

Fundamentals of Programming for Artificial Intelligence

Session 06
JSON - OOP

Instructors:

Dr. Lê Thanh Tùng

Dr. Nguyễn Tiến Huy

Content

- 1 JSON
- 2 OOP Terminologies
- 3 OOP Concepts

1. JSON

JSON Format

- JSON (JavaScript Object Notation) is a popular data format used for representing structured data
- JSON format is quite similar dictionary in Python

```
{  "emp_details": [  
    {  
        "emp_name": "Shubham",  
        "email": "ksingh.shubh@gmail.com",  
        "job_profile": "intern"  
    },  
    {  
        "emp_name": "Gaurav",  
        "email": "gaurav.singh@gmail.com",  
        "job_profile": "developer"  
    },  
    {  
        "emp_name": "Nikhil",  
        "email": "nikhil@geeksforgeeks.org",  
        "job_profile": "Full Time"  
    }  
  ]  
}
```

Parse JSON in Python

- The **json** module makes it easy to parse JSON strings and files containing JSON object
- In Python, JSON exists as a string

```
p = '{"name": "Bob", "languages": ["Python", "Java"]}'
```

- It's also common to store a JSON object in a file whose content can be read as a string

Parse JSON in Python

- For example,

```
import json #import json library to parse JSON Format

person = '{"name": "Bob", "languages": ["English", "French"]}'
person_dict = json.loads(person)

print( person_dict)
# Output: {'name': 'Bob', 'languages': ['English', 'French']}

print(person_dict['languages'])
# Output: ['English', 'French']
```

Parse JSON in Python

- For example, parse json from file

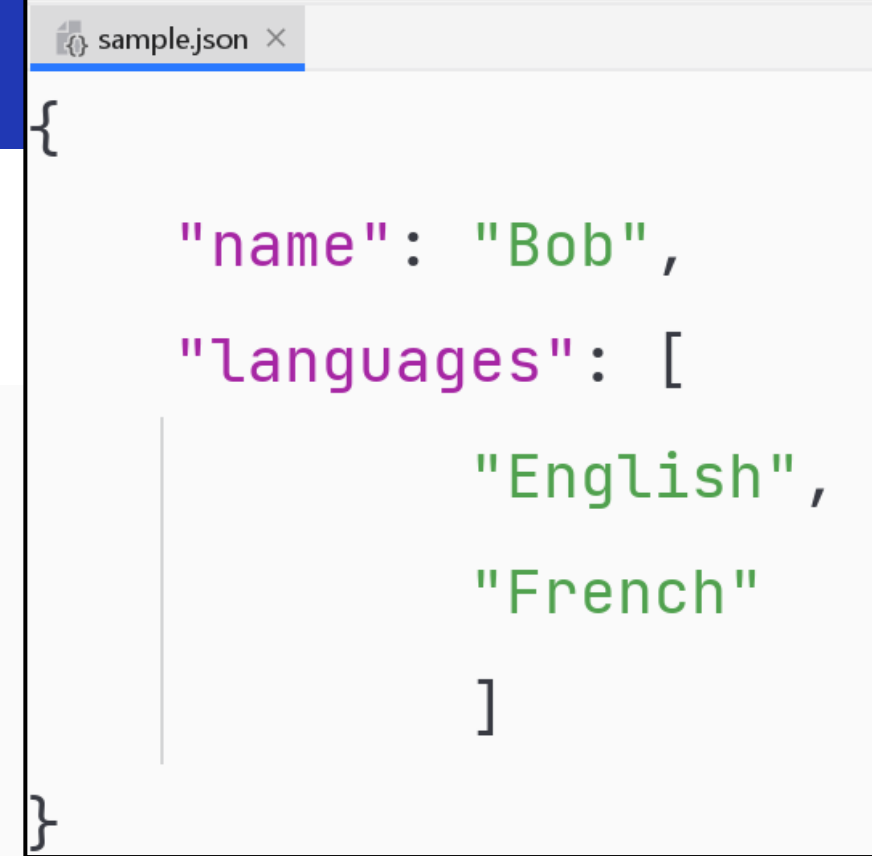
```
import json
```

```
inFile = open('sample.json', 'r')
```

```
data = json.load(inFile)
```

```
print(data)
```

```
# Output: {'name': 'Bob', 'languages': ['English', 'French']}
```



```
{  
    "name": "Bob",  
    "languages": [  
        "English",  
        "French"  
    ]  
}
```

Parse JSON in Python

- `json.load()` takes a file object and returns the json object
- `json.loads()` method can be used to parse a valid JSON string and convert it into a Python Dictionary

Convert to JSON string

- If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method
- By setting the `skipkeys` to `True` (default: `False`) we automatically skip the keys that are not of basic type

```
import json

Dictionary = {(1, 2, 3): 'Welcome',
              2: 'to', 3: 'Geeks', 4: 'for', 5: 'Geeks'}

json_string = json.dumps(Dictionary, skipkeys = True)
print(json_string)

# {"2": "to", "3": "Geeks", "4": "for", "5": "Geeks"}
```

Convert to JSON string

- If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method
- By setting the `skipkeys` to `True` (default: `False`) we automatically skip the keys that are not of basic type

```
import json

Dictionary = {(1, 2, 3): 'Welcome',
               2: 'to', 3: 'Geeks', 4: 'for', 5: 'Geeks'}

json_string = json.dumps(Dictionary, skipkeys = False)
# TypeError: keys must be str, int, float, bool or None, not tuple
print(json_string)
```

Convert to JSON string

- The **indent** parameter allows us to format the JSON array elements and object members in a more organized manner.
- A positive integer **indent** represents the number of spaces per level that should be used to indent the content.
- An indent level of 0 or negative will only insert newlines
- None (the default) selects the most compact representation.

Convert to JSON string

```
import json

content ={'name': 'Le Thanh Tung',
          'info': {
              'age': 18}
          }

json_string = json.dumps(content)
print(json_string)

# {"name": "Le Thanh Tung", "info": {"age": 18}}
```

```
import json

content ={'name': 'Le Thanh Tung',
          'info': {
              'age': 18}
          }

json_string = json.dumps(content, indent = 4)
print(json_string)

# {
#     "name": "Le Thanh Tung",
#     "info": {
#         "age": 18
#     }
# }
```

Convert to JSON string

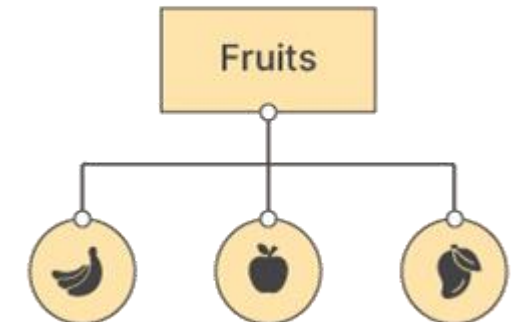
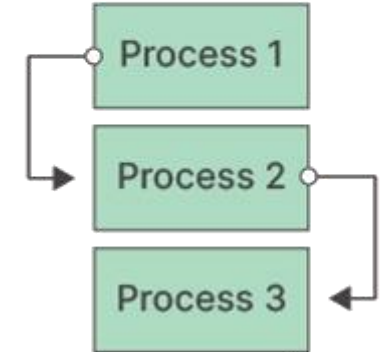
- Write a python program to find the student who has the highest score from the student list in JSON file.
- The format of input file is presented as follows:

```
{  
    "2212001": {  
        "name": "Nguyen Van A",  
        "age": 18,  
        "score": 9.2  
    },  
    "2212002": {  
        "name": "Tran Thi B",  
        "age": 18,  
        "score": 10.0  
    }  
}
```

