Object Oriented Programming

First, we will discuss When, Why and What is OOP. Then, we will jump on to key concepts in OOP.

WHEN: In the 1950s–1960s, programs were written procedurally, with data and functions separate. As systems grew complex, developers needed better ways to manage code, track state, reuse logic, and model real-world problems.

WHY: Even we can code without OOP. But, OOP makes our life easier by giving us way to make code more readable, maintainable, scalable in large systems and extendable.

WHAT: Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects" — self-contained units that combine data (attributes) and behavior (methods).

Key Concepts in OOP

- **Class:** A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.
- **Object:** An object is an instance of a class. It represents a specific entity that holds data and can perform actions defined by its class.
- **Encapsulation:** Encapsulation is the practice of bundling data and related methods into a single unit (class) and restricting direct access to internal details. This protects the object's state and enforces controlled interaction.
- **Inheritance:** Inheritance allows one class (child) to inherit properties and behaviors from another class (parent). It promotes code reuse and logical hierarchy.
- **Abstraction:** Abstraction means hiding unnecessary implementation details and showing only the relevant features to the user. It helps reduce complexity and focus on essential behavior. It is more about what is there/what needs to be there instead of how it is there/how to implement.
- **Polymorphism:** Polymorphism allows the same method name to behave differently depending on the object calling it. It enables flexibility and extensibility in code.

Let's take a scenario to understand these core concepts of OOP better.

Scenario: *Write code for extracting content from filetypes like HTML and PDF. (Which can be extended further to different filetypes like CSV, XLSX, XLS, XML., so on.)*

Let's understand the key concepts one by one by implementing with OOP as well as with procedural / functional programming and the complexities without OOP. Using Python language for implementation.

Class

Let's say we need to extract metadata and content for given html file. To do that I am defining a class here which will hold data like file_path, metadata and extracted content along with the methods needed for extracting metadata and content.

It is just a blueprint / template. It doesn't have any real values in it.

```
class HTMLExtractor:
    """A class to extract metadata and
     content from HTML files."""

    def __init__(self, file_path):
        self.file_path = file_path
        self.metadata = {}
        self.content = None

    def extract_metadata(self): ...

    def extract_content(self): ...
```

Object

Now we want pass in some real values which will be used for actual extraction of metadata and content.

This is how we create an object (which will hold some real data and methods to work on it) from class.

```
# Object instantiation for HTML extraction
html_extractor = HTMLExtractor("example.html")
```

Same thing can be implemented using simple functions (without OOP) as well. But this approach will get complicated once we started implementing functions for pdf and other files and as well as additional functionality for each file extraction. Will explain these complexity as we go on.

```
def extract_html_metadata(file_path): ...


def extract_html_content(file_path): ...


file = "example.html"
metadata = extract_html_metadata(file)
content = extract_html_content(file)
```

Encapsulation

We already have data and object at one place. Now, we want to restrict access to directly access the data so that we can avoid unnecessary changes from any developer who writes code later on top of it.

Before diving into restricting access lets understand different types of access specifiers in python. They don't enforce any restrictions they are just for readability and good practice.

- Public attribute accessible from outside the class.
- Protected which can be accessed within the class and subclasses.
- Private which cannot be accessed outside the class.

```
class HTMLExtractor:
    """A class to extract metadata and
     content from HTML files."""

    def __init__(self, file_path):
        self.file_path = file_path  # Public
        self._metadata = {}  # Protected
        self.__content = None  # Private

    # Public
    def extract_content(self): ...
    # Protected
    def _extract_metadata(self): ...
    # Private
    def __extract_encoding(self): ...



html_extractor = HTMLExtractor("example.html")
print(html_extractor.file_path)
print(html_extractor._metadata)
# Raises error.
print(html_extractor.__content)
# Name Mangling: accessing private variable.
print(html_extractor._HTMLExtractor__content)
```

Now we will define additional methods to access data.

