

# Object Oriented Programming

- [Object Oriented Basics](#)
- [Encapsulation](#)
- [Abstraction](#)
- [Inheritance](#)
- [Polymorphism](#)
- [Built-in Methods](#)
- [Class Creation: Getter/Setter vs Property](#)

# Object Oriented Basics

Yes, all the object-oriented programming (OOP) terms and concepts mentioned apply to Python. Python is a multi-paradigm programming language that fully supports object-oriented programming. In fact, OOP is one of the primary programming paradigms used in Python, and its syntax and features are designed to facilitate object-oriented design and development.

Here's how these OOP terms apply to Python specifically:

1. **Class:** In Python, classes are defined using the `class` keyword, and they encapsulate data (attributes) and behavior (methods). Objects are created by instantiating classes using the class name followed by parentheses, optionally passing arguments to the constructor (`__init__` method).
2. **Object:** Objects in Python are instances of classes. They have attributes (instance variables) and methods (functions defined within the class). Objects are created dynamically at runtime and can be assigned to variables, passed as arguments, and returned from functions.
3. **Attribute:** Attributes in Python are data items associated with objects. They can be accessed using dot notation (`object.attribute`) or through methods defined within the class. Attributes can be public, private (using name mangling), or protected (using single underscore convention).
4. **Method:** Methods in Python are functions defined within a class that operate on its instances. They are defined using the `def` keyword within the class definition. Methods can access and modify the object's state (attributes) and behavior.
5. **Inheritance:** Python supports single and multiple inheritance, allowing classes to inherit attributes and methods from one or more parent classes. Inheritance relationships are defined using parentheses after the class name in the class definition.
6. **Encapsulation:** Python supports encapsulation through the use of classes, which bundle data and methods together. Python does not have built-in support for access modifiers like private or protected, but encapsulation can be achieved through conventions and name mangling.
7. **Polymorphism:** Python supports polymorphism through method overriding and duck typing. Method overriding allows subclasses to provide their own implementation of methods inherited from parent classes, while duck typing allows objects to be treated uniformly based on their behavior rather than their type.
8. **Abstraction:** Python supports abstraction through classes and interfaces, allowing programmers to model real-world entities and concepts while hiding implementation details. Python encourages writing code that operates on interfaces rather than concrete implementations.
9. **Constructor and Destructor:** In Python, the constructor is defined using the `__init__` method, which is automatically called when an object is instantiated. Python does not have explicit destructors, but the `__del__` method can be used to define cleanup tasks.

when an object is garbage-collected.

10. **Instance, Class, and Instance Variables:** Python distinguishes between instance variables (attributes specific to individual objects) and class variables (attributes shared among all instances of a class). Instance variables are defined within methods using the `self` parameter, while class variables are defined outside methods within the class.
11. **Method Overriding and Overloading:** Python supports method overriding by allowing subclasses to provide their own implementations of methods inherited from parent classes. However, Python does not support method overloading in the traditional sense (having multiple methods with the same name but different signatures), but you can achieve similar functionality using default parameter values or variable-length argument lists.

Overall, Python's support for object-oriented programming makes it a versatile and powerful language for designing and implementing software systems using OOP principles.

In Python, the `self` keyword is used within methods of a class to refer to the instance of the class itself. It is passed implicitly as the first argument to instance methods, allowing those methods to access and modify attributes of the instance. Here's a guideline on when to use `self` and when not to:

## Use `self`:

1. **Inside Instance Methods:** Within instance methods, use `self` to access instance attributes and methods.

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def print_value(self):
        print(self.value)

obj = MyClass(10)
obj.print_value() # Output: 10
```

**When Assigning Instance Attributes:** Use `self` to assign instance attributes within the `__init__` method or other instance methods.

```
class MyClass:
    def __init__(self, value):
```

```
self.value = value
```

```
obj = MyClass(10)
```

## Do Not Use `self`:

**Outside Class Definition:** When accessing class attributes or methods outside the class definition, you do not need to use `self`.

```
class MyClass:
    class_attribute = 100

    def __init__(self, value):
        self.instance_attribute = value

obj = MyClass(10)
print(obj.instance_attribute) # Output: 10
print(MyClass.class_attribute) # Output: 100
```

**Static Methods and Class Methods:** In static methods and class methods, `self` is not used because these methods are not bound to a specific instance.