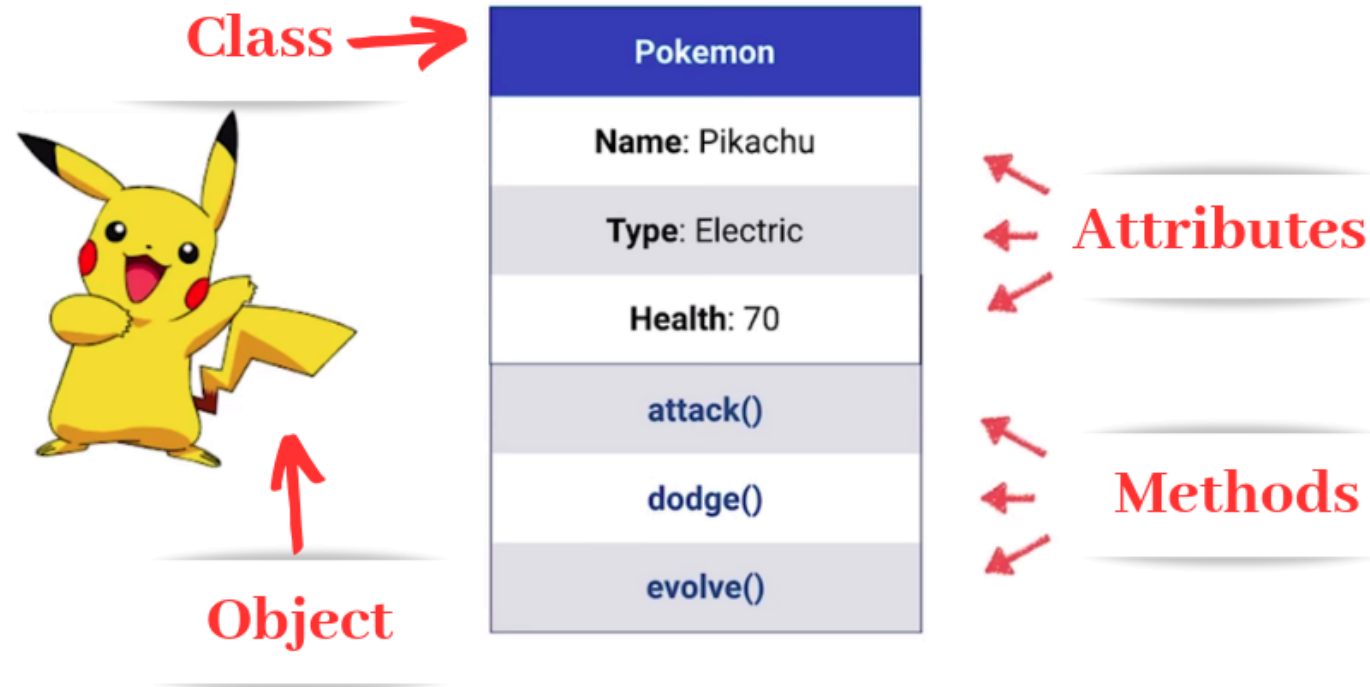
2. OOP Terminologies

Two basic programming paradigms:

- Procedural:
 - Organizing programs around functions or blocks of statements which manipulate data.
- Object-Oriented
 - combining data and functionality and wrap it inside what is called an object



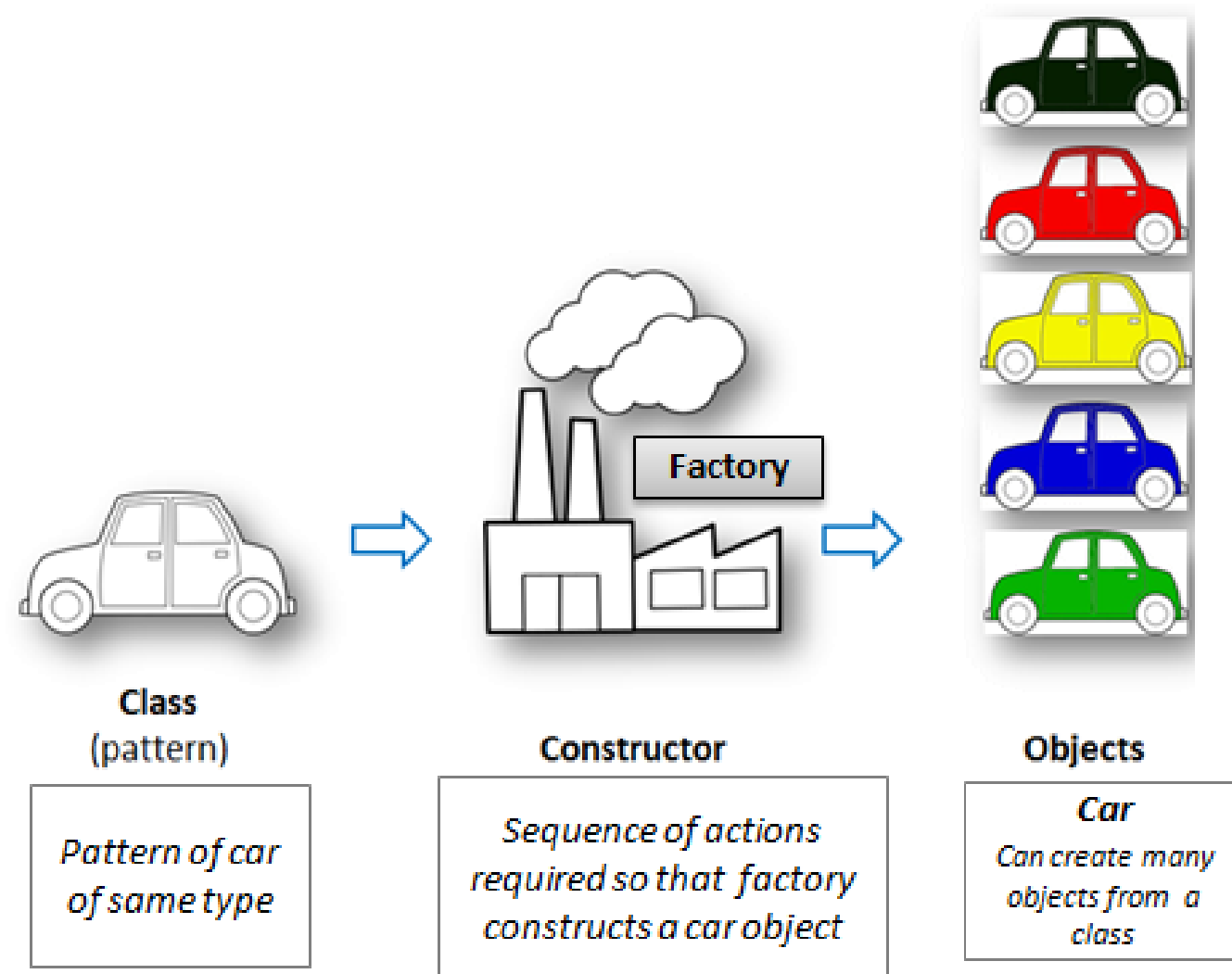
- Object-oriented programming (OOP) is a computer programming model that organizes software design around **data, or objects**, rather than functions and logic
- An object can be defined as a data field that has unique **attributes** and **behavior**



