Object Oriented Programming

Welcome to the exciting world of algorithms and data structures! This book is designed to gently introduce you to fundamental concepts like searching, sorting, and algorithm complexity, all while using the familiar and accessible environment of Google Colab with Python.

We'll start with a practical example to build intuition before diving into the technical details. Let's imagine you're building a music app...

1 Building a Music App: A Hierarchical Approach

Imagine you're creating your own music app. You want to organize your music collection effectively. Initially, you might think of simply listing all your songs in a long, single list. But that quickly becomes unwieldy as your collection grows.

Then your manager recommends that you could also integrate a podcast feature, so you think to yourself "oh no, more mess".

This is where the power of relationships and hierarchy comes in. You realize that songs and podcasts, while distinct, share common characteristics: they both have a title, a duration, and maybe even a release year. This commonality suggests a hierarchical structure:

- **Audio (Base):** A general category encompassing both songs and podcasts, sharing common attributes like title and duration.
- **Song (Derived):** A specific type of audio with attributes like artist.
- **Podcast (Derived):** Another specific type of audio with attributes like host.

This organization mirrors real-world hierarchies like the animal kingdom or organizational structures. Just as a manager was once an employee, a song and a podcast both begin as a common type of audio that share certain basic features.

This is the core concept of **Object-Oriented Programming (OOP)**, where we model relationships between different entities in our code.

Now, let's translate this into Python code within Google Colab.

2 Object-Oriented Programming (OOP) Fundamentals

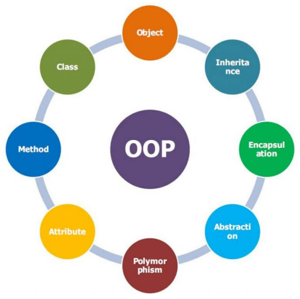


Figure 1: A hierarchical structure of the music app showing Audio, Song, and Podcast classes.

OOP is a programming paradigm that revolves around the concept of "objects," which combine data (attributes) and actions (methods).

- **Class:** A blueprint or template for creating objects. In our music app example, `Song`, and `Podcast` are classes.
- **Object:** An instance of a class. A specific song with a title and artist is an object.
- **Attributes:** Data associated with an object. Examples: `title`, `artist`, `host`, `duration`.
- **Methods:** Functions that an object can perform. (We'll add some later!)

Encapsulation: Encapsulation is the bundling of data (attributes) and methods that operate on that data into a single unit, or class.

Let's look at some code:

```python
class Song:
    def __init__(self, title, artist):
        self.title = title
        self.artist = artist

my_song = Song("Bones", "Imagine Dragons")
print(my_song.title) # Output: Bones
```

Here, `Song` is a class, `my_song` is an object of the `Song` class, and `title` and `artist` are attributes. The `__init__` method is a special method called a constructor. It's automatically called when a new object is created and initializes the object's attributes.

3 Inheritance: Building Hierarchies

Inheritance is a powerful feature of OOP that allows you to create new classes (derived classes) based on existing classes (base classes). The derived class inherits attributes and methods from the base class, promoting code reuse and establishing relationships.

Let's create our audio hierarchy:

```python
class Audio:
    def __init__(self, title):
        self.title = title

class Song(Audio):  # Song inherits from Audio
    def __init__(self, title, artist):
        super().__init__(title)  # Call the parent class's
            constructor
        self.artist = artist

class Podcast(Audio): # Podcast inherits from Audio
    def __init__(self, title, host):
        super().__init__(title)
        self.host = host

my_song = Song("Bones", "Imagine Dragons")
my_podcast = Podcast("Mind in AI", "HS")

print(my_song.title)  # Output: Bones
print(my_podcast.host) # Output: HS
```

- We define a base class, `Audio`, with a `title` attribute.
- The `Song` and `Podcast` classes inherit from `Audio` using the syntax `class Song(Audio):`.
- `super().__init__(title)` calls the constructor of the base class (`Audio`) to initialize the `title` attribute. This avoids redundant code.
- Now, both `Song` and `Podcast` automatically have the `title` attribute, along with their specific attributes (`artist` and `host`, respectively).

4 Protected and Private Members

Sometimes, you want to control the visibility of attributes and methods within a class. Python offers mechanisms for this:

- **Protected Members:** Attributes or methods prefixed with a single underscore (_). They are intended to be internal to the class and its subclasses but can still be accessed from outside.