```
class MyClass:
    @staticmethod
    def static_method():
        print("Static method")

    @classmethod
    def class_method(cls):
        print("Class method")

MyClass.static_method() # Output: Static method
MyClass.class_method() # Output: Class method
```

Remember that `self` is a convention in Python, and you could technically name the first parameter of an instance method anything you like, but using `self` is highly recommended for readability and consistency with Python conventions.



# Encapsulation

## Class Creation Considerations:

### Encapsulation in Python



1. **Encapsulation:** the bundling of Variables and Methods in a Class to ensure that the behavior of an object can only be affected through its Public API. It lets us control how much a change to one object will impact other parts of the system by ensuring that there are no unexpected dependencies between unrelated components.
  - Attributes: Variables are assigned attributes within a `__init__` function
  - Public API: Methods are functions that manipulate the Attributes, such as `get/set`
2. **Information Hiding:** Restricting access to implementation details and violating state invariance
  - Protected Variable:
    - Members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses
    - In Python, Using underscore (`_name`) to denote Protected
  - Private Variable:
    - Members can neither be accessed outside the class nor by any base class
    - In Python, double underscore (`__name`) to denote Private.
  - Note that Python is not a true Protected or Private enforced by the language like C++
    - Can still access protected and private variables via dot notation (`hello._name`)

```
from dataclasses import dataclass
from enum import Enum
from typing import Any
```

```
class PaymentStatus(Enum):
```

```
    CANCELLED = "cancelled"
```

```
    PENDING = "pending"
```

```
    PAID = "paid"
```

```
class PaymentStatusError(Exception):
```

```
    pass
```

```
@dataclass
```

```
class OrderNoEncapsulationNoInformationHiding:
```

```
    """Anyone can get the payment status directly via the instance variable.
```

```
    There are no boundaries whatsoever."""
```

```
    payment_status: PaymentStatus = PaymentStatus.PENDING
```

```
@dataclass
```

```
class OrderEncapsulationNoInformationHiding:
```

```
    """There's an interface now that you should use that provides encapsulation.
```

```
    Users of this class still need to know that the status is represented by an enum type."""
```

```
    _payment_status: PaymentStatus = PaymentStatus.PENDING
```

```
    def get_payment_status(self) -> PaymentStatus:
```

```
        return self._payment_status
```

```
    def set_payment_status(self, status: PaymentStatus) -> None:
```

```
        if self._payment_status == PaymentStatus.PAID:
```

```
            raise PaymentStatusError(
```

```
                "You can't change the status of an already paid order."
```

```
            )
```

```
        self._payment_status = status
```

```
@dataclass
```

```
class OrderEncapsulationAndInformationHiding:
```

""The status variable is set to 'private'. The only thing you're supposed to use is the is\_paid method, you need no knowledge of how status is represented (that information is 'hidden')."""

```
_payment_status: PaymentStatus = PaymentStatus.PENDING
```

```
def is_paid(self) -> bool:
```

```
    return self._payment_status == PaymentStatus.PAID
```

```
def is_cancelled(self) -> bool:
```

```
    return self._payment_status == PaymentStatus.CANCELLED
```

```
def cancel(self) -> None:
```

```
    if self._payment_status == PaymentStatus.PAID:
```

```
        raise PaymentStatusError("You can't cancel an already paid order.")
```

```
    self._payment_status = PaymentStatus.CANCELLED
```

```
def pay(self) -> None:
```

```
    if self._payment_status == PaymentStatus.PAID:
```

```
        raise PaymentStatusError("Order is already paid.")
```

```
    self._payment_status = PaymentStatus.PAID
```

```
@dataclass
```

```
class OrderInformationHidingWithoutEncapsulation:
```

```
    ""The status variable is public again (so there's no boundary),  
    but we don't know what the type is - that information is hidden. I know, it's a bit  
    of a contrived example - you wouldn't ever do this. But at least it shows that  
    it's possible.""
```

```
    payment_status: Any = None
```

```
def is_paid(self) -> bool:
```

```
    return self.payment_status == PaymentStatus.PAID
```

```
def is_cancelled(self) -> bool:
```

```
    return self.payment_status == PaymentStatus.CANCELLED
```

```
def cancel(self) -> None:
```

```
    if self.payment_status == PaymentStatus.PAID:
```

```
        raise PaymentStatusError("You can't cancel an already paid order.")
```