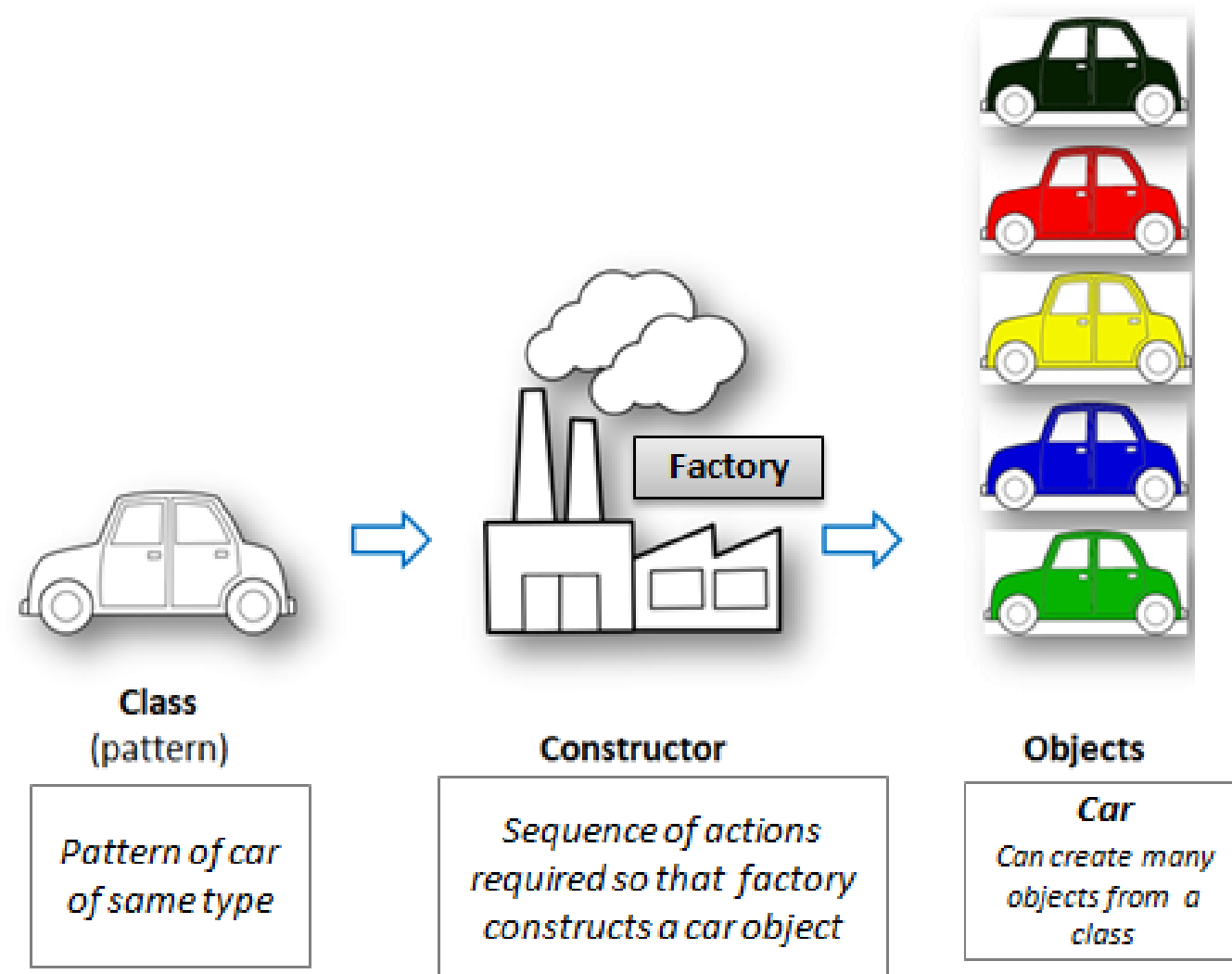
Object-Oriented (OOP) Vs Procedural Oriented (POP) Programming

OOP	POP
It is a bottom-up approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of Access modifiers “public”, “private”, “protected”	Doesn't use Access modifiers
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs
It supports inheritance	It does not support inheritance

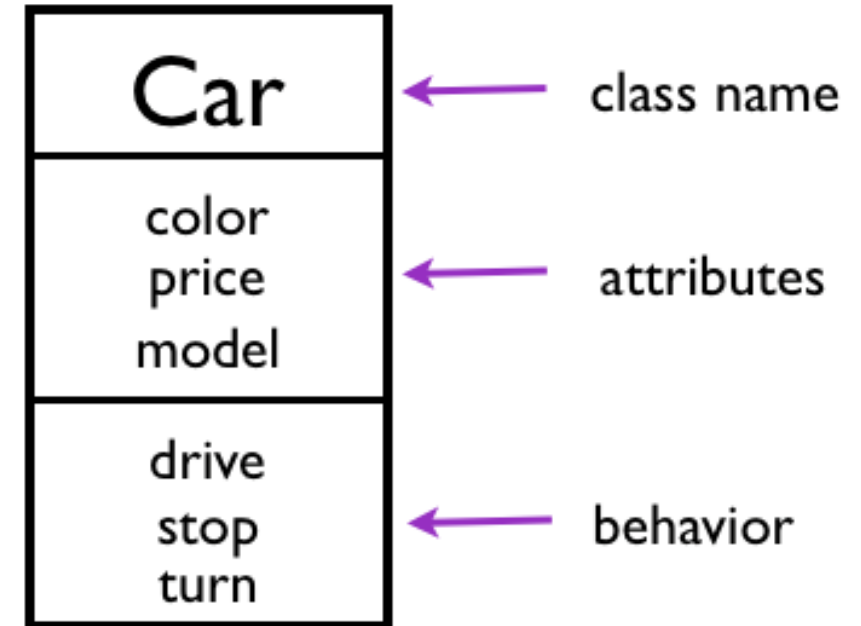
- A class is a collection of objects, or you can say it is a blueprint of objects defining the common **attributes (data/fields/instance variables/data members)** and **behavior (methods/member functions)**
- A class is a mechanism used to create new user-defined data structures
- Objects are instances of the class



- While the class is the blueprint, an instance is a copy of the class with actual values, literally an object belonging to a specific class
- It's not an idea anymore; it's an actual car,
 - like a car colored red
- Before you can create individual instances of an object, we must first specify what is needed by defining a class



- For example, think about the class of car:
 - What are the attributes of car?
 - What are the methods of car?

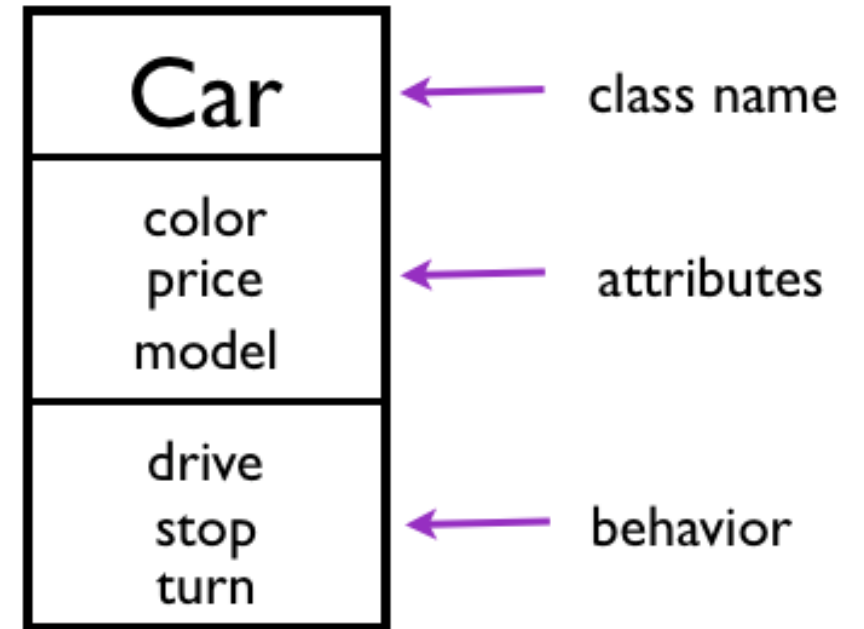


Class Car

- In python, we can create a class using the keyword `class`
- Method of class can be defined by keyword `def`
- The first parameter in the definition of a method is always `self` and method is called without the parameter `self`

- Create a class in Python

```
class Car:  
    # attributes ==> not recommend  
    color = "red"  
    price = 1500  
    model = "CX-5"  
  
    # methods  
    def drive(self):  
        pass
```

