CS112 Introduction to Python Programming

Session 13: Classes and Objects

Shengwei Hou
Ph.D., Assistant Professor
Department of Ocean Science and Engineering

Fall 2022

Contents

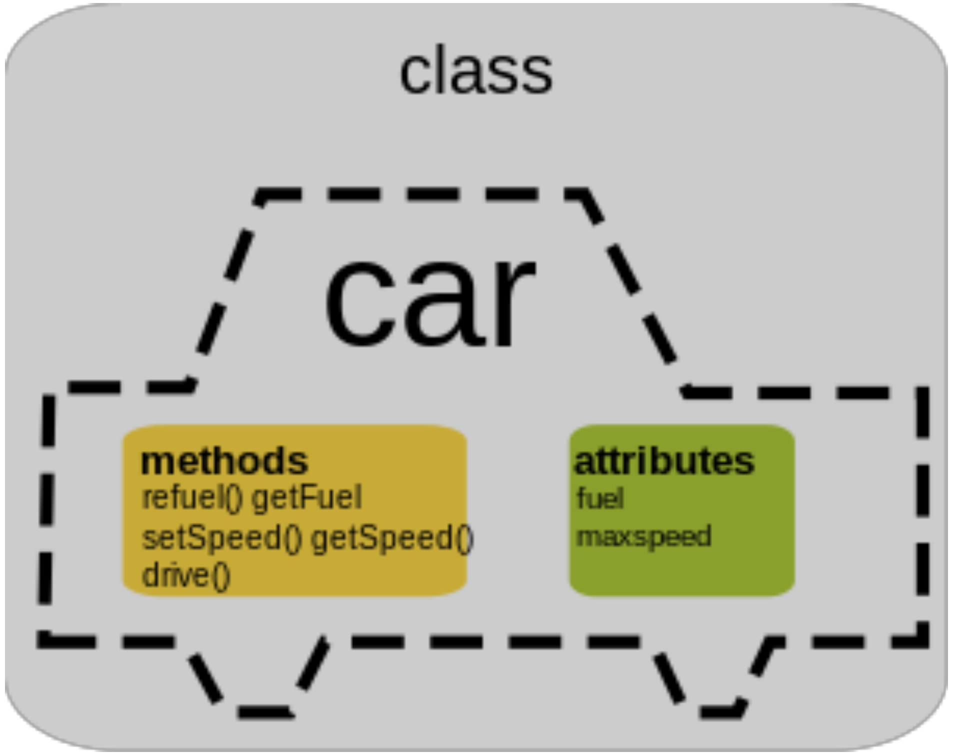
- Introduction
- The `__init__()` function
- Instance and Class variables
- The `self` parameter
- The `__str__()` function
- Instance, Class and Static Class methods
- Constructor and destructor
- Inheritance
- Encapsulation
- Polymorphism

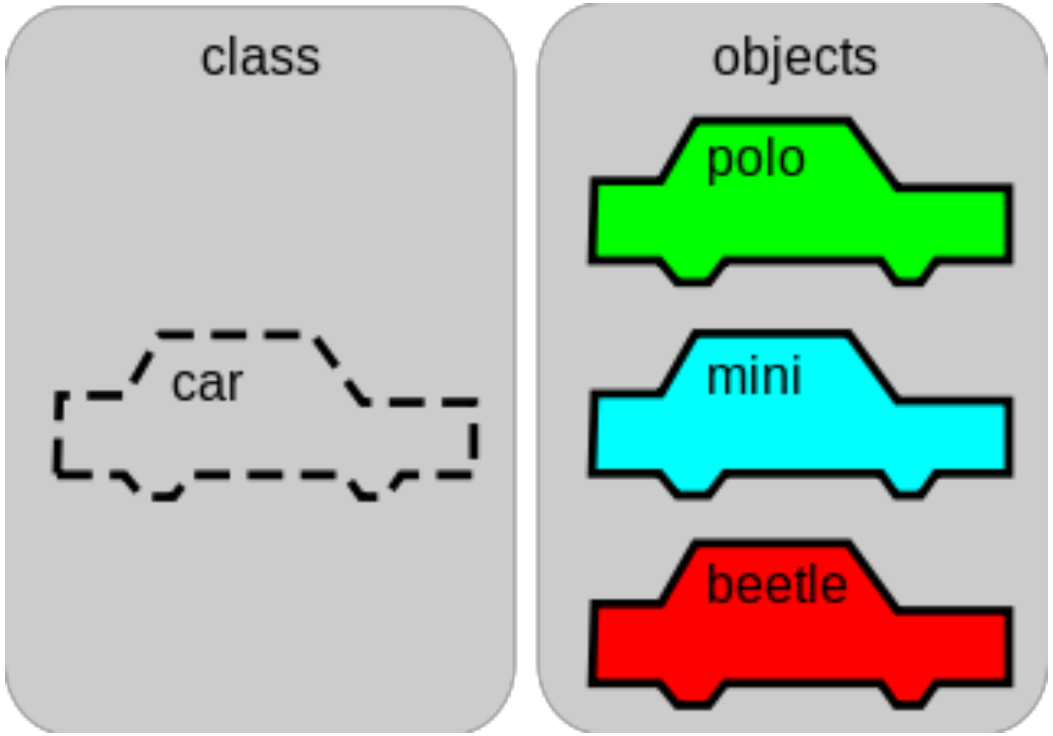
Introduction

- **Object-oriented programming (OOP)** is a programming paradigm based on the concept of “objects”, which can contain data and code.
  - The **data** is in the form of **fields** (often known as **attributes** or **properties**)
  - The **code** is in the form of **procedures** (often known as **methods**).

- Python is an OOP language. Almost everything in Python is an object, with its properties and methods.

Introduction

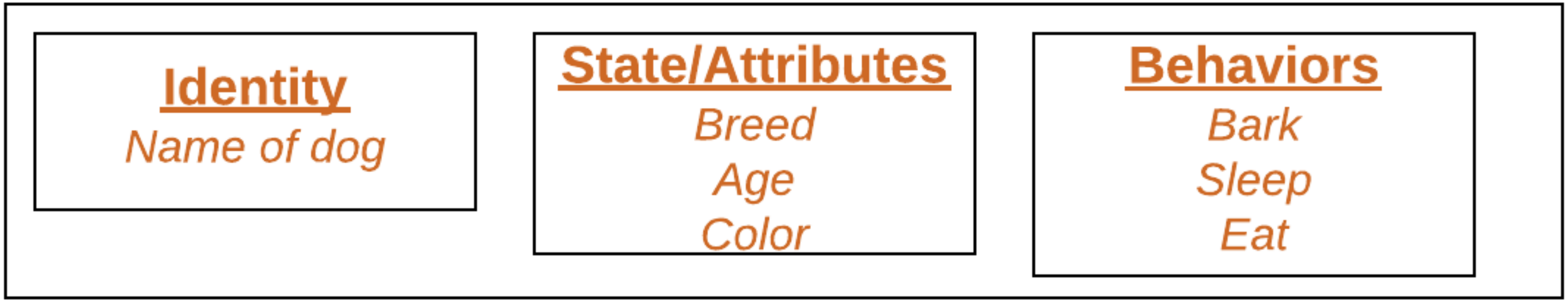




A **Class** is like an object constructor, or a "blueprint" for creating objects.

An **Object** is an instance of a Class. An Object is a copy of the class with *actual values*.

