[Chapter 1: REVIEW OF BASIC CONCEPTS 3](#_Toc159790664)

[1. Syntax of Python code 3](#_Toc159790665)

[2. Lists, tuples, sets, dictionaries 3](#_Toc159790666)

[3. Functions 4](#_Toc159790667)

[4. If elif else Statements 4](#_Toc159790668)

[5. For / While Loops 4](#_Toc159790669)

[6. Modules 5](#_Toc159790670)

[7. Package Managers: 5](#_Toc159790671)

[8. Regular Expressions 5](#_Toc159790672)

[CLASSES AND OBJECTS 6](#_Toc159790673)

[1. Definitions 6](#_Toc159790674)

[2. Creating a class in Python 6](#_Toc159790675)

[Instance Methods and Attributes 6](#_Toc159790676)

[The property keyword 7](#_Toc159790677)

[Class Methods and Attributes 8](#_Toc159790678)

[Accessing class properties within an instance 8](#_Toc159790679)

[Creating an Object 9](#_Toc159790680)

[CONSTRUCTORS AND DESTRUCTORS 10](#_Toc159790681)

[Constructors 10](#_Toc159790682)

[Different ways an object can be constructed: 10](#_Toc159790683)

[1. Using \_\_init\_\_() method: 10](#_Toc159790684)

[\_\_new\_\_ method: 10](#_Toc159790685)

[Difference Between \_\_init\_\_ and \_\_new\_\_: 11](#_Toc159790686)

[2. Using Factory Methods: 11](#_Toc159790687)

[3. Using Object Copying 12](#_Toc159790688)

[Destructors 12](#_Toc159790689)

[Other object destruction mechanisms: 13](#_Toc159790690)

[1. Context Managers 13](#_Toc159790691)

[2. Garbage Collection 14](#_Toc159790692)

[3. Resource Management Libraries 14](#_Toc159790693)

[4. Finalization Libraries 14](#_Toc159790694)

[INHERITANCE, TYPES OF INHERITANCE, AND POLYMORPHISM 15](#_Toc159790695)

[Inheritance 15](#_Toc159790696)

[Access modifiers 15](#_Toc159790697)

[Implementation of Access Modifiers: 17](#_Toc159790698)

[Types of Inheritance 17](#_Toc159790699)

[1. Single Inheritance 17](#_Toc159790700)

[2. Multiple Inheritance 18](#_Toc159790701)

[3. Multilevel Inheritance 18](#_Toc159790702)

[Polymorphism 19](#_Toc159790703)

[Compile-time polymorphism 19](#_Toc159790704)

[Run-time polymorphism 21](#_Toc159790705)

[Operator Overloading 22](#_Toc159790706)

[How it can be done: 22](#_Toc159790707)

[Inheritance and operator overloading: 22](#_Toc159790708)

[Best Practices 23](#_Toc159790709)

[Iterators and Generators 24](#_Toc159790710)

[Iterators 24](#_Toc159790711)

[Using Built-in iterators: 24](#_Toc159790712)

[Implementing custom iterators with classes 24](#_Toc159790713)

[Iterable objects 25](#_Toc159790714)

[Creating custom iterable objects 26](#_Toc159790715)

[Why create custom iterable objects? 26](#_Toc159790716)

[Generators 27](#_Toc159790717)

[Generator functions 27](#_Toc159790718)

[Difference between generator expressions and generator functions 28](#_Toc159790719)

[Advantages of using generators 28](#_Toc159790720)

[Generator Methods within classes: 28](#_Toc159790721)

[Incorporating Generator Expressions into Other Functions and Methods: 29](#_Toc159790722)

[Built-in functions for generators 30](#_Toc159790723)

[Closures and decorators 33](#_Toc159790724)

[Exceptions 34](#_Toc159790725)

[Testing 35](#_Toc159790726)

[Frameworks 36](#_Toc159790727)

# Chapter 1: REVIEW OF BASIC CONCEPTS

## Syntax of Python code

Python syntax is simple and easy to understand. Here are some basic rules.



## Lists, tuples, sets, dictionaries

Python offers several data structures for storing collections of data:

Lists: Ordered and mutable collection of elements.

***my\_list = [1,2,3,4,5]***

Tuples: Ordered and immutable collection of elements

***My\_tuple = (1,2,3,4,5)***

Sets: Unordered collection of unique elements

***my\_set = {1,2,3,4,5}***

Dictionaries: Key-value pairs.

My\_dict = {“name”: “John”, “age”: 30, “city”: “New York”}

## Functions

Functions are block of reusable code that perform a specific task.



