***PYTHON***

SHAIK NAGUR SHARIF

Document Date: October 12 2023, Number #123456, Volume: 00

**Author:** Shaik Nagur Sharif

**Address Information**

00 Flower Street City, AA 00000, PK

**Phone:** 000.123.45678 | **Fax:** 123.456.7890 | **Web:** https://UsedtoTech.com

Contents

[Objects 3](#_Toc148200571)

[Level 2 Heading Goes Here 11](#_Toc148200572)

[Level 3 heading goes here 11](#_Toc148200573)

[Again Level 2 Heading 11](#_Toc148200574)

[Level 2 Heading Again 12](#_Toc148200575)

[Level 3 heading goes here 12](#_Toc148200576)

[Level 4 heading goes here 12](#_Toc148200577)

[Level 4 heading again 12](#_Toc148200578)

[Level 2 Heading AGain 13](#_Toc148200579)

[Level 3 heading goes here 13](#_Toc148200580)

[Level 4 heading goes here 13](#_Toc148200581)

## Objects

1. **Objects**:

Objects are structures that contain both data and procedures. For example, a student is an object which has a name and age.

1. **Class**: A class is a template that explains the details of an object.
2. **Inheritance:**

Inheritance is a technique to reuse existing code again and again. The class that is inherited is called a base class and a class that it inherits is called a derived class.

1. **Polymorphism:**

Polymorphism means many, which is requesting the same operation to perform differently.

1. **Abstractions:**

It refers to displaying only essential features of the application and covering the details,

1. **Encapsulation:** It means wrapping the data and functions together into a class.

**1. What is an Object?**

In OOP, an object is a self-contained structure that contains both data and methods to manipulate that data. It's like a virtual representation of a real-world entity.

**Note Points**:

* Objects are instances of classes.
* They can represent real-world entities.
* An object encapsulates state (data) and behavior (methods).

**Objects in the "Educational Website" Example**

Taking our educational website as an instance, many real-world entities can be represented as objects. For example:

* A **Student** is an object.
* A **Course** is an object.
* A **Lesson** within a course is an object.

**2. Object Identity**

Every object typically has a unique identifier, which distinguishes it from every other object.

Example:

Two courses might have the same name and description, but they are distinct courses with different students, instructors, and perhaps even content. Thus, even if their attributes are the same, their identity is different.

**Note Points**:

* In many programming languages, this identity corresponds to the memory location where the object is stored.
* Identity ensures that even if two objects have the same content, they can still be distinguished.

**3. Attributes of an Object (State)**

Objects have attributes, which represent the state of the object. They store the data about the object.

**a. For the Student object**:

* **student\_id**: A unique identifier.
* **name**: Name of the student.
* **email**: Email address.
* **enrolled\_courses**: List of courses the student has enrolled in.

**b. For the Course object**:

* **course\_id**: Unique identifier for the course.
* **title**: The name of the course.
* **description**: A brief overview of the course.

**Note Points**:

* Attributes provide characteristics to an object.
* They are also referred to as properties, fields, or data members.

**4. Behavior of an Object (Methods)**

Objects have methods, which represent the actions or behavior the object can perform.

**a. For the Student object**:

* **enroll(course)**: Enroll the student in a course.
* **drop(course)**: Drop a course.
* **view\_progress(course)**: View progress in a particular course.

**b. For the Course object**:

* **add\_student(student)**: Add a student to the course.
* **remove\_student(student)**: Remove a student.
* **update\_description(new\_description)**: Update the course description.

**Note Points**:

* Methods operate on an object's attributes.
* They define what an object can do, or what can be done to it.

**5. Object's State vs. Behavior**

The state of an object is encapsulated in its attributes, while behavior is encapsulated in its methods.

Example:

For our **Course** object:

**State**:

* **course\_id**: **CS101**
* **title**: **Introduction to Computer Science**

**Behavior**:

* Can enroll a student.
* Can modify course content.
* Can provide a list of all enrolled students.