- Generic way to set and get (setters and getters). We can add validations as well while setting and getting like only allow to set if metadata has author in it.

```
class HTMLExtractor:
    """A class to extract metadata and
     content from HTML files."""

    def __init__(self, file_path):
        self.file_path = file_path  # Public
        self._metadata = {}  # Protected
        self.__content = None  # Private

    def extract_content(self): ...
    def _extract_metadata(self): ...

    def get_metadata(self):
        return self._metadata

    def set_metadata(self, metadata):
        self._metadata = metadata


html_extractor = HTMLExtractor("example.html")
html_extractor.set_metadata({"author": "John"})
print(html_extractor.get_metadata())
```

- Pythonic way of getting and setting. Using @property and @item.setter

```
class HTMLExtractor:
    """A class to extract metadata and
     content from HTML files."""

    def __init__(self, file_path):
        self.file_path = file_path  # Public
        self._metadata = {}  # Protected
        self.__content = None  # Private

    def extract_content(self): ...
    def _extract_metadata(self): ...

    @property
    def metadata(self):
        return self._metadata

    @metadata.setter
    def metadata(self, metadata):
        if "author" not in metadata:
            raise ValueError("No Author")
        self._metadata = metadata


html_extractor = HTMLExtractor("example.html")
html_extractor.metadata = {"author": "John"}
print(html_extractor.metadata)
```

The @property method for content must be defined before the setter for content. You cannot decorate a setter before defining the property.

Let see the approach without OOP. Which will make variables vulnerable to modifications.

```
# Dictionary to hold all state
html_extractor = {
    'file_path': "example.html",
    '_metadata': {},
    '__content': None,
}

def get_metadata(extractor_dict):
    return extractor_dict['_metadata']

def set_metadata(extractor_dict, metadata):
    if "author" not in metadata:
        raise ValueError("No Author")
    extractor_dict['_metadata'] = metadata

def extract_content(extractor_dict):
    pass

def _extract_metadata(extractor_dict):
    pass
# setter
set_metadata(html_extractor, {"author": "John"})
# getter
print(get_metadata(html_extractor))

# The value can be modified without using the setter
html_extractor['_metadata'] = {"authors": "Janes"}
```

In Python OOP also we can modify the values which are declared as protected and private.

- Properties and naming conventions protect you from accidental misuse, not deliberate subversion.
- You can't prevent a determined person from changing private/protected data, but you can make it obvious they're breaking the class contract.
- Most IDE's will highlight such misusage.

Inheritance

Let's say I want to implement extractor for PDF now. One way is to copy the HTML class code and change the name but most of the data and methods will be same across both the classes. To reuse the code, we can use inheritance by declaring a base class with some common properties and inherit those base class properties to child classes.

```
class FileExtractor:
    """A base class to extract metadata and
     content from files."""

    def __init__(self, file_path):
        self.file_path = file_path  # Public
        self._metadata = {}  # Protected
        self.__content = None  # Private

    @property
    def metadata(self):
        return self._metadata

    @metadata.setter
    def metadata(self, metadata):
        if "author" not in metadata:
            raise ValueError("No Author")
        self._metadata = metadata


class HTMLExtractor(FileExtractor):
    """A class to extract metadata and
     content from HTML files."""
    def extract_content(self):
        # Logic to extract content from HTML file
        pass

    def _extract_metadata(self):
        # Logic to extract metadata from HTML file
        pass


class PDFExtractor(FileExtractor):
    """A class to extract metadata and
     content from PDF files."""
    def extract_content(self):
        # Logic to extract content from PDF file
        pass

    def _extract_metadata(self):
        # Logic to extract metadata from PDF file
        pass


html_extractor = HTMLExtractor("example.html")
html_extractor.metadata = {"author": "John"}
print(html_extractor.metadata)

pdf_extractor = PDFExtractor("example.pdf")
pdf_extractor.metadata = {"author": "Jane"}
print(pdf_extractor.metadata)
```

Here we defined a base class FileExtractor with data and methods for getting and setting of metadata of file.