## If elif else Statements

Conditional statements are used to execute different blocks of code based on certain conditions.



## For / While Loops

Loops are used to iterate over a sequence of elements.

For Loop:



While Loop:



## Modules

Modules are files containing Python code that can be reused in other Python programs



## Package Managers:

Tools used to install, manage, and uninstall packages or libraries in Python.

**PyPI (Python Package Index)**: Repository of software packages for Python.

**Pip**: Package installer for Python.

**Conda**: Package, dependency, and environment management for any language—Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, FORTRAN.



## Regular Expressions

They are sequences of characters that define a search pattern



# CHAPTER 2: CLASSES AND OBJECTS

## Definitions

* Classes: Classes are blueprints for creating objects. They encapsulate data for the object and methods to operate on that data.
* Objects: objects are instances of classes. They are created based on the structure defined by the class

## Creating a class in Python

### Instance Methods and Attributes

Instance methods are functions defined inside a class and are used to manipulate the object’s state



## The property keyword

The property keyword is used to define properties for a class, allowing custom behavior when getting, setting, or deleting attributes.



## Class Methods and Attributes

These are methods that are bound to the class rather than its instances



## Accessing class properties within an instance



## Creating an Object



# CHAPTER 3: CONSTRUCTORS AND DESTRUCTORS

Constructors and destructors are special methods used during the creation and destruction of objects, respectively.

## Constructors

* Constructors are special methods that get called when an object is instantiated.
* They are used to initialize the object’s state by setting initial values to its attributes.
* In Python, the constructor method is `\_\_init\_\_()`.

### Different ways an object can be constructed:

#### Using \_\_init\_\_() method:

* This is the most common way to construct an object in Python.
* The `\_\_init\_\_()` method is called automatically when an object is created.



##### \_\_new\_\_ method:

* It is responsible for creating a new instance of a class
* Unlike \_\_init\_\_, it is a static method that is called before \_\_init\_\_.
* Mainly used in advanced scenarios where customization of the instance creation process is required.



##### Difference Between \_\_init\_\_ and \_\_new\_\_:

* **\_\_init\_\_** is responsible for initializing the newly created object by setting up its initial state, while **\_\_new\_\_** is responsible for creating the object itself.
* **\_\_init\_\_** is an instance method and is called after the object has been created, while **\_\_new\_\_** is a static method and is called before the object creation.
* **\_\_init\_\_** receives the newly created object as its first argument (**self**), while **\_\_new\_\_** receives the class itself as its first argument (**cls**).

#### Using Factory Methods:

* Factory methods are class methods that return an instance of the class.
* They provide flexibility in object creation by allowing different ways to create objects.



#### Using Object Copying

* Objects can be created by copying the state of existing objects.
* This can be achieved using methods like **`copy.copy()`** or **`copy.deepcopy()`.**



## Destructors

* Special methods used to clean up resources or perform finalization when an object is destroyed.
* In Python, the destructor method is `\_\_del\_\_()`.
* The \_\_del\_\_() method is automatically called when the reference count of the object reaches 0, indicating that there are no more references to the object and it’s about to be garbage collected.



### Other object destruction mechanisms:

In addition to the **\_\_del\_\_()** method, there are other mechanisms for managing resource cleanup and finalization in Python:

#### Context Managers

Context managers allow you to allocate and release resources precisely when you want to. They are typically used with the **with** statement. Context managers are created by implementing two special methods: **\_\_enter\_\_()** and **\_\_exit\_\_()**. Resources are automatically released when exiting the **with** block, even if an exception occurs.



#### Garbage Collection

Python's garbage collector (**gc** module) automatically reclaims memory occupied by objects that are no longer referenced. While you can't control the timing of garbage collection directly, it serves as a mechanism for cleanup in certain scenarios.



#### Resource Management Libraries

There are also libraries in Python designed for managing specific types of resources, such as file handling (**with open()** for file I/O), database connections (**contextlib.closing()** for database connections), and network sockets (**socketserver** module for network servers), which handle resource cleanup automatically.

#### Finalization Libraries

Python also has libraries like **atexit** module, which allows registering functions to be called when a program is about to exit, providing a way to perform cleanup tasks before the program terminates.



# CHAPTER 4: INHERITANCE, TYPES OF INHERITANCE, AND POLYMORPHISM

## Inheritance

* A fundamental concept in OOP where a class (subclass) inherits attributes and behaviors from another class (superclass).
* The subclass can reuse code from the superclass, extending or modifying its functionality.



