) are optimized and faster than custom solutions.
4. **Leverage `map()` and `filter()`**: These functions can be faster than `for` loops for processing iterables.
5. **Profile code**: Use tools like `cProfile` or `timeit` to identify bottlenecks in your code.
6. **Avoid excessive memory usage**: Use generators instead of lists when working with large datasets to avoid loading the entire dataset into memory.

What are mock objects, and how are they used in testing?

Mock objects are simulated objects used in testing to mimic the behavior of real objects in a controlled way. They are useful for unit testing, where external systems or complex components might not be available.

Use cases:

- To isolate the code being tested by replacing dependencies with mock objects.
- To simulate network responses, database queries, etc., without needing actual resources.

Python's `unittest.mock` module provides `Mock` and `MagicMock` classes for this purpose.

Explain the concept of Test-Driven Development (TDD).

Test-Driven Development (TDD) is a software development process in which you write tests before writing the actual code. The typical TDD cycle consists of the following steps:

1. **Write a test**: Write a test case that defines a function or improvement you want to implement.
2. **Run the test**: Run the test, which should fail since the function or feature is not yet implemented.
3. **Write the code**: Implement the minimal code required to pass the test.
4. **Run the tests again**: After writing the code, run the test again to ensure it passes.
5. **Refactor**: Refactor the code to improve it while ensuring the test still passes.

What are fixtures in pytest, and how do they help in test automation?

Fixtures in `pytest` are a way to set up preconditions for tests. They allow you to define reusable setup code that can be shared across multiple test functions. Fixtures can be used to create test data, open/close files, or set up databases, among other tasks.