Let's say we want to implement extraction for csv and xlsx. Now, we can simply extend the base class and implement methods for content extraction.

Different types of inheritance

- Single
- Multilevel
- Multiple
- Hierarchical
- Hybrid

Let see the without OOP.

```
# Procedural "Extractor" Implementation

# --- Common Utilities ---
def create_extractor(file_path):
    return {
        'file_path': file_path,
        'metadata': {},
        '__content': None
    }

def set_metadata(extractor, metadata):
    if "author" not in metadata:
        raise ValueError("No Author")
    extractor['metadata'] = metadata

def get_metadata(extractor):
    return extractor['metadata']

# --- HTML-specific Operations ---
def extract_content_html(extractor):
    # Example logic (dummy)
    pass

def extract_metadata_html(extractor):
    # Example logic (dummy)
    extractor['metadata'] = {"author": "john"}

# --- PDF-specific Operations ---
def extract_content_pdf(extractor):
    # Example logic (dummy)
    pass

def extract_metadata_pdf(extractor):
    # Example logic (dummy)
    extractor['metadata'] = {"author": "jane"}

# --- Usage ---
html_extractor = create_extractor("example.html")
set_metadata(html_extractor, {"author": "John"})
print(get_metadata(html_extractor))
extract_content_html(html_extractor)

pdf_extractor = create_extractor("example.pdf")
set_metadata(pdf_extractor, {"author": "Jane"})
print(get_metadata(pdf_extractor))
extract_content_pdf(pdf_extractor)
```

Disadvantages of not using OOP Inheritance here.

- Shared logic must be copy-pasted, increasing duplication and inconsistency.
- Specialized behavior for each type requires separate functions and more code to maintain.
- There's no clear hierarchy or structure between general and specific logic; all code is flat.
- Cannot treat all "extractors" the same way; you must write branching logic for each type.
- Adding a new type means creating more functions and updating code in many places.
- No way to enforce that all extractors have required functions.
- Duplication is common, making code harder to maintain and update.

Using OOP Inheritance makes easy to extend and implementing new extractors for new file types. Also, reduces duplication of code. We just need to change at one place instead of multiple places.

Abstraction

Abstraction is about defining what features or behavior are required rather than dictating how they should be implemented. For example, if I want you to develop an extractor for XML files, I don't want to specify every detail of *how* you should write the extraction logic; I just want to ensure that your extractor provides exactly the functionality I need.

Without abstraction, I would have to describe my requirements verbally or in documentation, but I couldn't guarantee that you would implement all the necessary parts, or that your implementation would fit seamlessly into the rest of the system.

With abstraction, however, I can define an abstract class or interface that specifies *what methods your extractor must provide* (such as extract_content). When you create your XML extractor, you are forced—by the structure of the code itself—to implement these required methods. This guarantees that your XML extractor does exactly what's needed, without me dictating how you should achieve it.

There are two variants of this one is complete abstraction called as interface another one abstract class with some shared functionality. Here are the examples for both. Using @abstractmethod makes the difference between both.

-Using an abstract class with some shared functionality.

```
"""Using an Abstract Class
(with some shared functionality)"""
from abc import ABC, abstractmethod

class FileExtractor(ABC):
    def __init__(self, filepath):
        self.filepath = filepath

    @abstractmethod
    def extract_content(self):
        pass

    def info(self):
        print(f"Extracting from: {self.filepath}")

class HTMLExtractor(FileExtractor):
    def extract_content(self):
        self.info()
        print("Extracting HTML content...")

class PDFExtractor(FileExtractor):
    def extract_content(self):
        self.info()
        print("Extracting PDF content...")

extractors = [
    HTMLExtractor('file.html'),
    PDFExtractor('file.pdf'),
]

for ext in extractors:
    ext.extract_content()
```