### Access modifiers

In Python, access control to class members (attributes and methods) is achieved through the use of access modifiers. However, unlike some other programming languages like Java, Python does not have explicit access modifiers like **public**, **private**, or **protected**. Instead, it follows a convention-based approach for controlling access to class members.

Public Access

* By default, all attributes and methods in a Python class are considered public.
* Public members can be accessed from outside the class without any restrictions.



Protected Access

* Conventionally, attributes and methods intended for internal use within the class or its subclasses are prefixed with a single underscore **\_**.
* While there are no strict access restrictions in Python, this naming convention serves as a signal to other programmers that the member should be treated as protected.



Private Access

* Conventionally, attributes and methods that are intended to be used only within the class (not accessible from outside or even from subclasses) are prefixed with double underscore **\_\_**.
* While not strictly enforced, Python performs name mangling for these members, making them harder to access from outside the class.



### Implementation of Access Modifiers:

* **Public Access:** All attributes and methods are accessible from outside the class without any restrictions.
* **Protected Access:** Attributes and methods with a single underscore **\_** prefix can be accessed from outside the class, but it's a convention to treat them as protected.
* **Private Access:** Attributes and methods with a double underscore **\_\_** prefix undergo name mangling by Python. While technically accessible, they are intended to be used only within the class itself.

Although Python does not enforce access modifiers like some other languages, it provides a convention-based approach for controlling access to class members. However, it's important to note that these conventions are not enforced by the language itself, and accessing protected or private members from outside the class is still possible.

Top of Form

## Types of Inheritance

### Single Inheritance

* Occurs when a subclass inherits from only one superclass
* The subclass extends the superclass’ functionality.



### Multiple Inheritance

* Occurs when a subclass inherits from multiple superclasses.
* The subclass inherits attributes and behaviors from all the superclasses.



### Multilevel Inheritance

* Occurs when a subclass inherits from another subclass.
* It forms a chain of inheritance where each subclass inherits from its immediate superclass.



## Polymorphism

* It is the ability of different classes to be treated as objects of a common superclass.
* It allows methods to be called on objects of different classes, and the appropriate method is executed based on the object’s actual class.
* The different types of polymorphism in OOP are; compile-time polymorphism (static binding), run-time polymorphism (dynamic binding)

### Compile-time polymorphism

* Occurs when the method to be invoked is determined at compile time.
* It is achieved through function overloading and operator overloading.

Function overloading



Operator overloading:



### Run-time polymorphism

* Occurs when the method invoked is determined at runtime.
* Achieved through method overriding.



# CHAPTER 5: Operator Overloading

A type of compile-time polymorphism that allows developers to define custom behavior for built-in operators like **+**, **-**, **\***, **/**, **==**, **!=**, etc., when applied to objects of user-defined classes.

It enables developers to extend the functionality of operators beyond their predefined behavior for built-in types.

## How it can be done:

* Operator overloading in Python is achieved by defining special methods with double underscores (**\_\_**) also known as dunder methods.
* Each operator has a corresponding special method. For example, **\_\_add\_\_()** for **+**, **\_\_sub\_\_()** for **-**, **\_\_mul\_\_()** for **\***, and so on.



## Inheritance and operator overloading:

* Operator overloading is not automatically passed down through inheritance.
* If you want subclasses to support the same behavior for operators as their superclass, you need to explicitly implement the corresponding special methods in the subclass.

## Best Practices

* **Follow Pythonic Naming Conventions:** Stick to the standard naming conventions for special methods in Python.
* **Document Custom Behavior:** Provide clear documentation for the behavior of overloaded operators to make your code more understandable to others.
* **Avoid Unexpected Behavior:** Ensure that the overloaded operators behave consistently with their standard meanings to avoid confusion for other developers.
* **Use sparingly and judiciously:** Operator overloading can make code concise and expressive, but it should be used judiciously to avoid making code overly complex or difficult to understand.

# CHAPTER 6: Iterators and Generators

In this chapter, we delve into the fundamental concepts of iterators and generators in Python. We explore their significance in Python programming and provide an overview of iterable objects, laying the groundwork for understanding iterators and generators.

## Iterators

* In Python, an iterator is an object that allows sequential traversal through a collection of elements, such as lists, tuples, dictionaries, or custom-defined objects.
* Iterators implement the iterator protocol, which consists of two methods: **\_\_iter\_\_()** and **\_\_next\_\_()**. These methods allow an iterator to be used in a **for** loop or to retrieve elements one at a time using the **next()** function.
* An object is considered an iterator if it implements the iterator protocol, which requires the presence of two special methods: **\_\_iter\_\_()** and **\_\_next\_\_()**.
* The **\_\_iter\_\_()** method returns the iterator object itself and is required to allow iteration over the object.
* The **\_\_next\_\_()** method returns the next element in the iterator and raises a **StopIteration** exception when there are no more elements to be returned.