Introduction



Identity
Name of dog

State/Attributes
Breed
Age
Color

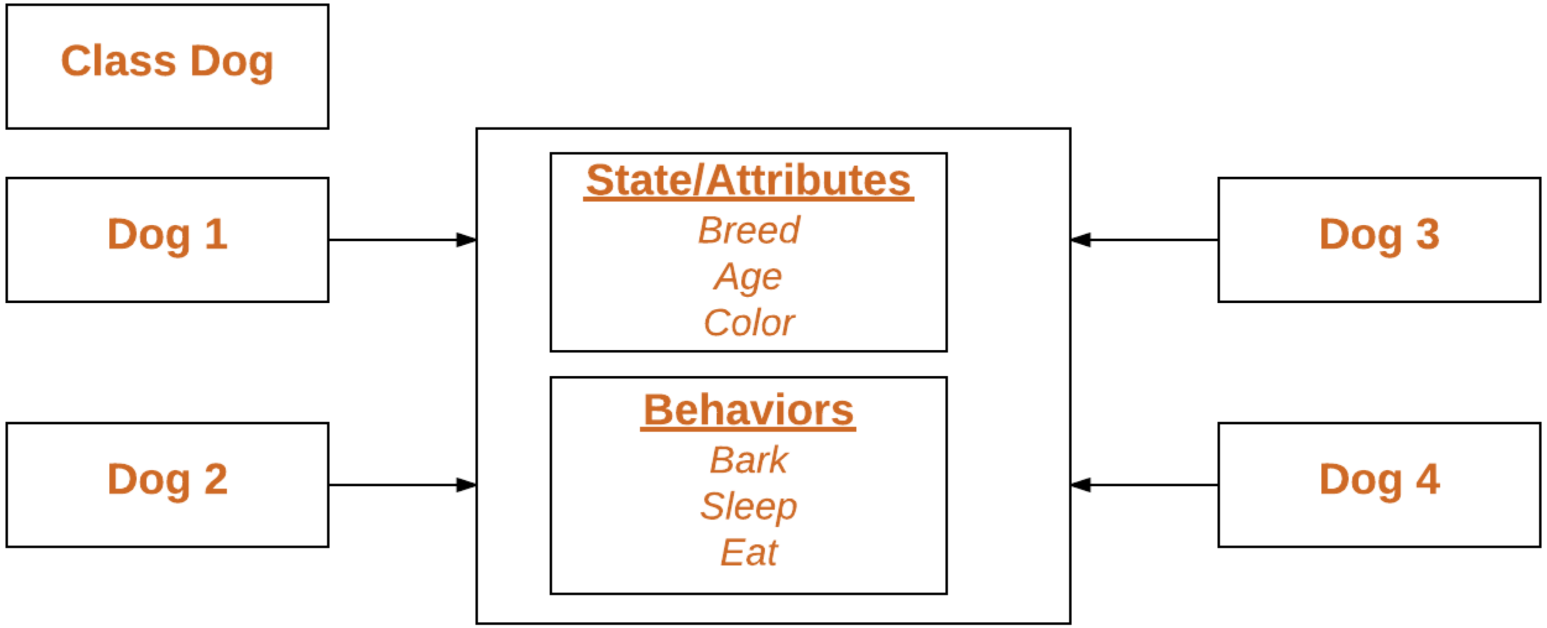
Behaviors
Bark
Sleep
Eat

An object consists of :

- **Identity:** It gives a unique ~~name~~ to an object and enables one object to interact with other objects.
- **States/Attributes:** These reflect the properties of an object.
- **Behaviors:** They are represented by the **methods** of an object, or the responses of an object to other objects.

https://www.geeksforgeeks.org/python-classes-and-objects/

Introduction



Declaring Objects (also called instantiating a class)

- All the instances share the attributes and the behavior of the class.
- But the values of those attributes are unique for each object.
- A single class may have any number of instances.

https://www.geeksforgeeks.org/python-classes-and-objects/

Introduction

- Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

- Now we can use the class named MyClass to create objects:
  - Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
print(type(p1))
print(type(p1.x))
```

```
5
<class '__main__.MyClass'>
<class 'int'>
```

https://www.w3schools.com/python/python_classes.asp

The `__init__()` Function

- All classes have a function called `__init__()`, which is always executed when the class is being initiated.
- The `__init__()` function is commonly used to assign values to object properties, along with other operations that are necessary when creating objects:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

```
John
36
```

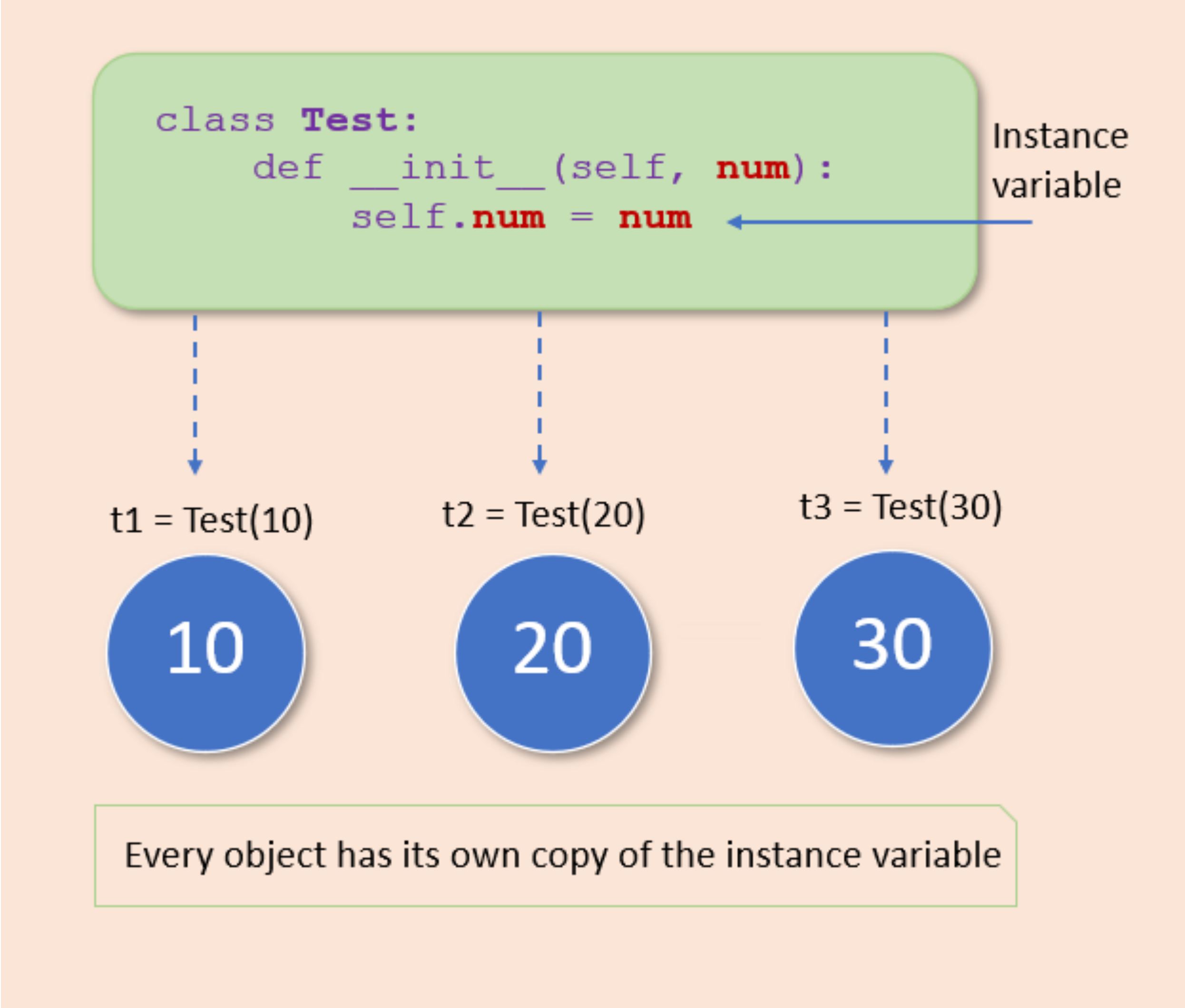
https://www.w3schools.com/python/python_classes.asp

Instance variables

- We can access the instance variable using the object and dot (.) operator.