**Note Points**:

* The state defines "what it is", and Behavior defines "what it does".

**6. Importance of Objects**

In OOP, objects are crucial because they allow you to model real-world scenarios in a structured and scalable manner. They're the building blocks of OOP-based applications.

**Note Points**:

* Objects help in organizing large codebases by representing modular and reusable components.
* Properly designed objects can make a system easier to maintain, scale, and understand.

**7. Objects and Data Storage**

Objects in a program exist in memory during the application's runtime. However, their data often needs to persist beyond a single run (e.g., student data, course content).

Example:

Our **Course** object's details might be saved in a database. When the website needs to display the course, it'll fetch the data, populate a new **Course** object, and use that object during runtime.

**Note Points**:

* Object-relational mapping (ORM) is a technique that connects objects in code to database entries.

**8. Object's Lifespan**

Objects are created and eventually destroyed. The time between creation and destruction is the object's lifespan.

Example:

When a new course is created on our platform, a **Course** object is instantiated. It exists as long as the course is available. If the course is deleted, the corresponding object should also be destroyed to free resources.

**Note Points**:

* Efficiently managing an object's lifespan ensures that resources (like memory) are used optimally.

**9. Life Cycle of an Object**

Objects have a lifecycle:

1. **Creation (Instantiation)**: When an object is created.
2. **Usage**: When you're accessing and modifying the object's attributes and invoking its methods.
3. **Destruction**: In many languages, there's a mechanism to destroy objects when they're no longer needed, often using garbage collection.

**Note Points**:

* It's essential to manage object resources efficiently, especially in environments without automatic garbage collection.

**10. Objects Instantiation (Creating) and Constructors**

Creating an object is called instantiation.

When an object is created, it often requires some initial setup. This setup is typically done using constructors.

Example:

For the **Course** object, when a new course is created, you might want to set its title, initial content, and perhaps the instructor.

Once an object is created, you can interact with its attributes and methods.

*# Create a student object from the Student class*

John = Student (name="John Doe", email="john.doe@example.com")

*# Interact with the object*

John. enroll (course="Mathematics 101")

**Note Points**:

* An object is an instance of a class.
* When you create an object, you're instantiating a class.
* Constructors are special methods that help in setting the initial state of an object.
* Some languages allow for multiple constructors, offering various ways to initialize an object.
* You interact with the object using dot notation to access its attributes and methods.

**11. Object Finalization and Destructors**

Just as objects have a beginning (construction), they often have an end, especially in languages without automatic garbage collection.

Example:

If a **Course** object loads some resources (like videos), a destructor might ensure those resources are freed when the object is no longer needed.

**Note Points**:

* Destructors are methods that are invoked when an object is about to be destroyed.
* They help in cleaning up resources and ensuring the system doesn't leak memory.

**12. Object Relationships**

Objects often relate to each other in various ways:

* **Association**: A simple bi-directional relationship. A **Student** is associated with a **Course** they've enrolled in.
* **Aggregation**: Represents a "whole-part" relationship but not a strong ownership. A **Course** has multiple **Lesson** objects, but lessons can exist independently.
* **Composition**: A strong "whole-part" relationship. If our **Course** has an embedded **Syllabus**, the syllabus can't exist without the course.

**Note Points**:

* Understanding relationships is key to designing robust systems.
* It ensures clarity when defining interactions between objects.

**13. Objects Interacting with Each Other**

Objects don't exist in isolation; they often interact with each other.

For example, when a **Student** enrolls in a **Course**, the **enroll** method of the **Student** object might interact with the **add\_student** method of the **Course** object.

**Note Points**:

* Objects can have relationships with one another (like association, aggregation, composition).
* One object can use another object's methods and attributes.

**14. Object Copy vs. Reference**

In many languages, you can either reference an existing object or create a copy of it.

Example:

If two instructors are referencing the same **Course** object, changes made by one instructor will be visible to the other. However, if they have separate copies, changes by one won't affect the other's copy.