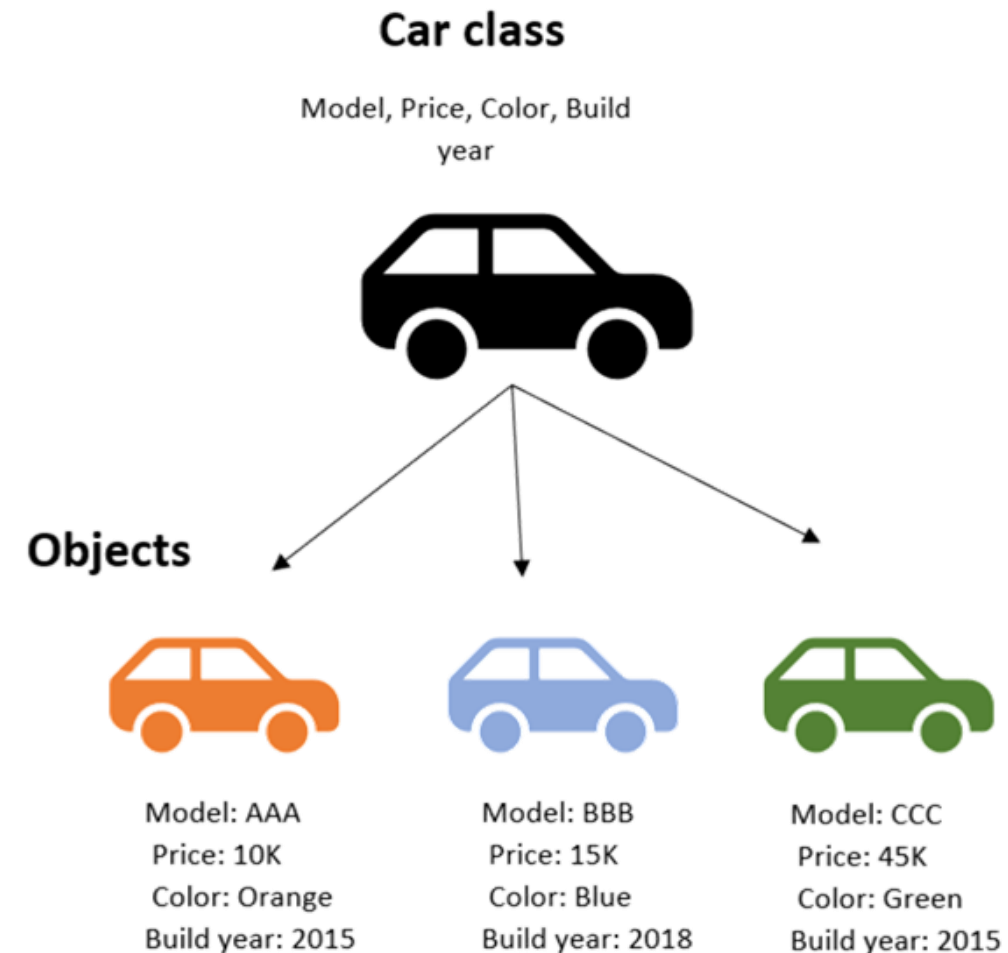


Class Car

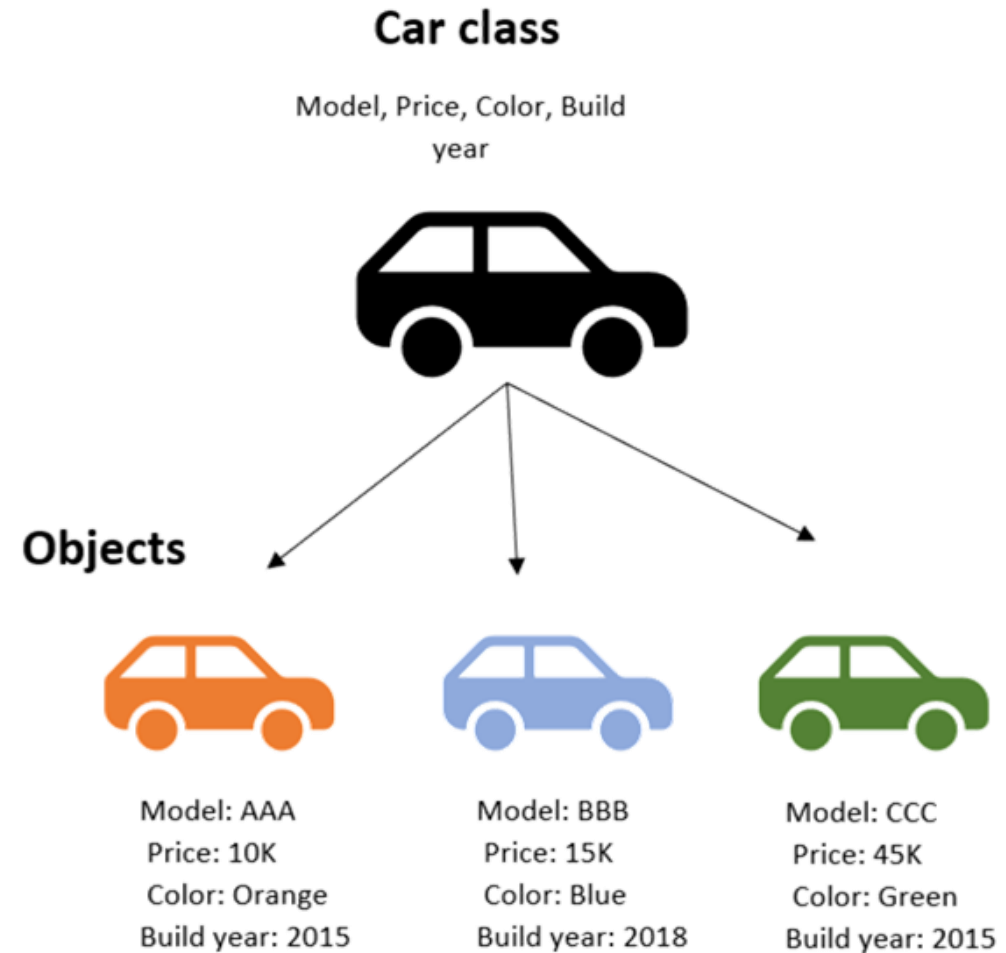
- All classes create objects, and all objects contain characteristics called attributes/properties
- Use the `__init__` method - initializer to instantiate objects
 - used to initialize an object's initial attributes by giving them their default value (or state)
 - This method takes one argument `self`, which refers to the object itself
- Instantiating Objects:
 - To instantiate an object, type the class name, followed by `()`
 - Assign this to a variable to keep track of the object

- How about the object
 - It is the specific instant from the class

```
car1 = Car("orange", 10000, "AAA")  
car2 = Car("blue", 15000, "BBB")  
car3 = Car("green", 45000, "CCC")
```



- Constructor is a special method
- You can think of as a function which initializes or activates the attributes or properties of the class for a object
- For example, you can think of constructor as entire sequence of actions required so that the factory constructs a car object out of the class design pattern



- Objects are instances of a class
- Words “instance” and “object” are used interchangeably
- The process of creating an object of a class is called **instantiation**
- In the following example, we are asking user to input values as the initial attributes of the specific car
- The methods `__init__` is called when ever an object of the class is constructed



Model: AAA
Price: 10K
Color: Orange
Build year: 2015



Model: BBB
Price: 15K
Color: Blue
Build year: 2018



Model: CCC
Price: 45K
Color: Green
Build year: 2015

```
class Car:
    # attributes
    __color = "red"
    __price = 1500
    __model = "CX-5"

    # Constructor
    def __init__(self, color, price, model):
        self.__model = model
        self.__color = color
        self.__price = price
```

```
car1 = Car("orange", 10000, "AAA")
car2 = Car("blue", 15000, "BBB")
car3 = Car("green", 45000, "CCC")
```



Model: AAA
Price: 10K
Color: Orange
Build year: 2015



Model: BBB
Price: 15K
Color: Blue
Build year: 2018



Model: CCC
Price: 45K
Color: Green
Build year: 2015

- The **self** parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class
- This handy keyword allows you to access variables, attributes, and methods of a defined class in Python
- The self parameter doesn't have to be named "self," as you can call it by any other name. However, the self parameter **must always be the first** parameter of any class function, regardless of the name chosen.

```
1  class Address:
2      def __init__(mine, street, number):
3          mine.street = street
4          mine.number = number
5
6      def myfunc(abc):
7          print("My Address is " + abc.street)
8
9  p1 = Address("Albert Street", 20)
10 p1.myfunc()
```

- You can add the initial attributes into class via `__init__` method

```
class Dog:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

- Classes are created by keyword `class`.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (`.`) operator

```
class Dog:
    # Class Attribute
    species = 'mammal'
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

An object consists of

- State: It is represented by the attributes of an object. It also reflects the properties of an object.
- Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- Identity: It gives a unique name to an object and enables one object to interact with other objects.