```
>>> print(t1.num + t2.num)
```

- In Python, to work with an instance variable and method, we use the `self` keyword. We use the `self` keyword as the first parameter to a method. The `self` refers to the current object.



```
class Test:
    def __init__(self, num):
        self.num = num
```

Every object has its own copy of the instance variable

https://pynative.com/python-classes-and-objects/

The `self` `parameter`

- Instance methods must have an extra first parameter (usually named as `self`) in the method definition.
- We do not give a value for this parameter when we call the method, Python provides it.
- If we have a method that takes no arguments, we still have one argument, the instance itself (`self`).
- When we call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)`

https://www.geeksforgeeks.org/python-classes-and-objects/

The `self` parameter

- It does not have to be named `self`, you can call it whatever you like, but it has to be the first parameter of any function in the class.

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

OUTPUT

```
Hello my name is John
```

https://www.w3schools.com/python/python_classes.asp

Modify instance variables

```
class Student:
    def __init__(self, name, age):
    # Instance variable
        self.name = name
        self.age = age

# create object
stud = Student("Jessa", 20)

print('Before')
print('Name:', stud.name, 'Age:', stud.age)

# modify instance variable
stud.name = 'Emma'
stud.age = 15

print('After')
print('Name:', stud.name, 'Age:', stud.age)
```

OUTPUT

```
Before
Name: Jessa Age: 20
After
Name: Emma Age: 15
```

https://pynative.com/python-class-variables/

Access instance variables

```
class Student:
   def __init__(self, name, age):
   # Instance variable
      self.name = name
      self.age = age

# create object
stud = Student("Jessa", 20)

# Use the dot
print('Name:', stud.name)

# Use getattr instead of the dot
print('Name:', ~~getattr~~(stud, 'name'))
```

OUTPUT

```
Name: Jessa
Name: Jessa
```

`getattr(object, name[, default]) -> value`

Get a named attribute from an object; `getattr(x, 'y')` is equivalent to `x.y`.
When a default argument is given, it is returned when the attribute doesn’t exist, otherwise an exception will be raised.

https://pynative.com/python-class-variables/

Dynamically add instance variables

```
class Student:
    def __init__(self, name, age):
    # Instance variable
        self.name = name
        self.age = age

# create object
stud = Student("Jessa", 20)

print('Before')
print('Name:', stud.name, 'Age:', stud.age)

# add new instance variable 'marks' to stud
stud.marks = 75

print('After')
print('Name:', stud.name, 'Age:', stud.age,
'Marks:', stud.marks)
```

OUTPUT

```
Before
Name: Jessa Age: 20
After
Name: Jessa Age: 20 Marks:
75
```

https://pynative.com/python-class-variables/

Dynamically delete instance variables

```python
class Student:
    def __init__(self, name, age):
    # Instance variable
        self.name = name
        self.age = age

# create object
stud = Student("Jessa", 20)

print('Before')
print('Name:', stud.name, 'Age:', stud.age)

# delete instance variable
del stud.age

print('After')
print('Name:', stud.name, 'Age:', stud.age)
```

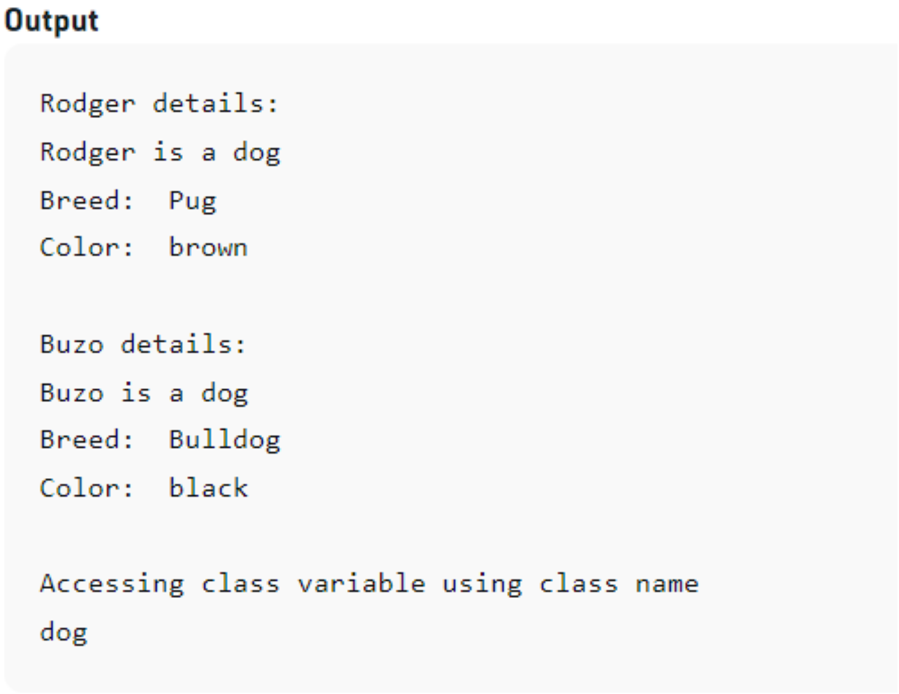
OUTPUT

```
Before
Name: Jessa Age: 20
After
Traceback (most recent call last):
  File "/Users/shengwei/test.py",
line 18, in <module>
    print('Name:', stud.name,
'Age:', stud.age)
AttributeError: 'Student' object
has no attribute 'age'
```

https://pynative.com/python-class-variables/

Class variables

```python
class Dog:
    # Class Variable
    animal = 'dog'

    def __init__(self, breed, color):
        # Instance Variable
        self.breed = breed
        self.color = color
```



Output

```
Rodger details:
Rodger is a dog
Breed:   Pug
Color:   brown

Buzo details:
Buzo is a dog
Breed:   Bulldog
Color:   black

Accessing class variable using class name
dog
```

```python
# Objects of Dog class
Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")
print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)

# Class variables can be accessed using class
# name also
print("\nAccessing class variable using class name")
print(Dog.animal)
```

https://www.geeksforgeeks.org/python-classes-and-objects/

Modify class variables

```python
class Pet:
    # Class Variable
    animal = 'cat'

    def __init__(self, name):
        # Instance Variable
        self.name = name

# create Objects
p1 = Pet('Momo')
p2 = Pet('Stella')
print(p1.animal, p2.animal, Pet.animal)

Pet.animal = 'dog'
print(p1.animal, p2.animal, Pet.animal)