**Note Points**:

* This distinction is crucial for understanding how data changes and how those changes propagate through the system.

**16. Immutability**

Some objects are designed to be immutable, meaning their state can't change after they're created.

Example:

Once a **Certificate** object is created after a student completes a course, it might be designed to be immutable to ensure its integrity.

**Note Points**:

* Immutability can simplify design, improve performance, and enhance security.

**17. Concurrency Issues with Objects**

If multiple users or processes try to modify an object simultaneously, it can lead to inconsistent data.

Example:

Two students trying to enroll in the last spot of a **Course** simultaneously could lead to overbooking.

**Note Points**:

* Techniques like locking, atomic operations, or transactional databases can help manage concurrent access.

**Summary:**

Understanding objects is fundamental in OOP. They encapsulate data for the user and provide a clear and organized method to manage and manipulate that data. By using objects, we can create robust and scalable systems, like our educational website, and represent real-world entities and their interactions effectively

Objects are the cornerstone of OOP. They allow for representing real-world entities, encapsulating data and behavior, and providing modular and reusable components. When dealing with objects, it's essential to understand their identity, state, behavior, and relationships. This understanding, combined with knowledge of their lifecycle and the challenges posed by issues like concurrency, forms the foundation of effective object-oriented design and programming.

**Extra Points**

**1. Static vs. Instance Members**

Classes can have static members (attributes/methods) that belong to the class itself, rather than to instances of the class.

Example:

All **Student** objects might reference a static **school\_name** attribute, which remains consistent across all student objects.

**Note Points**:

* Static members are shared across all instances.
* They can be accessed without creating an instance of the class.

**2. Overloading and Overriding**

Overloading allows a class to have multiple methods with the same name but different parameters. Overriding lets a subclass provide a specific implementation for a method that's already defined in its superclass.

Example:

* **Overloading**: The **Student** class might have multiple **register** methods; one takes email and password, while another takes additional personal details.
* **Overriding**: If our platform has a premium course subclass called **Premium Course**, it might override the base **Course** class's **price** method to incorporate additional charges.

**Note Points**:

* Overloading is about method signatures (name + parameters).
* Overriding concerns the method's implementation in derived classes.

**3. Object's Metadata and Reflection**

Some languages allow for reflection, which means you can inspect the properties and capabilities of an object at runtime.

Example:

You might want to inspect a **Lesson** object to see all its methods, then dynamically call a method based on user input.

**Note Points**:

* Reflection is powerful but can introduce complexity.
* It's handy for building flexible and generic code, like plugins or extensions.

**4. Nested or Inner Classes**

Classes can be defined inside other classes. Such nested classes can access private attributes of the outer class.

Example:

Inside the **Course** class, you might have a nested **CourseStatistics** class that calculates and holds data about course performance, views, and other metrics.

**Note Points**:

* Nested classes are a way of logically grouping classes.
* They can lead to more readable and maintainable code.

**Summary:**

Objects, while a foundational concept in OOP, have numerous intricacies. From their lifecycle to their interactions, the depth and breadth of considerations around objects are vast. As you delve into advanced object-oriented design and real-world applications, these nuances become increasingly critical. Understanding them deeply not only prepares you for complex software development challenges but also aids in creating efficient, robust, and scalable systems.

## Class

## c

## c

## c

## c

## c

## c

## Dictionary

**Slicing Dictionaries:**

**Key Point**: Unlike lists or strings, dictionaries in Python don't support native slicing due to their key-value pair nature. However, subsets of dictionaries can be extracted using other techniques.

**Practical Techniques:**

1. **Dictionary Comprehension**:
   * Most common way to extract subsets from dictionaries.

Python Copy code

dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

subset = {k: dic[k] for k in list(dic.keys())[:2]} # Grabs first two key-value pairs.

**Note**: Remember the dictionary order. As of Python 3.7, dictionaries maintain the order of insertion.

1. **Using itertools.islice**:
   * Can be more efficient for larger dictionaries as it avoids creating an intermediate list of keys.