- **Private Members:** Attributes or methods prefixed with a double underscore (__). They are meant to be strictly internal to the class and are more difficult to access from outside.

```python
class Song(Audio):
    def __init__(self, title, artist):
        super().__init__(title)
        self.artist = artist
        self._duration = 3 # Protected member

class Podcast(Audio):
    def __init__(self, title, host):
        super().__init__(title)
        self.host = host
        self.__rating = 5  # Private member

    def get_rating(self):
        return self.__rating  # Access private member
            through a getter method

my_song = Song("Bones", "Imagine Dragons")
my_podcast = Podcast("Mind in AI", "HS")

print(my_song._duration) # Accessing a protected member
    (generally discouraged)
print(my_podcast.get_rating()) # Accessing a private member
    through a getter method
```

Trying to directly access `my_podcast.__rating` will result in an error. This is because of **name mangling**. Python internally renames private attributes to prevent accidental access from outside the class.

Name Mangling: To truly hide the name, Python uses name mangling to create a function called `_className__attributeName`

The `get_rating` method provides a controlled way to access the private `__rating` attribute. These methods are often called "getters" (for retrieving values) and "setters" (for modifying values). They encapsulate the internal workings of the class and provide an interface for interacting with it.

5 Sorting: Putting Things in Order

Now, let's move on to sorting. Sorting is the process of arranging elements in a specific order (e.g., ascending or descending).

Imagine you're a gambler who's trying to arrange a deck of cards in his hands. You are dealt with a card from the deck, and you have to arrange the cards one by one to the correct place. The way you arrange the cards is exactly the thought process of insertion sort.

Let's focus on **Insertion Sort**, a simple and intuitive sorting algorithm.

Insertion Sort Intuition:

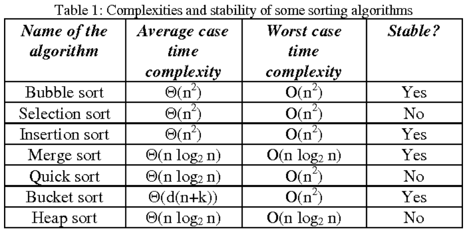
The way you sort cards that you pick from the deck is exactly how Insertion sort works. You maintain a sorted subarray and iteratively insert elements from the unsorted portion into their correct positions within the sorted subarray.

Here is the general idea of an Insertion sort Algorithm:

1. Start with the second element in the array. This element is what will be compared with the previously seen array.
2. If the previous element is smaller than the value that is selected, that means it's at the right place, so continue to the next value.
3. If the previous value is greater than the currently selected value, use a temporary variable to copy the current value. Shift the greater element up to the next element. Keep doing this till you reach a smaller value, and copy the variable at the next element.

5.1 Best, Average, and Worst Cases

Table 1: Complexities and stability of some sorting algorithms



Name of the algorithm	*Average case time complexity*	*Worst case time complexity*	*Stable?*
Bubble sort	$\Theta(n^2)$	$O(n^2)$	Yes
Selection sort	$\Theta(n^2)$	$O(n^2)$	No
Insertion sort	$\Theta(n^2)$	$O(n^2)$	Yes
Merge sort	$\Theta(n \log_2 n)$	$O(n \log_2 n)$	Yes
Quick sort	$\Theta(n \log_2 n)$	$O(n^2)$	No
Bucket sort	$\Theta(d(n+k))$	$O(n^2)$	Yes
Heap sort	$\Theta(n \log_2 n)$	$O(n \log_2 n)$	No

When analyzing sorting algorithms, we consider different scenarios:

- **Best Case:** The array is already sorted. In insertion sort, the number of comparisons is minimal (linear time complexity).
- **Average Case:** The array is randomly ordered. Insertion sort performs reasonably well (quadratic time complexity).
- **Worst Case:** The array is sorted in reverse order. Insertion sort requires the maximum number of comparisons and shifts (quadratic time complexity).

Insertion sort's best case performance is linear, while bubble sort still takes quadratic time. Therefore, if it is already sorted, Insertion sort is the better method to use.

6 Conclusion

This book has provided a gentle introduction to fundamental programming concepts such as Object Oriented Programming, hierarchical design, Encapsulation, and sorting algorithms. We used the real world example of building a music app to contextualize the code that you would be writing. By experimenting with code examples and building your own projects, you'll solidify your understanding and gain valuable problem-solving skills. Happy coding!