- An Object will call a method to modify its attribute and affect its work via dot operator: `.<name of method>`

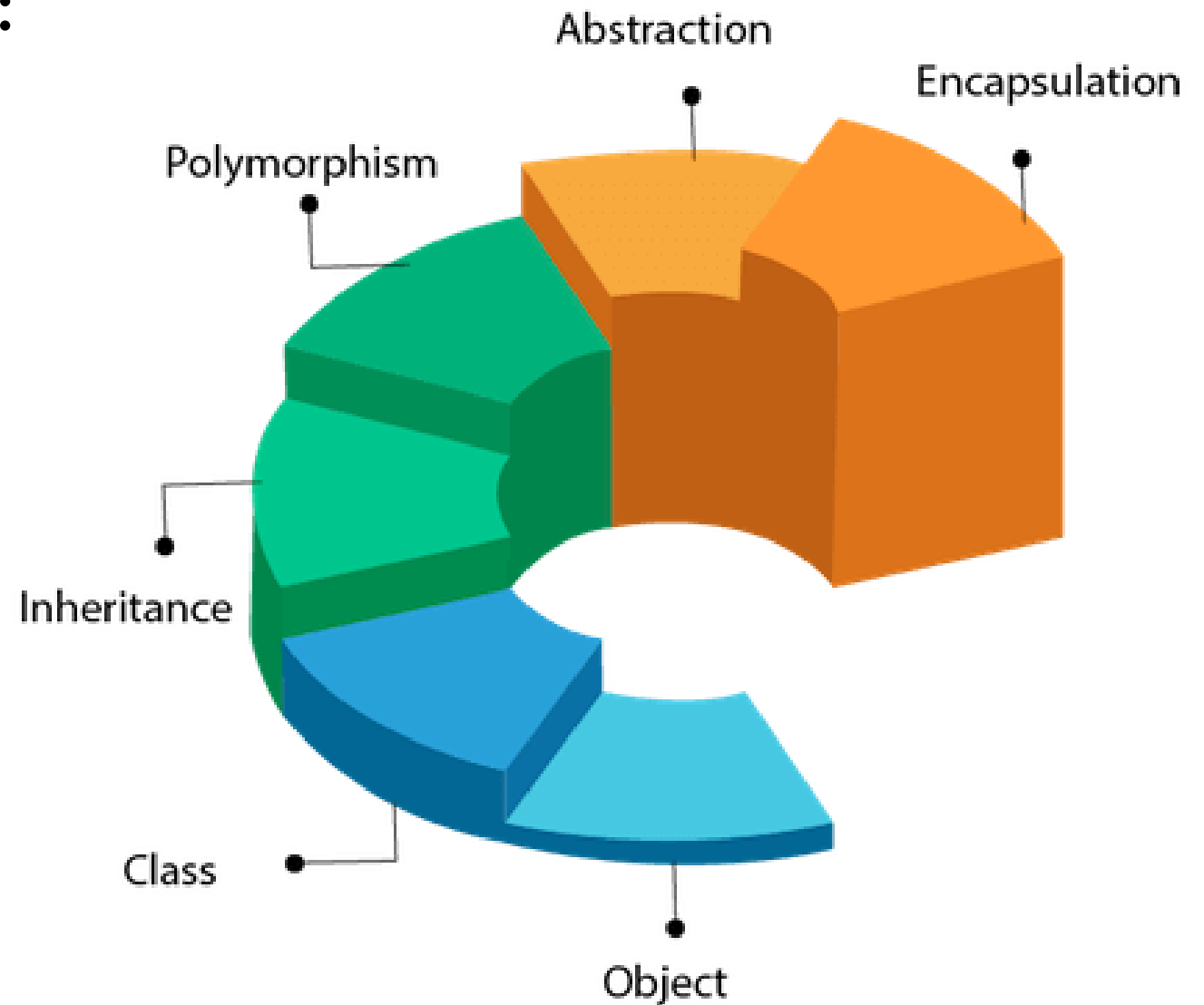
```
class Car:  
    # ...  
    def update_price(self, new_price):  
        self.__price = new_price
```

```
car1 = Car("orange", 10000, "AAA")  
car1.update_price(50000)
```


2. OOP Principles

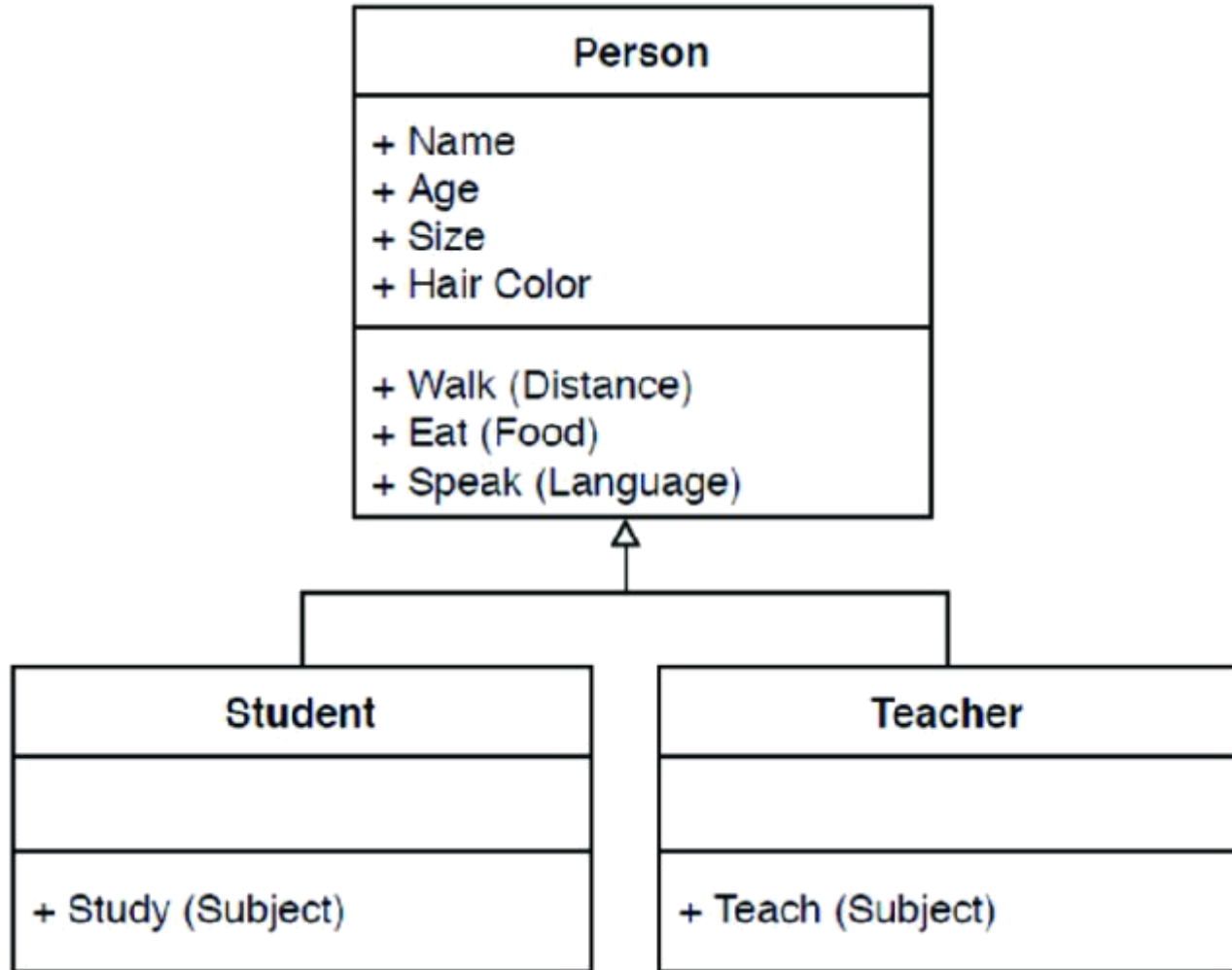
There are 4 major pillars of OOP:

- Encapsulation
- Abstraction
- Polymorphism
- Inheritance



- Inheritance is possible to create a new class that modifies the behavior of an existing class
- In an inheritance relationship
 - Classes that inherit from another are called derived classes, subclasses, or subtypes
 - Classes from which other classes are derived are called base classes or super classes
 - A derived class is said to derive, inherit, or extend a base class

- Child classes override or extend the functionality (e.g., attributes and behaviors) of parent classes
- In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow



- Child classes can also override attributes and behaviors from the parent class
- Overriding is when a child class creates a new implementation of an inherited method

```
class Shape():  
    def area(self):  
        return 0
```

```
class Square(Shape):  
    def __init__(self, length):  
        self.length = length  
  
    def area(self):  
        return self.length * self.length
```

- In Python, we can change the way operators work for user-defined types
- For example, the **+** operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings
- Here are some of the special functions available in Python,

Function	Description
<code>__init__()</code>	initialize the attributes of the object
<code>__str__()</code>	returns a string representation of the object
<code>__len__()</code>	returns the length of the object
<code>__add__()</code>	adds two objects
<code>__call__()</code>	call objects of the class like a normal function

- To overload the + operator, we will need to implement `__add__()` function in the class

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
```

```
p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)

# Output: (3,5)
```

- Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below

Operator	Expression	Internally
Addition	<code>p1 + p2</code>	<code>p1.__add__(p2)</code>
Subtraction	<code>p1 - p2</code>	<code>p1.__sub__(p2)</code>
Multiplication	<code>p1 * p2</code>	<code>p1.__mul__(p2)</code>
Power	<code>p1 ** p2</code>	<code>p1.__pow__(p2)</code>
Division	<code>p1 / p2</code>	<code>p1.__truediv__(p2)</code>
Floor Division	<code>p1 // p2</code>	<code>p1.__floordiv__(p2)</code>
Remainder (modulo)	<code>p1 % p2</code>	<code>p1.__mod__(p2)</code>
Bitwise Left Shift	<code>p1 << p2</code>	<code>p1.__lshift__(p2)</code>
Bitwise Right Shift	<code>p1 >> p2</code>	<code>p1.__rshift__(p2)</code>
Bitwise AND	<code>p1 & p2</code>	<code>p1.__and__(p2)</code>
Bitwise OR	<code>p1 p2</code>	<code>p1.__or__(p2)</code>
Bitwise XOR	<code>p1 ^ p2</code>	<code>p1.__xor__(p2)</code>
Bitwise NOT	<code>~p1</code>	<code>p1.__invert__()</code>

- Encapsulation is achieved by declaring a class's data members and methods as either private or protected
- In Python, there is no keywords like '**public**', '**protected**' and '**private**' to define the accessibility
- In other words, in Python, it acquiesce that all attributes are public
- But there is a method in Python to define Private:
 - Add " " in front of the variable and function name can hide them when accessing them from out of class

- **Public Member:** Accessible anywhere from outside the class.
- **Private Member:** Accessible within the class.
- **Protected Member:** Accessible within the class and its sub-classes

```
class Employee:
    # constructor
    def __init__(self, name, id, salary, project):
        # data members
        #Public (accessible from outside and inside the class)
        self.name = name
        self.id = id
        #Protected (accessible within the class and its subclass)
        self._project = project
        #Private (accessible only inside
        #the class it is declared)
        self.__salary = salary
```

```
a = Employee("John", "ID01", "DSA", 5000)
print(a.name, a.id)
print(a._project)
print(a.__salary)
```

- Important advantage of OOP consists in the encapsulation of data. We can say that object-oriented programming relies heavily on encapsulation.
- The terms encapsulation and abstraction (also data hiding) are often used as synonyms. They are nearly synonymous, i.e. abstraction is achieved through encapsulation.
- Data hiding and encapsulation are the same concept, so it's correct to use them as synonyms
- Generally speaking encapsulation is the mechanism for restricting the access to some of an object's components, this means, that the internal representation of an object can't be seen from outside of the object's definition

- We can access and modify private attributes with class getters and setters defined as class methods

```
class Employee:
    def __init__(self, name, id, salary, project):
        # data members
        self.name = name
        self.id = id
        self._project = project
        #Private
        self.__salary = salary
```

```
def get_salary(self):
    return self.__salary
def set_salary(self, new_salary):
    self.__salary = new_salary
```

```
a = Employee("John", "ID01", "DSA", 5000)
print(a.name, a.id)
print(a._project)
print(a.get_salary()) # 5000
a.set_salary(500)
print(a.get_salary()) # 500
```

- In object-oriented programming, methods which are dedicated to accessing and changing private attributes are usually called getters and setters
- Not all Python programmers use the terms "getter" and "setter", but the concept of properties outlined below is very similar
- So, above we created some public methods for accessing private attributes, but there is a more straightforward, "pythonic" way of accessing attributes
- the getter method, i.e. the @property

```
class Wallet:
    def __init__(self):
        self.__money = 0

    # A getter method
    @property
    def money(self):
        return self.__money

    # A setter method
    @money.setter
    def money(self, money):
        if money >= 0:
            self.__money = money
```

```
class Wallet:
    def __init__(self):
        self.__money = 0

    # A getter method
    @property
    def money(self):
        return self.__money

    # A setter method
    @money.setter
    def money(self, money):
        if money >= 0:
            self.__money = money
```

```
wallet = Wallet()
print(wallet.money)
```

```
wallet.money = 50
print(wallet.money)
```

```
wallet.money = -30
print(wallet.money)
```

```
0
50
50
```

- the getter method, i.e. the @property decorator, must be introduced **before** the setter method, or there will be an error when the class is executed

- Python doesn't block access to private data entirely. It just leaves it to the wisdom of the programmer, not to write any code that accesses it from outside the class.
- Name mangling is the process of changing name of a member with double underscore to the form **object._class__variable**.

```
class Wallet:
    def __init__(self):
        self.__money = 50

    def setMoney(self, new_money):
        self.__money = new_money

wallet = Wallet()
print(wallet.money) # Error
print(wallet._Wallet__money) # 50
wallet.setMoney(100)
print(wallet._Wallet__money) # 100
```

- **Security:** The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
- **Data Hiding:** The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
- **Simplicity:** It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
- **Aesthetics:** Bundling data and methods within a class makes code more readable and maintainable

- Polymorphism is a fundamental concept in object-oriented programming
- This term refers to the ability of different classes to be treated as instances of the same class through inheritance
- In Python, polymorphism is implemented through various mechanisms, including duck typing, operator overloading, and method overriding

- **Duck Typing:** Duck typing is a concept in Python that focuses on the behavior of objects rather than their type or class. It allows you to work with objects based on their capabilities or methods, rather than explicitly checking their types. If an object behaves like a duck, it's treated as a duck

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

def animal_sound(animal):
    return animal.speak()

# Usage of duck typing
dog = Dog()
cat = Cat()
print(animal_sound(dog))
print(animal_sound(cat))
```

- **Operator overloading**
allows you to define how operators like `+`, `-`, `*`, etc., behave when applied to objects of your custom classes. This is achieved by defining special methods in your class, such as `__add__`, `__sub__`, `__mul__`, etc.

```
class ComplexNumber:
    def __init__(self, real, image):
        self.real = real
        self.imag = image

    def __add__(self, other):
        return ComplexNumber(
            self.real + other.real,
            self.imag + other.imag)

    def __str__(self):
        return f"{self.real} + {self.imag}i"

# Usage of operator overloading
num1 = ComplexNumber(2, 3)
num2 = ComplexNumber(1, 4)
result = num1 + num2 # 3 + 7i
```

- **Method Overriding:** Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is a form of polymorphism where a subclass can replace or extend the behavior of its superclass's method