-Using abstract class an interface so that everything needs to be implemented at derived class.

```
"""Using an Interface-like Base Class
 (all methods abstract)"""
from abc import ABC, abstractmethod

class IFileExtractor(ABC):
    @abstractmethod
    def extract_content(self):
        pass

class HTMLExtractor(IFileExtractor):
    def extract_content(self):
        print("Extracting HTML content...")

class PDFExtractor(IFileExtractor):
    def extract_content(self):
        print("Extracting PDF content...")

extractors = [
    HTMLExtractor(),
    PDFExtractor(),
]

for ext in extractors:
    ext.extract_content()
```

Let's see without OOP

```
def extract_html_content():
    print("Extracting HTML content...")

def extract_pdf_content():
    print("Extracting PDF content...")

def extract_txt_content():
    print("Extracting TXT content...")

# Master function with if-else dispatch
def extract_content(filetype):
    if filetype == "html":
        extract_html_content()
    elif filetype == "pdf":
        extract_pdf_content()
    elif filetype == "txt":
        extract_txt_content()
    else:
        print("Unknown file type!")

file_types = ["html", "pdf", "txt"]
for ftype in file_types:
    extract_content(ftype)
```

Disadvantages of not using OOP Abstraction here.

- No unified interface for extractors; each function has to be called differently.
- Implementation details are exposed everywhere, reducing code clarity.
- Hard to hide "how" extraction works; users interact directly with low-level details.
- Adding or changing extraction logic increases code complexity and risk of errors.
- Difficult to enforce on what needs to be implemented.

Polymorphism

```python
from abc import ABC, abstractmethod

# Abstract base class
class FileExtractor(ABC):
    @abstractmethod
    def extract(self, filepath):
        pass

# Concrete class for HTML
class HtmlExtractor(FileExtractor):
    def extract(self, filepath):
        print(f"Extracting HTML : {filepath}")

# Concrete class for PDF
class PdfExtractor(FileExtractor):
    def extract(self, filepath):
        print(f"Extracting PDF: {filepath}")

# Function using polymorphism
def process_extraction(extractor, filepath):
    extractor.extract(filepath)

# Usage
extractors = [HtmlExtractor(), PdfExtractor()]
files = ['document.html', 'ebook.pdf']

for extractor, file in zip(extractors, files):
    process_extraction(extractor, file)
```

- Both HtmlExtractor and PdfExtractor implement the same extract() method name.
- The same method call (extract()) behaves differently, depending on which object (extractor) is used.
- process_extraction can work with any extractor, so your code is flexible and extensible—add more extractors (for Word, Excel, etc.) without changing the main logic.

Let's see an example without OOP.

```python
def extract_html(filepath):
    print(f"Extracting HTML: {filepath}")

def extract_pdf(filepath):
    print(f"Extracting PDF: {filepath}")

# Mapping file types to functions
extractors = {
    "html": extract_html,
    "pdf": extract_pdf,
}

def process_extraction(filetype, filepath):
    extractor_func = extractors.get(filetype)
    if extractor_func:
        extractor_func(filepath)
    else:
        print(f"No extractor for: {filetype}")

# Usage
files = [("html", "document.html"),
         ("pdf", "ebook.pdf")]

for filetype, filepath in files:
    process_extraction(filetype, filepath)
```

This approach "simulates" polymorphism by using the same **function call** (extract_func(filepath)) for different file types, letting each function define its own behavior.

Here are the disadvantages of not using OOP here.

- No Interface Enforcement: There's no guarantee all extractor functions follow the same signature or behavior, leading to possible runtime errors.
- Manual Dispatch Instead of Automatic: You must manually select the function using a dictionary, whereas true polymorphism allows the correct method to be called automatically on each object.
- Reduced Flexibility in Usage: You cannot pass different extractors around and treat them uniformly through a shared extract method, losing the main benefit of polymorphism.