Python Copy code

from itertools import islice

subset = dict(islice(dic.items(), 2)) # Grabs first two key-value pairs.

**Note**: Remember to convert the **islice** object back to a dictionary.

1. **Conditions for Subsetting**:
   * Use conditions to filter out unwanted key-value pairs.

Python Copy code

even\_subset = {k: v for k, v in dic.items() if v % 2 == 0}

**Note**: This is immensely useful when dealing with large datasets and you need data based on certain criteria.

**Dictionary Merging:**

**Key Point**: Combining multiple dictionaries can be essential when aggregating data from different sources.

1. **Using update() method**:
   * Modifies the first dictionary in-place.

Python Copy code

d1 = {'a': 1, 'b': 2}

d2 = {'c': 3}

d1.update(d2)

**Note**: This method modifies the original dictionary and doesn't return a new one.

1. **Using {\*\*d1, \*\*d2} unpacking**:

Python Copy code

merged = {\*\*d1, \*\*d2}

**Note**: This creates a new dictionary without modifying the originals.

**Default Values with setdefault and defaultdict:**

**Key Point**: Handling missing keys gracefully can prevent errors and simplify code.

1. **Using setdefault**:

Python Copy code

count = {}

for item in ['a', 'b', 'a']:

count.setdefault(item, 0)

count[item] += 1

**Note**: If the key isn't present, it's set with the default value.

1. **Using defaultdict from collections**:

Python Copy code

from collections import defaultdict

count = defaultdict(int)

for item in ['a', 'b', 'a']:

count[item] += 1

**Note**: **defaultdict** automatically handles missing keys based on the provided default type.

**Nested Dictionaries and JSON Data:**

**Key Point**: In data-centric roles, you'll frequently work with nested data, often sourced from JSON.

1. **Deep Access with Helper Functions**:
   * Sometimes you need to access a deep key, which might cause a KeyError if any intermediate key is missing.

pythonCopy code

def deep\_get(dictionary, \*keys):

for key in keys:

try:

dictionary = dictionary[key]

except KeyError:

return None

return dictionary

**Note**: This function lets you safely access **dictionary[key1][key2]...[keyN]** without errors.

1. **Flattening Nested Dictionaries**:
   * Sometimes you need to transform nested data into a flat form for easier processing or storage.

Python Copy code

def flatten(d, parent\_key='', sep='\_'):

items = {}

for k, v in d.items():

new\_key = f"{parent\_key}{sep}{k}" if parent\_key else k

if isinstance(v, dict):

items.update(flatten(v, new\_key, sep=sep))

else:

items[new\_key] = v

return items

**Note**: Useful for saving to CSV or database tables where nested structures aren't as convenient.

**Dictionary Keys & Hashing:**

**Key Point**: Dictionary keys must be hashable, meaning they must be immutable. This is why lists can't be used as dictionary keys, but tuples can, as long as they contain only hashable items.

1. **Custom Objects as Keys**:
   * If creating custom classes, you can make instances usable as dictionary keys.

pythonCopy code

class MyClass:

def \_\_init\_\_(self, value1, value2):

self.value1 = value1

self.value2 = value2

def \_\_hash\_\_(self):

return hash((self.value1, self.value2))

def \_\_eq\_\_(self, other):

return self.value1 == other.value1 and self.value2 == other.value2

**Note**: Implementing **\_\_hash\_\_** and **\_\_eq\_\_** allows instances to be used as dictionary keys.

Top of Form

Bottom of Form

## Level 2 Heading Goes Here

|  |  |  |
| --- | --- | --- |
| On the Insert tab, the galleries include items that are designed to coordinate with the overall look of your document. You can use these galleries to insert tables, headers, footers, lists, cover pages, and other document building blocks.  When you create pictures, charts, or diagrams, they also coordinate with your current document look. You can easily change the formatting of selected text in the document text by choosing a look for the selected text from the Quick Styles gallery on the Home tab.  You can also format text directly by using the other controls on the Home tab. Most controls offer a choice of using the look from the current theme or using a format that you specify directly. Most controls offer a choice of using the look from the current theme or using a format that you specify directly. |  |  |

### Level 3 heading goes here

On the Insert tab, the galleries include items that are designed to coordinate with the overall look of your document. You can use these galleries to insert tables, headers, footers, lists, cover pages, and other document building blocks. When you create pictures, charts, or diagrams, they also coordinate with your current document look.