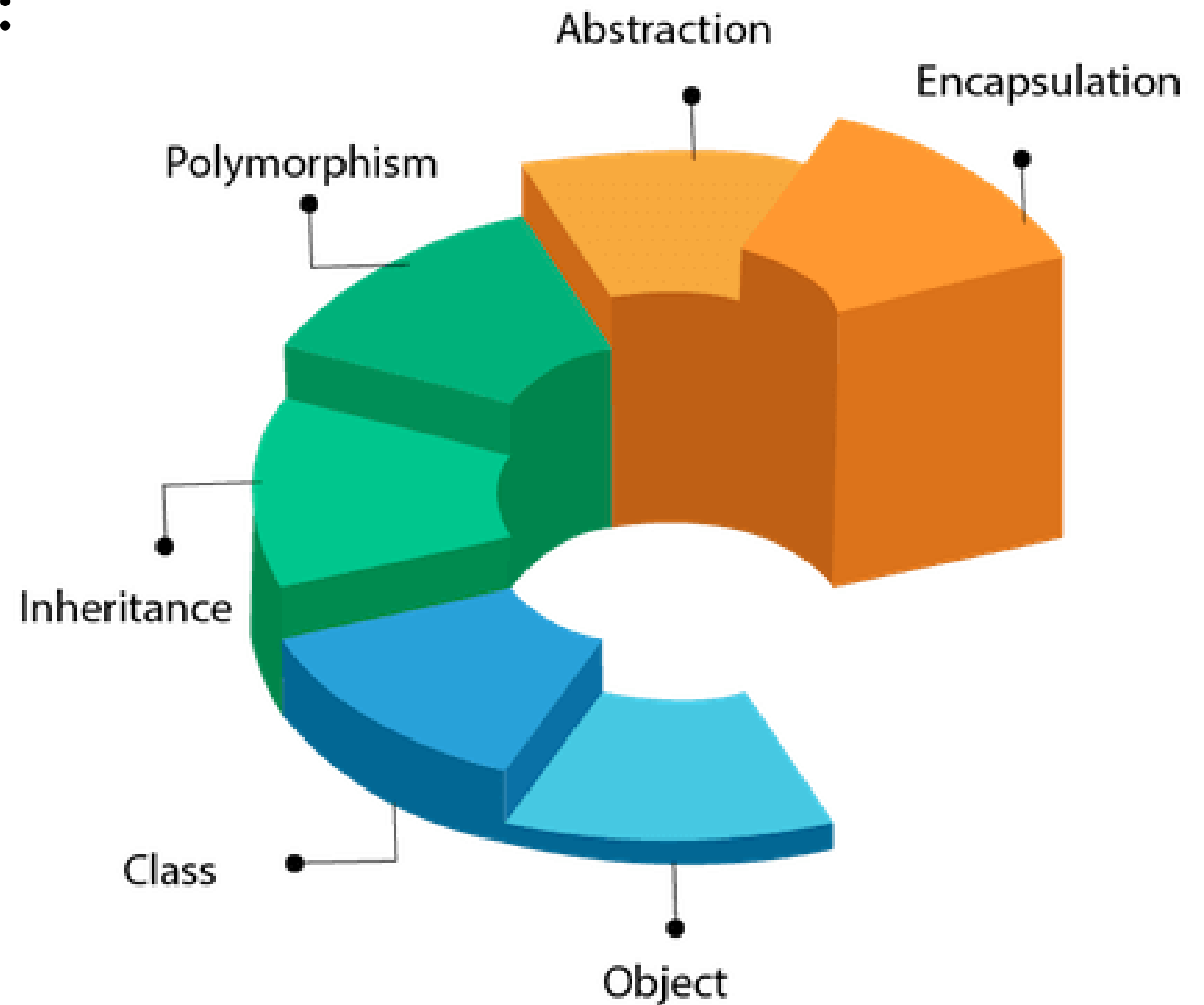
```
class Animal:
    def speak(self):
        return "Can do something"

class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"

# Usage of method overriding
dog = Dog()
cat = Cat()
print(dog.speak())
print(cat.speak())
```

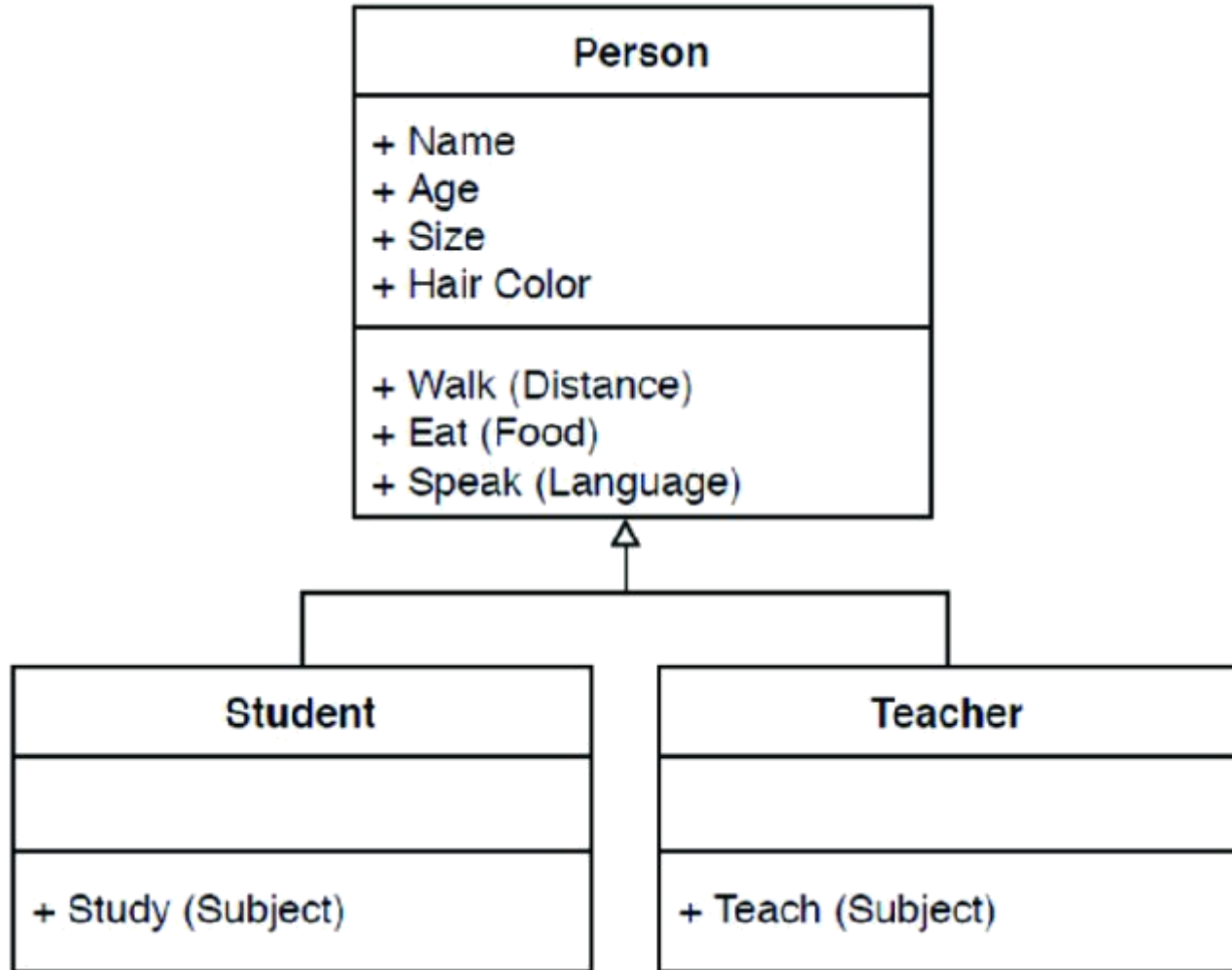
There are 4 major pillars of OOP:

- Encapsulation
- Abstraction
- Polymorphism
- Inheritance



- Inheritance is possible to create a new class that modifies the behavior of an existing class
- In an inheritance relationship
 - Classes that inherit from another are called derived classes, subclasses, or subtypes
 - Classes from which other classes are derived are called base classes or super classes
 - A derived class is said to derive, inherit, or extend a base class

- Child classes override or extend the functionality (e.g., attributes and behaviors) of parent classes
- In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow



- Child classes can also override attributes and behaviors from the parent class
- Overriding is when a child class creates a new implementation of an inherited method

```
class Shape():  
    def area(self):  
        return 0
```

```
class Square(Shape):  
    def __init__(self, length):  
        self.length = length  
  
    def area(self):  
        return self.length * self.length
```