p1.animal = 'cat'
print(p1.animal, p2.animal, Pet.animal)
```

- It is best practice to **use a class name to change the value of a class variable.**

- Because if we try to change the class variable’s value by using an object, a new instance variable is created for that particular object, which shadows the class variables.

OUTPUT

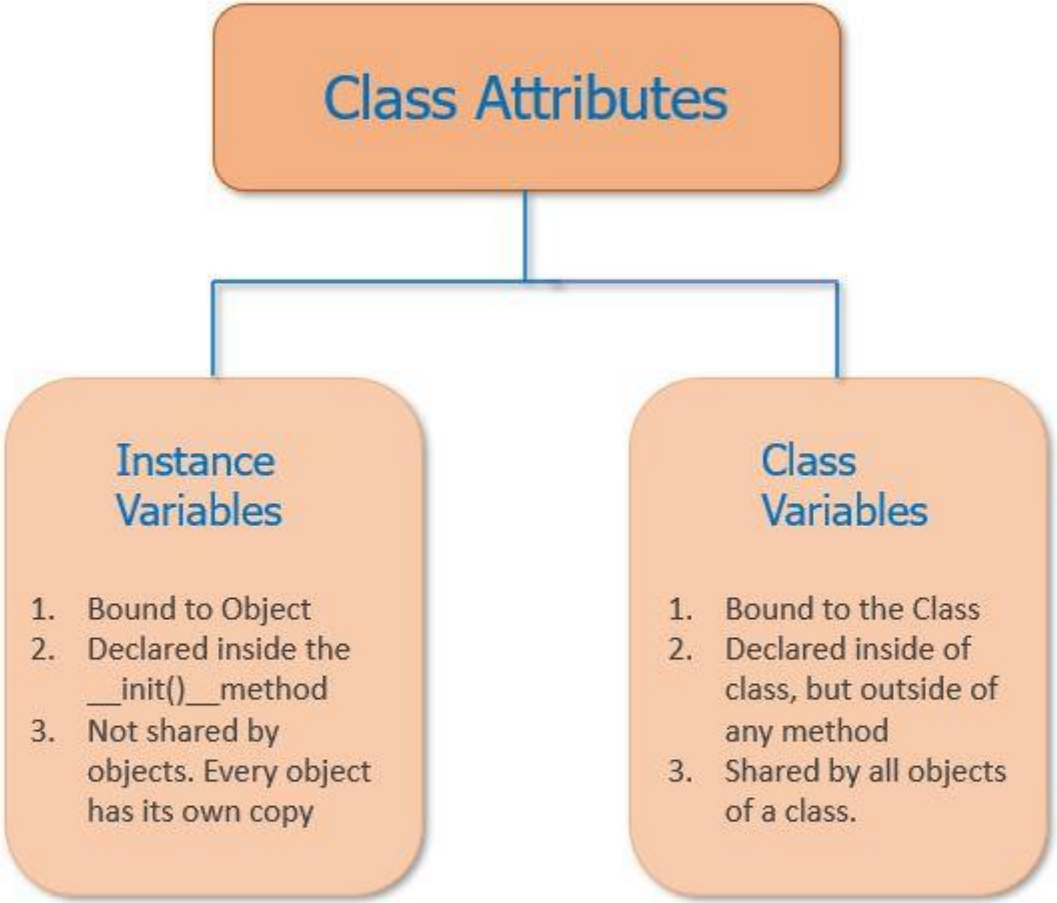
```
cat cat cat
dog dog dog
cat dog dog
```

https://pynative.com/python-class-variables/

Instance and Class variables

In Class, attributes can be defined into two parts:

- **Instance variables**: The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor (the `__init__()` method of a class).
- **Class Variables**: A class variable is a variable that is declared inside of a class, but outside of any instance method or the `__init__()` method. Class variables are shared by all instances of this class, and the class object.



Class Attributes

Instance Variables

1. Bound to Object
2. Declared inside the __init()__method
3. Not shared by objects. Every object has its own copy

Class Variables

1. Bound to the Class
2. Declared inside of class, but outside of any method
3. Shared by all objects of a class.

https://pynative.com/python-classes-and-objects/

The `__str__()` Function

- The `__str__()` function controls what should be returned when the class object is represented as a string.
- If the `__str__()` function is not set, the string representation of the object (defined by `__repr__()`) is returned

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

p1 = Person("John", 36)
print(p1)
print(repr(p1))
```

OUTPUT

```
<__main__.Person object at
0x2ae2083ab100>
<__main__.Person object at
0x2ae2083ab100>
```

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def __str__(self):
    return f"{self.name}({self.age})"
 def __repr__(self):
    return "Person(name={},
age={})”.format(self.name, self.age)

p1 = Person("John", 36)
print(p1)
print(str(p1))
print(repr(p1))
```

```
John(36)
John(36)
Person(name=John, age=36)
```

Instance methods

- Objects (instances) can also contain methods.
- Instance methods are functions that belong to the object.
- Used to access or modify the object state.
- Use instance variables inside a method.
- Must have a `self` parameter to refer to the current object.

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

OUTPUT

```
Hello my name is John
```

https://www.w3schools.com/python/python_classes.asp
https://pynative.com/python-instance-methods/

Instance methods

```python
class Student:
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age

    # inst. method to modify inst. var.
    def update_age(self, age):
        self.age = age

    # inst. method to add inst. var.
    def add_marks(self, marks):
        self.marks = marks
```

```python
# create object
stud = Student("Emma", 14)
print(stud.name, stud.age)

# call instance method
stud.update_age(18)
stud.add_marks(75)
print(stud.name, stud.age, stud.marks)
```

OUTPUT

```
Emma 14
Emma 18 75
```

https://www.w3schools.com/python/python_classes.asp
https://pynative.com/python-instance-methods/

Class methods

- A **class method** is bound to the class and ~~not the object of the class~~. It can access only class variables.
- They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
- It can modify a class state that would apply across all the instances of the class. For example, it can modify a class variable that will be applicable to all the instances.

https://www.w3schools.com/python/python_classes.asp

Class methods

- The `@classmethod` is a built-in function ***decorator*** that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition.
- A class method receives the class as an implicit first argument, just like an instance method receives the instance
- Syntax for creating a class method:

Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behavior of a function or class, and returns a modified version of that function or class.

```
class C(object):
      @classmethod
      def fun(cls, arg1, arg2, ...):
            ....
```

fun: function that needs to be converted into a class method

Class methods

```
class Student:
    school_name = 'ABC School' # class
variable

    def __init__(self, name, age):
        self.name = name # instance variables
        self.age = age # instance variables

    # instance method
    def show(self):
        # access instance variables and class
variables
        print('Student:', self.name, self.age,
Student.school_name)

    @classmethod
    def modify_school_name(cls, new_name):
        # modify class variable
        cls.school_name = new_name
```

```
stud = Student('Emma', 14)
stud.show()
Student.modify_school_name("XYZ School")
stud.show()
```

OUTPUT

```
Student: Emma, 14, at ABC School
Student: Emma, 14, at XYZ School
```

https://www.w3schools.com/python/python_classes.asp

Static Class methods

- A static method does **NOT** receive an implicit first argument.
- A static method is also a method that is bound to the class and not the object of the class.
- This method can’t access or modify the class state.
- Syntax for creating a static method:

Static Method

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...
```

`@staticmethod` is also a built-in function decorator

Class Method

```
class C(object):
      @classmethod
      def fun(cls, arg1, arg2, ...):
          ....
```

`fun`: function that needs to be converted into a class method

https://www.geeksforgeeks.org/python-classes-and-objects/

Static Class methods

```python
from datetime import date

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

    # a class method to create
    # a Person object by birth year.
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # a static method to check
    # if a Person is an adult or not.
    @staticmethod
    def isAdult(age):
        return age > 18
```

```python
p1 = Person('mayank', 21)
p2 = Person.fromBirthYear('mayank', 1996)

print(p1.age)
print(p2.age)

# print the result
print(Person.isAdult(22))
```