### Using Built-in iterators:

* Many built-in data types in Python, such as lists, tuples, dictionaries, and strings, are iterable and can be used directly in **for** loops or with the **iter()** and **next()** functions.



### Implementing custom iterators with classes

* Custom iterators can be implemented by defining a class that contains the **\_\_iter\_\_()** and **\_\_next\_\_()** methods.
* The **\_\_iter\_\_()** method returns the iterator object itself.
* The **\_\_next\_\_()** method computes and returns the next element in the iteration sequence.



### Iterable objects

In Python, an iterable object is any object that can be iterated over, meaning it can be used in a **for** loop or with functions like **iter()** and **next()**. Iterable objects implement the iterator protocol, which requires the presence of the **\_\_iter\_\_()** method. This method returns an iterator object that allows sequential access to the elements of the iterable. Some built-in iterable objects include lists, tuples, strings and dictionaries.

#### Creating custom iterable objects

Sometimes, it's necessary to create custom iterable objects to represent collections of data or to encapsulate complex behavior. This can be done by defining a class that implements the iterator protocol, specifically the **\_\_iter\_\_()** method.

The \_\_iter\_\_() method must return an iterator object.



### Why create custom iterable objects?

1. **Encapsulation of Complex Behavior:**
   * Custom iterable objects can encapsulate complex behavior or data structures, making code more modular and easier to understand.
2. **Adaptation to Specific Requirements:**
   * Sometimes, built-in iterable objects may not meet specific requirements, and custom iterable objects allow developers to tailor the behavior according to their needs.
3. **Integration with Existing Codebase:**
   * Custom iterable objects can be designed to seamlessly integrate with existing codebases, providing a consistent interface for working with collections of data.
4. **Abstraction and Reusability:**
   * By creating custom iterable objects, developers can abstract away implementation details and promote code reusability across different parts of the codebase.

## Generators

A generator is a special type of iterable that allows you to iterate over a sequence of values lazily, generating values on the fly rather than storing them all in memory at once. Generators are defined using either generator functions or generator expressions.

### Generator functions

* **Generator functions** are defined using the **def** keyword and contain one or more **yield** statements.
* When called, a generator function returns a generator object, which can then be iterated over to produce values.
* The **yield** statement suspends the function's execution, saving its state, and returns a value to the caller. The function can then be resumed later, picking up from where it left off.



### Generator expressions

* **Generator expressions** are similar to list comprehensions but produce values lazily, one at a time, rather than creating a full list in memory.
* Generator expressions are defined using parentheses **()** instead of square brackets **[]**.



### Difference between generator expressions and generator functions

* **Syntax:** Generator expressions use a concise syntax resembling list comprehensions, while generator functions are defined using the **def** keyword and contain one or more **yield** statements.
* **Use Cases:**
  + Generator expressions are suitable for simple transformations or filtering of data.
  + Generator functions offer more flexibility and are better suited for complex logic or operations that require multiple **yield** statements.

### Advantages of using generators

1. **Memory Efficiency:** Generators produce values lazily, only when needed, so they consume less memory compared to creating an entire list of values upfront.
2. **Performance Optimization:** Generators can be more efficient when dealing with large datasets or infinite sequences since they generate values on the fly rather than precomputing them.
3. **Simplified Syntax:** Generator expressions provide a concise and readable way to create generators without the need for defining a separate function.

### Generator Methods within classes:

Generator methods within classes can be defined just like regular methods but include the **yield** statement to yield values one at a time. These methods can then be called to generate values lazily within the context of an object instance.



### Incorporating Generator Expressions into Other Functions and Methods:

Generator expressions can be incorporated into other functions and methods by passing them as arguments or using them directly within the function or method definition.



### Built-in functions for generators

Built-in functions like **next()**, **iter()**, and **yield from** play crucial roles in working with generators in Python.

1. **. next()**

* The **next()** function is used to retrieve the next value from a generator.
* It advances the generator's internal state and returns the value produced by the generator.



1. **iter()**

* The **iter()** function is used to create an iterator from an iterable object.
* It allows generators to be used in contexts where an iterator is expected.



1. **yield from**

* The **yield from** statement delegates to another generator or iterable, allowing it to yield values directly.
* It simplifies the process of delegating iteration to sub-generators or iterables.



# Closures and decorators

# Exceptions

# Testing

# Frameworks