You can easily change the formatting of selected text in the document text by choosing a look for the selected text from the Quick Styles gallery on the Home tab. You can also format text directly by using the other controls on the Home tab. Most controls offer a choice of using the look from the current theme or using a format that you specify directly.

## Again Level 2 Heading

You can easily change the formatting of selected text in the document text by choosing a look for the selected text from the Quick Styles gallery on the Home tab. You can also format text directly by using the other controls on the Home tab. Most controls offer a choice of using the look from the current theme or using a format that you specify directly.

* On the Insert tab, the galleries include items that are designed to coordinate with the overall look of your document. You can use these galleries to insert tables, headers, footers, lists, cover pages, and other document building blocks. When you create pictures, charts, or diagrams, they also coordinate with your current document look.
* You can easily change the formatting of selected text in the document text by choosing a look for the selected text from the Quick Styles gallery on the Home tab. You can also format text directly by using the other controls on the Home tab. Most controls offer a choice of using the look from the current theme or using a format that you specify directly.

## Level 2 Heading Again

### Level 3 heading goes here

On the Insert tab, the galleries include items that are designed to coordinate with the overall look of your document. You can use these galleries to insert tables, headers, footers, lists, cover pages, and other document building blocks. When you create pictures, charts, or diagrams, they also coordinate with your current document look.

You can easily change the formatting of selected text in the document text by choosing a look for the selected text from the Quick Styles gallery on the Home tab. You can also format text directly by using the other controls on the Home tab. Most controls offer a choice of using the look from the current theme or using a format that you specify directly.

#### Level 4 heading goes here

On the Insert tab, the galleries include items that are designed to coordinate with the overall look of your document. You can use these galleries to insert tables, headers, footers, lists, cover pages, and other document building blocks. When you create pictures, charts, or diagrams, they also coordinate with your current document look.

#### Level 4 heading again

On the Insert tab, the galleries include items that are designed to coordinate with the overall look of your document. You can use these galleries to insert tables, headers, footers, lists, cover pages, and other document building blocks. When you create pictures, charts, or diagrams.

You can easily change the formatting of selected text in the document text by choosing a look for the selected text from the Quick Styles gallery on the Home tab. You can also format text directly by using the other controls on the Home tab. Most controls offer a choice of using the look from the current theme or using a format that you specify directly.

You can easily change the formatting of selected text in the document text by choosing a look for the selected text from the Quick Styles gallery on the Home tab. You can also format text directly by using the other controls on the Home tab. Most controls offer a choice of using the look from the current theme or using a format that you specify directly.

## Level 2 Heading AGain

### Level 3 heading goes here

On the Insert tab, the galleries include items that are designed to coordinate with the overall look of your document. You can use these galleries to insert tables, headers, footers, lists, cover pages, and other document building blocks. When you create pictures, charts, or diagrams, they also coordinate with your current document look.

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Data Set 1** | **Data Set 2** | **Data Set 3** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

You can easily change the formatting of selected text in the document text by choosing a look for the selected text from the Quick Styles gallery on the Home tab. You can also format text directly by using the other controls on the Home tab. Most controls offer a choice of using the look from the current theme or using a format that you specify directly.

#### Level 4 heading goes here

On the Insert tab, the galleries include items that are designed to coordinate with the overall look of your document. You can use these galleries to insert tables, headers, footers, lists, cover pages, and other document building blocks. When you create pictures, charts, or diagrams, they also coordinate with your current document look.