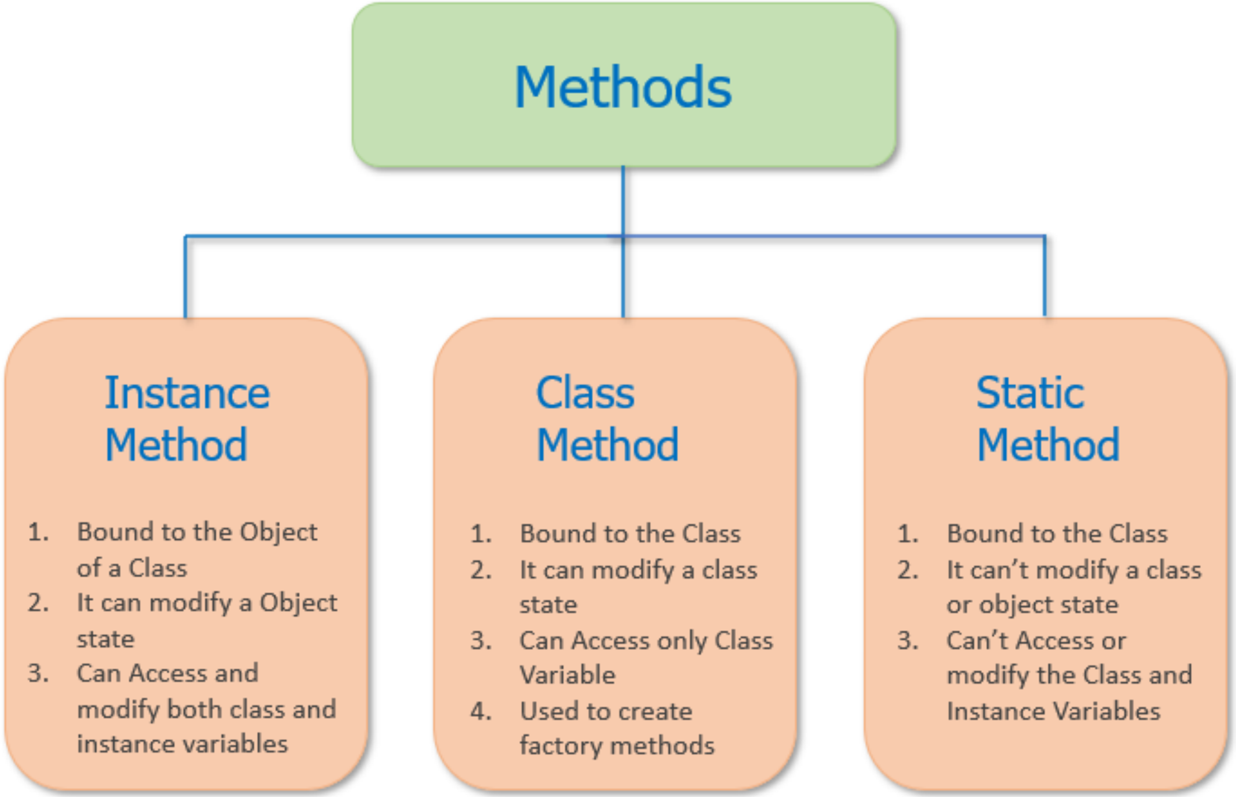
OUTPUT

```
21
26
True
```

https://www.geeksforgeeks.org/python-classes-and-objects/

Instance vs Class vs Static Class methods

- **Instance method** performs a set of actions on the data/value provided by the instance variables. If we use instance variables inside a method, such methods are called instance methods.
- **Class method** is method that is called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.
- **Static method** is a general utility method that performs a task in isolation. This method doesn't have access to the instance and class variable.



Methods

Instance Method

1. Bound to the Object of a Class
2. It can modify a Object state
3. Can Access and modify both class and instance variables

Class Method

1. Bound to the Class
2. It can modify a class state
3. Can Access only Class Variable
4. Used to create factory methods

Static Method

1. Bound to the Class
2. It can't modify a class or object state
3. Can't Access or modify the Class and Instance Variables

https://www.geeksforgeeks.org/python-classes-and-objects/

Constructors

- In object-oriented programming, a constructor is a special method used to create and initialize an object of a class. This method is defined in the class.
- The primary use of a constructor is to declare and initialize data member or instance variables of a class.
- The constructor contains a collection of statements (i.e., instructions) that executes at the time of object creation to initialize the attributes of an object.
- In Python, Object creation is divided into two parts
  - **Object Creation**: Internally, `__new__()` is the method that creates the object
  - **Object initialization**: Using the `__init__()` method, we can implement a constructor to initialize the object.

Constructors

In Python, we have the following three types of constructors:

- Default Constructor
- Non-parametrized constructor
- Parameterized constructor

```
# Default Constructor
class Student1:
    def show():
        print('Hi!')

# Non-Parameterized Constructor
class Student2:
    def __init__(self):
        self.name = 'Harry Potter'

# Parameterized Constructor
class Student3:
    def __init__(self, name):
        self.name = name

# Constructor With Default Values
class Student4:
    def __init__(self, name='Harry Potter'):
        self.name = name
```

https://www.geeksforgeeks.org/python-classes-and-objects/

Destructors

- In object-oriented programming, a destructor is a special method that is called when an object gets destroyed. The destructor is the reverse of the constructor.
- Destructor is used to perform the clean-up activity before destroying the object, such as closing database connections or filehandle.
- In Python, destructor is not called manually but completely automatic.
- The special method `__del__()` is used to define a destructor.
- For example, when we execute `del object_name` destructor gets called automatically and the object gets garbage collected.

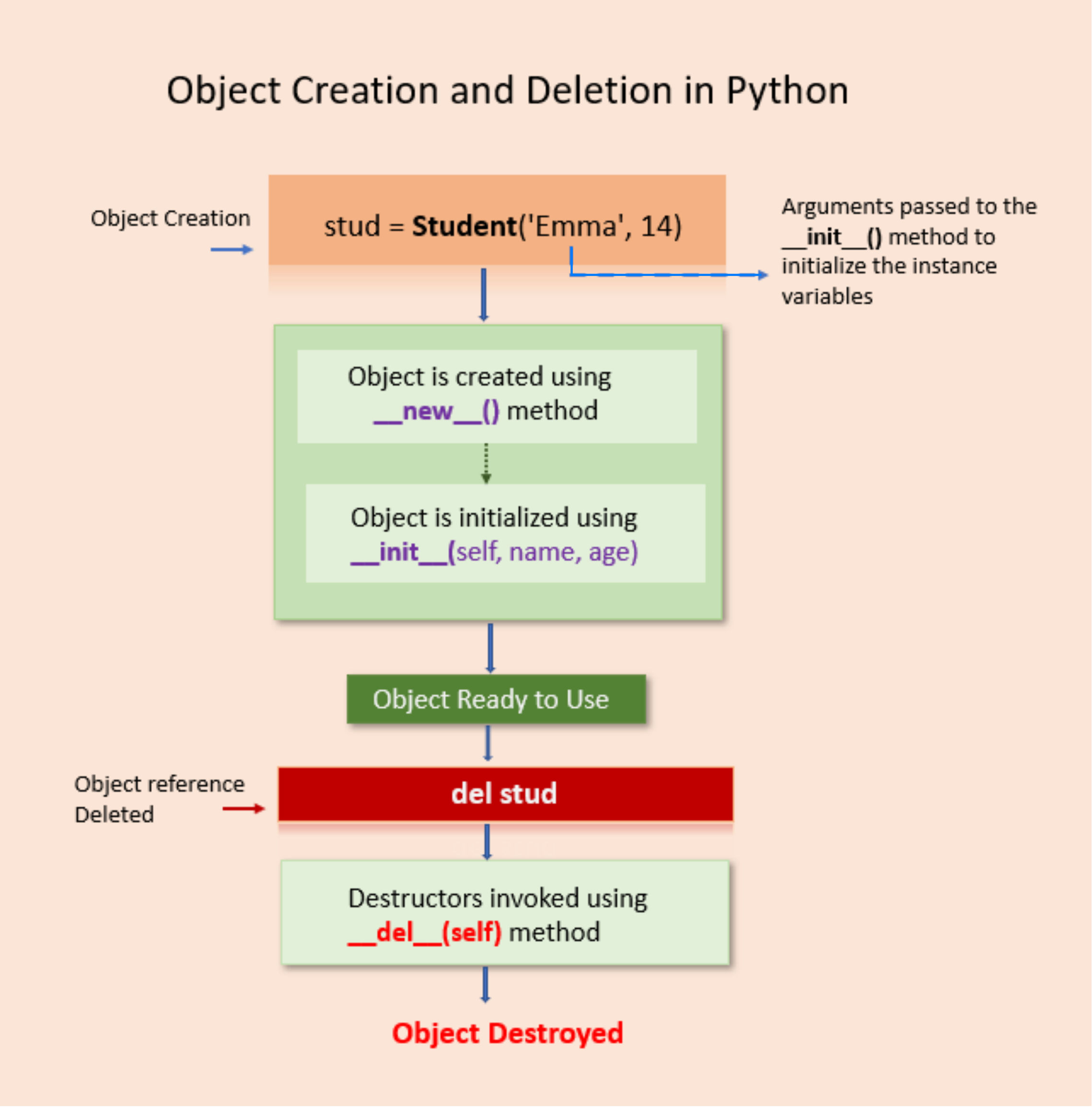
https://pynative.com/python-destructor/

Destructors

```python
class Student:

    # constructor
    def __init__(self, name, age):
        print('Inside Constructor')
        self.name = name
        self.age = age
        print('Object initialized')

    # destructor
    def __del__(self):
        print('Inside destructor')
        print('Object destroyed')

# create object
s1 = Student('Emma', 14)

# delete object
del s1
```