- In Python, we can change the way operators work for user-defined types
- For example, the **+** operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings
- Here are some of the special functions available in Python,

Function	Description
<code>__init__()</code>	initialize the attributes of the object
<code>__str__()</code>	returns a string representation of the object
<code>__len__()</code>	returns the length of the object
<code>__add__()</code>	adds two objects
<code>__call__()</code>	call objects of the class like a normal function

- To overload the + operator, we will need to implement `__add__()` function in the class

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
```

```
p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)

# Output: (3,5)
```

- Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below

Operator	Expression	Internally
Addition	<code>p1 + p2</code>	<code>p1.__add__(p2)</code>
Subtraction	<code>p1 - p2</code>	<code>p1.__sub__(p2)</code>
Multiplication	<code>p1 * p2</code>	<code>p1.__mul__(p2)</code>
Power	<code>p1 ** p2</code>	<code>p1.__pow__(p2)</code>
Division	<code>p1 / p2</code>	<code>p1.__truediv__(p2)</code>
Floor Division	<code>p1 // p2</code>	<code>p1.__floordiv__(p2)</code>
Remainder (modulo)	<code>p1 % p2</code>	<code>p1.__mod__(p2)</code>
Bitwise Left Shift	<code>p1 << p2</code>	<code>p1.__lshift__(p2)</code>
Bitwise Right Shift	<code>p1 >> p2</code>	<code>p1.__rshift__(p2)</code>
Bitwise AND	<code>p1 & p2</code>	<code>p1.__and__(p2)</code>
Bitwise OR	<code>p1 p2</code>	<code>p1.__or__(p2)</code>
Bitwise XOR	<code>p1 ^ p2</code>	<code>p1.__xor__(p2)</code>
Bitwise NOT	<code>~p1</code>	<code>p1.__invert__()</code>

- Encapsulation is achieved by declaring a class's data members and methods as either private or protected
- In Python, there is no keywords like '**public**', '**protected**' and '**private**' to define the accessibility
- In other words, in Python, it acquiesce that all attributes are public
- But there is a method in Python to define Private:
 - Add " " in front of the variable and function name can hide them when accessing them from out of class

- **Public Member:** Accessible anywhere from outside the class.
- **Private Member:** Accessible within the class.
- **Protected Member:** Accessible within the class and its sub-classes

```
class Employee:
    # constructor
    def __init__(self, name, id, salary, project):
        # data members
        #Public (accessible from outside and inside the class)
        self.name = name
        self.id = id
        #Protected (accessible within the class and its subclass)
        self._project = project
        #Private (accessible only inside
        #the class it is declared)
        self.__salary = salary
```

```
a = Employee("John", "ID01", "DSA", 5000)
print(a.name, a.id)
print(a._project)
print(a.__salary)
```

- Important advantage of OOP consists in the encapsulation of data. We can say that object-oriented programming relies heavily on encapsulation.
- The terms encapsulation and abstraction (also data hiding) are often used as synonyms. They are nearly synonymous, i.e. abstraction is achieved through encapsulation.
- Data hiding and encapsulation are the same concept, so it's correct to use them as synonyms
- Generally speaking encapsulation is the mechanism for restricting the access to some of an object's components, this means, that the internal representation of an object can't be seen from outside of the object's definition

- We can access and modify private attributes with class getters and setters defined as class methods

```
class Employee:
    def __init__(self, name, id, salary, project):
        # data members
        self.name = name
        self.id = id
        self._project = project
        #Private
        self.__salary = salary
```

```
def get_salary(self):
    return self.__salary
def set_salary(self, new_salary):
    self.__salary = new_salary
```

```
a = Employee("John", "ID01", "DSA", 5000)
print(a.name, a.id)
print(a._project)
print(a.get_salary()) # 5000
a.set_salary(500)
print(a.get_salary()) # 500
```

- In object-oriented programming, methods which are dedicated to accessing and changing private attributes are usually called getters and setters
- Not all Python programmers use the terms "getter" and "setter", but the concept of properties outlined below is very similar
- So, above we created some public methods for accessing private attributes, but there is a more straightforward, "pythonic" way of accessing attributes
- the getter method, i.e. the @property

```
class Wallet:
    def __init__(self):
        self.__money = 0

    # A getter method
    @property
    def money(self):
        return self.__money

    # A setter method
    @money.setter
    def money(self, money):
        if money >= 0:
            self.__money = money
```



```
class Wallet:
    def __init__(self):
        self.__money = 0

    # A getter method
    @property
    def money(self):
        return self.__money

    # A setter method
    @money.setter
    def money(self, money):
        if money >= 0:
            self.__money = money
```

```
wallet = Wallet()
print(wallet.money)
```

```
wallet.money = 50
print(wallet.money)
```

```
wallet.money = -30
print(wallet.money)
```

```
0
50
50
```

- the getter method, i.e. the @property decorator, must be introduced **before** the setter method, or there will be an error when the class is executed

- Python doesn't block access to private data entirely. It just leaves it to the wisdom of the programmer, not to write any code that accesses it from outside the class.
- Name mangling is the process of changing name of a member with double underscore to the form **object._class__variable**.

```
class Wallet:
    def __init__(self):
        self.__money = 50

    def setMoney(self, new_money):
        self.__money = new_money

wallet = Wallet()
print(wallet.money) # Error
print(wallet._Wallet__money) # 50
wallet.setMoney(100)
print(wallet._Wallet__money) # 100
```

- **Security:** The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
- **Data Hiding:** The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
- **Simplicity:** It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
- **Aesthetics:** Bundling data and methods within a class makes code more readable and maintainable

- Polymorphism is a fundamental concept in object-oriented programming
- This term refers to the ability of different classes to be treated as instances of the same class through inheritance
- In Python, polymorphism is implemented through various mechanisms, including duck typing, operator overloading, and method overriding

- **Duck Typing:** Duck typing is a concept in Python that focuses on the behavior of objects rather than their type or class. It allows you to work with objects based on their capabilities or methods, rather than explicitly checking their types. If an object behaves like a duck, it's treated as a duck

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

def animal_sound(animal):
    return animal.speak()

# Usage of duck typing
dog = Dog()
cat = Cat()
print(animal_sound(dog))
print(animal_sound(cat))
```

- **Operator overloading**
allows you to define how operators like `+`, `-`, `*`, etc., behave when applied to objects of your custom classes. This is achieved by defining special methods in your class, such as `__add__`, `__sub__`, `__mul__`, etc.

```
class ComplexNumber:
    def __init__(self, real, image):
        self.real = real
        self.imag = image

    def __add__(self, other):
        return ComplexNumber(
            self.real + other.real,
            self.imag + other.imag)

    def __str__(self):
        return f"{self.real} + {self.imag}i"

# Usage of operator overloading
num1 = ComplexNumber(2, 3)
num2 = ComplexNumber(1, 4)
result = num1 + num2 # 3 + 7i
```

- **Method Overriding:** Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is a form of polymorphism where a subclass can replace or extend the behavior of its superclass's method

```
class Animal:
    def speak(self):
        return "Can do something"

class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"

# Usage of method overriding
dog = Dog()
cat = Cat()
print(dog.speak())
print(cat.speak())
```

- You are building a **library system** that allows users to manage books. The system stores book information, such as the title, author, and quantity, and saves this data in a **JSON file**. Additionally, the system must load data from the JSON file when it starts, ensuring that all book information persists between runs.
- **Class Book:** Stores the title, author, and quantity of each book.
- **Class Library:** Manages a collection of books and supports the following actions:
 - **add_book(book):** Add a new book to the library.
 - **remove_book(title):** Remove a book by title.
 - **save_to_json(filename):** Save the library data to a JSON file.
 - **load_from_json(filename):** Load library data from a JSON file.


```
[
  {
    "title": "1984",
    "author": "George Orwell",
    "quantity": 5
  },
  {
    "title": "Moby Dick",
    "author": "Herman Melville",
    "quantity": 3
  },
  {
    "title": "The Great Gatsby",
    "author": "F. Scott Fitzgerald",
    "quantity": 4
  }
]
```

THANK YOU
for YOUR ATTENTION