OUTPUT

```
Inside constructor
Object initialized
Inside destructor
Object destroyed
```



Object Creation and Deletion in Python

Object Creation

stud = **Student**('Emma', 14)

Arguments passed to the __init__() method to initialize the instance variables

Object is created using __new__() method

Object is initialized using __init__(self, name, age)

Object Ready to Use

Object reference Deleted

del stud

Destructors invoked using __del__(self) method

Object Destroyed

https://pynative.com/python-destructor/

Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called **base class**.
- **Child class** is the class that inherits from another class, also called **derived class**.
- In OOP, inheritance is an important aspect. The main purpose of inheritance is the reusability of code because we can use the existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires all the data members, properties, and functions from the parent class. Also, a child class can also provide its specific implementation to the methods of the parent class.
- For example, In the real world, Car is a sub-class of a Vehicle class. We can create a Car by inheriting the properties of a Vehicle such as Wheels, Colors, Fuel tank, engine, and add extra properties in Car as required.

https://pynative.com/python-inheritance/
https://www.w3schools.com/python/python_inheritance.asp

Inheritance

In Python, based upon the number of child and parent classes involved, there are five types of inheritance. The type of inheritance are listed below:

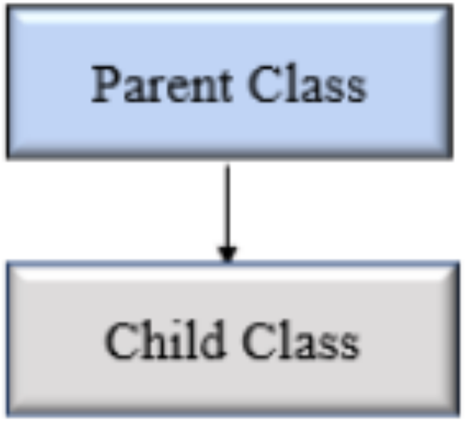
1. Single inheritance
2. Multiple Inheritance
3. Multilevel inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

https://pynative.com/python-inheritance/
https://www.w3schools.com/python/python_inheritance.asp

Inheritance

Single inheritance: a child class inherits from a single-parent class

```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Create object of Car
car = Car()

# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```



OUTPUT

```
Inside Vehicle class
Inside Car class
```

https://pynative.com/python-inheritance/
https://www.w3schools.com/python/python_inheritance.asp

Inheritance

SUSTech Southern University of Science and Technology

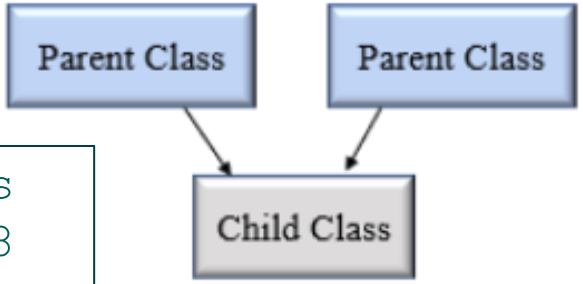
Multiple inheritance: a child class can inherit from multiple parent classes

```python
# Parent class 1
class Person:
    def person_info(self, name, age):
        print('Inside Person class')
        print('Name:', name, 'Age:', age)

# Parent class 2
class Company:
    def company_info(self, company_name):
        print('Inside Company class')
        print('Name:', company_name)

# Child class
class Employee(Person, Company):
    def Employee_info(self, salary):
        print('Inside Employee class')
        print('Salary:', salary)
```

```python
# Create object of Employee
emp = Employee()

# access data
emp.person_info('Jessa', 28)
emp.company_info('Google')
emp.Employee_info(12000)
```

OUTPUT

```
Inside Person class
Name: Jessa Age: 28
Inside Company class
Name: Google
Inside Employee class
Salary: 12000
```



https://pynative.com/python-inheritance/
https://www.w3schools.com/python/python_inheritance.asp

Inheritance

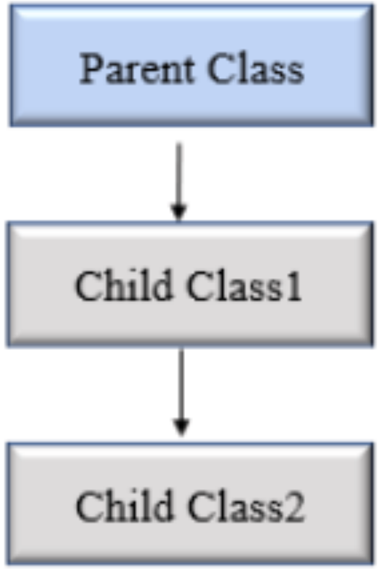
Multilevel inheritance: a class inherits from a child class or derived class (a chain of classes).

```python
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class 1
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Child class 2
class SportsCar(Car):
    def sports_car_info(self):
        print('Inside SportsCar class')
```

```python
# Create object of SportsCar
s_car = SportsCar()

# access Vehicle's and Car info
using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

OUTPUT

```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```



https://pynative.com/python-inheritance/
https://www.w3schools.com/python/python_inheritance.asp

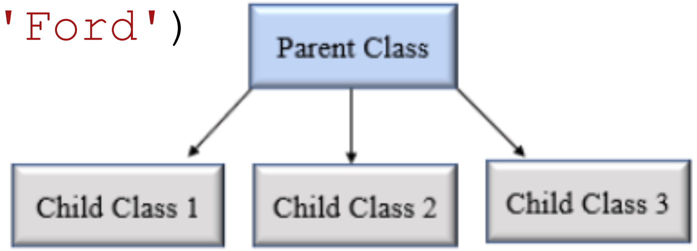
Inheritance

SUSTech Southern University of Science and Technology

Hierarchical inheritance: more than one child class is derived from a single parent class.

```
# Base class
class Vehicle:
    def info(self):
        print("This is Vehicle")

# Child class 1
class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)

# Child class 2
class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)
```

```
obj1 = Car()
obj1.info()
obj1.car_info('BMW')

obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```



OUTPUT

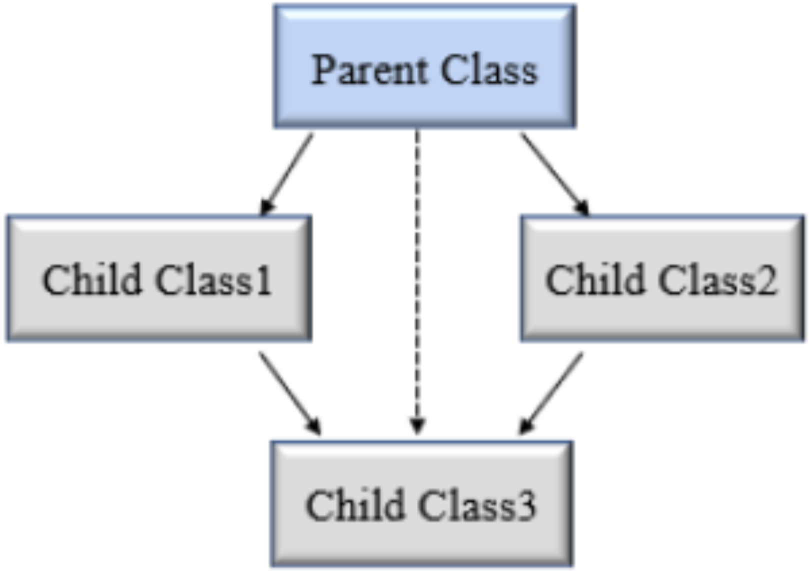
```
This is Vehicle
Car name is: BMW
This is Vehicle
Truck name is: Ford
```

https://pynative.com/python-inheritance/
https://www.w3schools.com/python/python_inheritance.asp

Inheritance

Hybrid inheritance: When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.

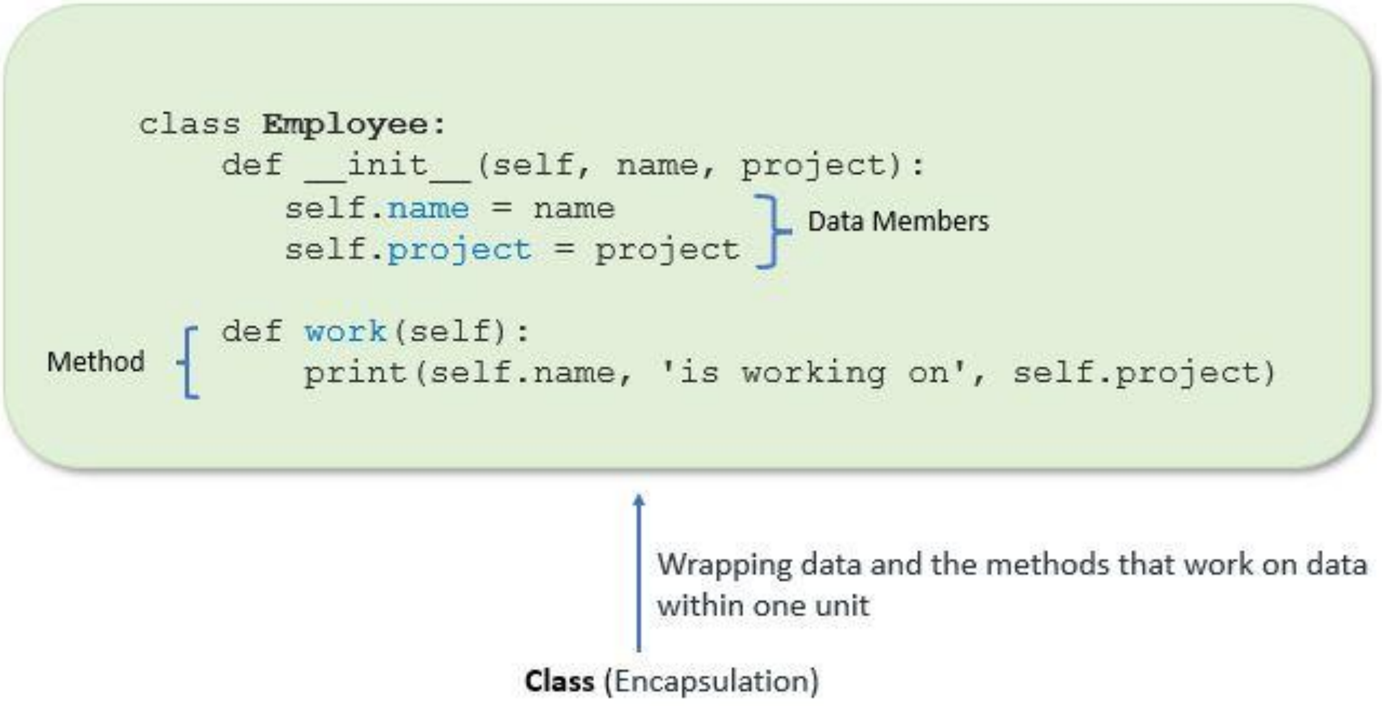
```python
# Example of hierarchical and multiple inheritance exists together.
class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)

class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)

class SportsCar(Car, Vehicle):
    def sports_car_info(self):
        print("Inside SportsCar class")
```



https://pynative.com/python-inheritance/
https://www.w3schools.com/python/python_inheritance.asp

Encapsulation

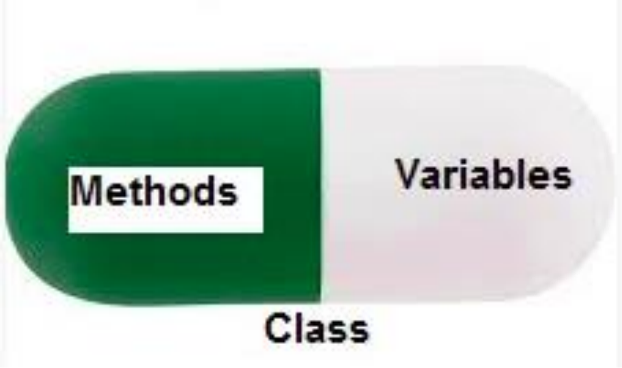
- **Encapsulation** is one of the fundamental concepts in OOP. It describes the concept of bundling data and methods within a single unit.
- A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.



```
class Employee:
    def __init__(self, name, project):
        self.name = name
        self.project = project

    def work(self):
        print(self.name, 'is working on', self.project)
```

Wrapping data and the methods that work on data within one unit

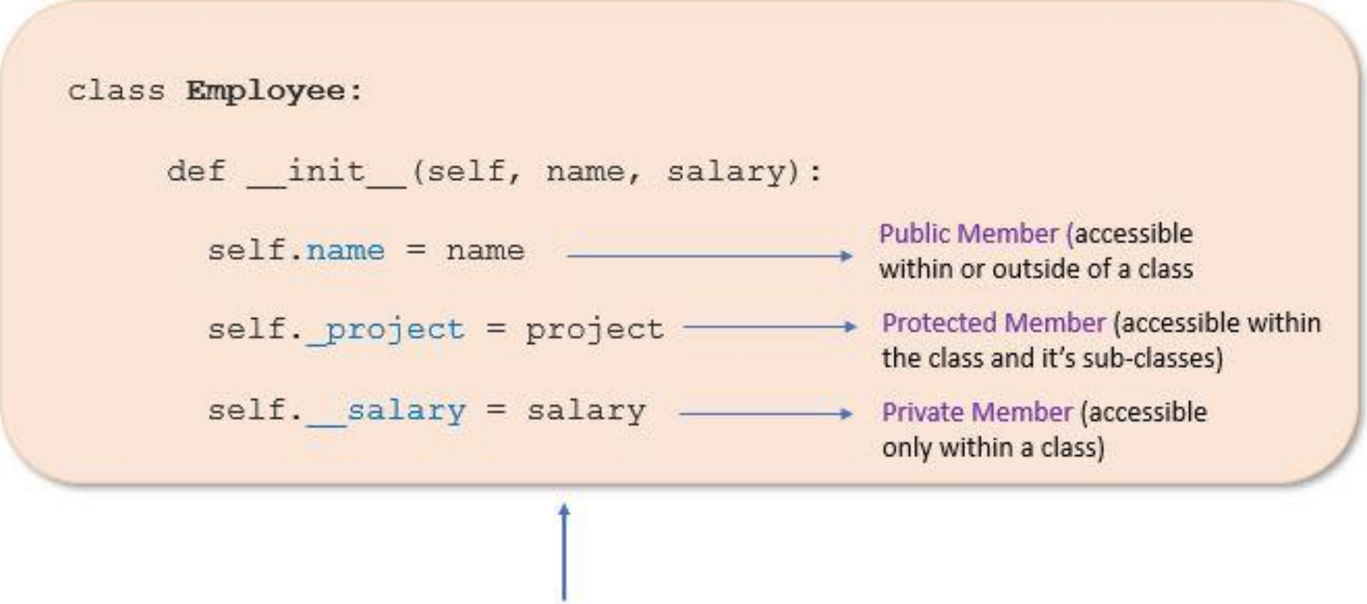
Class (Encapsulation)



https://pynative.com/python-encapsulation/
https://www.geeksforgeeks.org/encapsulation-in-python

Encapsulation

- Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected:
  - **Public Member**: Accessible anywhere from within or outside class.
  - **Protected Member (_)**: Accessible within the class and its sub-classes
  - **Private Member (__)**: Accessible only within the class
- Python doesn’t have direct access modifiers for these members, but we can use **single underscore (_)** and **double underscores (__)**.



```
class Employee:

    def __init__(self, name, salary):

      self.name = name

      self._project = project

      self.__salary = salary
```

Public Member (accessible within or outside of a class
Protected Member (accessible within the class and it’s sub-classes)
Private Member (accessible only within a class)

Data Hiding using Encapsulation

https://pynative.com/python-encapsulation/
https://www.geeksforgeeks.org/encapsulation-in-python

Encapsulation

- **Public Member**: Accessible anywhere from within or outside class.

```python
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)
```

https://pynative.com/python-encapsulation/
https://www.geeksforgeeks.org/encapsulation-in-python

Encapsulation

- **Protected Member (_)**: Accessible within the class and its sub-classes.
- Protected data members are used when you implement inheritance and want to allow data members access to only child classes.

```
# base class
class Company:
    def __init__(self):
        # Protected member
        self._project = "NLP"

# child class
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)

    def show(self):
        print("Employee name :", self.name)
        # Accessing protected member in child class
        print("Working on project :", self._project)
```

```
c = Employee("Jessa")
c.show()
# Direct access protected data member
# This should be avoided!!!
print('Project:', c._project)
```

OUTPUT

```
Employee name : Jessa
Working on project : NLP
Project: NLP
```

NOTE: Protected instance variables don’t exist in Python. The "_" is a convention and should be considered an implementation detail.

https://pynative.com/python-encapsulation/
https://www.geeksforgeeks.org/encapsulation-in-python

Encapsulation

- **Private Member (__):** Accessible only within the class.

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing private data members
print('Salary:', emp.__salary)
```

OUTPUT

```
Traceback (most recent call last):
  File "c:\Users\shimi\Downloads\test.py",
line 13, in <module>
    print('Salary:', emp.__salary)
AttributeError: 'Employee' object has no
attribute '__salary'
```

https://pynative.com/python-encapsulation/
https://www.geeksforgeeks.org/encapsulation-in-python

Encapsulation

- **Private Member (__):** Accessible only ~~within the class~~.
- Two ways of accessing private members: **public method** and **name mangling** (_classname__dataMember).

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

    # public instance methods
    def show_salary(self):
        # private members are accessible
        # from a class
        print('Salary:', self.__salary)
```

```
# creating object of a class
emp = Employee('Jessa', 10000)

# calling public method of the class
emp.show_salary()

# direct access to private member using
name mangling
print('Salary:', emp._Employee__salary)
```

OUTPUT

```
Salary: 10000
Salary: 10000
```

https://pynative.com/python-encapsulation/
https://www.geeksforgeeks.org/encapsulation-in-python

Encapsulation

Advantages of Encapsulation:

- **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.
- **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.

https://pynative.com/python-encapsulation/
https://www.geeksforgeeks.org/encapsulation-in-python

Polymorphism

- **Polymorphism** is another essential feature of OOP.
- Polymorphism in Python is the ability of an object to take many forms.
- In polymorphism, a method can process objects differently depending on the class type or data type.

```python
# Python program to demonstrate in-built
# polymorphic functions

# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

OUTPUT

```
5
3
```

https://pynative.com/python-polymorphism/
https://www.geeksforgeeks.org/polymorphism-in-python

Polymorphism

Polymorphism with inheritance

```
class Vehicle:

    def __init__(self, name, color, price):
        self.name = name
        self.color = color
        self.price = price

    def max_speed(self):
        print('Vehicle max speed is 150')


# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')
```

```
# Car Object
car = Car('Car x1', 'Red', 20000)
# calls methods from Car class
car.max_speed()

# Vehicle Object
vehicle = Vehicle('Truck x1', 'white',
75000)
# calls method from a Vehicle class
vehicle.max_speed()
```

OUTPUT

```
Car max speed is 240
Vehicle max speed is 150
```

Method Overriding: the `max_speed()` method in `Vehicle` objects is overridden by the one in `Car` objects.

https://pynative.com/python-polymorphism/
https://www.geeksforgeeks.org/polymorphism-in-python

Polymorphism

Polymorphism with Function and Objects

```python
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

# normal function
def car_details(obj):
    obj.fuel_type()
    obj.max_speed()
```

```python
ferrari = Ferrari()
bmw = BMW()

car_details(ferrari)
car_details(bmw)
```

OUTPUT

```
Petrol
Max speed 350
Diesel
Max speed is 240
```

The `car_details()` function works for both the `Ferrari` objects is overridden by the one in `BMW` objects.

https://pynative.com/python-polymorphism/
https://www.geeksforgeeks.org/polymorphism-in-python

Exercises

Exercise 1

Write a Python program to create a `Vehicle` class with `name`, `max_speed` and `mileage` instance attributes.

Exercise 2

Create a child class called `Bus` that will inherit all of the variables and methods of the `Vehicle` class in exercise 1.

Add a class variable `vehicle_type = "bus"` to `Bus`.

Write a method called `show` that prints out all the class and instance variables of the `bus` object.

Exercise 3

1. Write a **Rectangle class** in Python language, allowing you to build a rectangle with **length** and **width** attributes.

2. Create a **Perimeter()** method to calculate the perimeter of the rectangle and an **Area()** method to calculate the area of the rectangle.

3. Create a method **display()** that display the length, width, perimeter and area of an object created using an instantiation on rectangle class.

4. Create a **Parallelepipede** child class **inheriting** from the **Rectangle class** and with a **height** attribute and another **Volume()** method to calculate the volume of the **Parallelepiped**.

5. Test your class with the following code

```
myRectangle = Rectangle(7 , 5)
myRectangle.display()
myParallelepipede = Parallelepipede(7 , 5 , 2)
print("the volume is: " , myParallelepipede.volume())
```

OUTPUT

```
The length of rectangle is:  7
The width of rectangle is:  5
The perimeter of rectangle is:  24
The area of rectangle is:  35
the volume is:  70
```

Exercise 4

1. Create a Python class called `BankAccount` which represents a bank account, having as attributes: `accountNumber` (numeric type), `name` (name of the account owner as string type), `balance`.
2. Create a **constructor** with parameters: `accountNumber`, `name`, `balance`.
3. Create a `Deposit()` method which manages the deposit actions.
4. Create a `Withdrawal()` **method** which manages withdrawals actions.
5. Create an `bankFees()` method to apply the bank fees with a percentage of 5% of the balance account.
6. Create a `display()` method to display account details.
7. Give the complete code for the `BankAccount` **class**.
8. Test your code with the following example:

```python
# Testing the code :
newAccount = BankAccount(2178514584, "Albert" , 2700)
# Creating Withdrawal Test
newAccount.Withdrawal(300)
# Create deposit test
newAccount.Deposit(200)
# Display account informations
newAccount.display()
```

OUTPUT

```
Account Number :  2178514584
Account Name :  Albert
Account Balance :  2600 $
```