```
self.payment_status = PaymentStatus.CANCELLED
```

```
def pay(self) -> None:
```

```
    if self.payment_status == PaymentStatus.PAID:
```

```
        raise PaymentStatusError("Order is already paid.")
```

```
    self.payment_status = PaymentStatus.PAID
```

```
def main() -> None:
```

```
    test = OrderInformationHidingWithoutEncapsulation()
```

```
    test.pay()
```

```
    print("Is paid: ", test.is_paid())
```

```
if __name__ == "__main__":
```

```
    main()
```

# Abstraction

## Prereq: Inheritance

**Abstraction** is used to hide something too, but in a **higher degree (class, module)**. Clients who use an abstract class (or interface) do not care about what it was, they just need to know what it can do.

**Abstract class** contains one or more abstract methods, and is considered a blueprint for other [classes](#). It allows you to create a set of methods that must be created within any child classes built from the abstract class.

**Abstract method** is a method that has a declaration but does not have an implementation.

- An abstract class is a class that is declared abstract — it may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed. This is the differentiation from Inheritance
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class

Be aware, Abstraction reduces code repetition but increases coupling.

<https://www.geeksforgeeks.org/abstract-classes-in-python/>

## Working on Python Abstract classes

Here, This code defines an abstract base class called “Animal” using the ABC (Abstract Base Class) module in Python. The “Animal” class has a non-abstract method called “move” that does not have any implementation. There are four subclasses of “Animal” defined: “Human,” “Snake,” “Dog,” and “Lion.” Each of these subclasses overrides the “move” method and provides its own implementation by printing a specific movement characteristic.

Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

```
# abstract base class work
from abc import ABC, abstractmethod
```

```
class Animal(ABC):

    def move(self):
        pass

class Human(Animal):

    def move(self):
        print("I can walk and run")

class Snake(Animal):

    def move(self):
        print("I can crawl")

class Dog(Animal):

    def move(self):
        print("I can bark")

class Lion(Animal):

    def move(self):
        print("I can roar")

# Driver code
R = Human()
R.move()

K = Snake()
K.move()

R = Dog()
R.move()

K = Lion()
K.move()
```

## Concrete Methods in Abstract Base Classes (Super)

Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

The concrete class provides an implementation of abstract methods, the abstract base class can also provide an implementation by invoking the methods via `super()`. Let look over the example to invoke the method using `super()`:

```
# Python program invoking a
# method using super()
```

```
from abc import ABC

class R(ABC):
    def rk(self):
        print("Abstract Base Class")

class K(R):
    def rk(self):
        super().rk()
        print("subclass ")

# Driver code
r = K()
r.rk()
```

Abstract Base Class  
subclass

## Abstract Property

- use the decorator to prevent users from calling abstract method

```
# Python program showing
# abstract properties

import abc
from abc import ABC, abstractmethod

class parent(ABC):
    @abc.abstractproperty
    def geeks(self):
        return "parent class"
class child(parent):

    @property
    def geeks(self):
        return "child class"

try:
    r =parent()
    print( r.geeks)
except Exception as err:
    print (err)

r = child()
print (r.geeks)
```

Can't instantiate abstract class parent with abstract methods geeks  
child class

Protocol vs ABC

# Inheritance

“ Inheritance is a mechanism that allows a class to inherit properties and behaviors from another class.

- PRO: Promotes code reuse and establishes relationships between classes.
- CON: Increases Coupling

Inheritance is based on a hierarchical relationship between classes, where a derived class (also known as a subclass or child class) inherits the characteristics of a base class (also known as a superclass or parent class). The derived class extends the functionality of the base class by adding new features or overriding existing ones.

The key idea behind inheritance is that the derived class inherits all the attributes (data members) and behaviors (methods) of the base class, and it can also introduce its own specific attributes and behaviors. This allows for creating a hierarchy of classes with increasing specialization.

## Template

```
class parent_class:  
    body of parent class  
  
class child_class( parent_class):  
    body of child class
```

## Python Code:

```
class Car:      #parent class  
  
    def __init__(self, name, mileage):  
        self.name = name  
        self.mileage = mileage  
  
    def description(self):  
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"  
  
class BMW(Car):  #child class  
    pass
```

```
class Audi(Car):    #child class
    def audi_desc(self):
        return "This is the description method of class Audi."
obj1 = BMW("BMW 7-series",39.53)
print(obj1.description())

obj2 = Audi("Audi A8 L",14)
print(obj2.description())
print(obj2.audi_desc())
```

We can check the base or parent class of any class using a built-in class attribute **\_\_bases\_\_**

```
print(BMW.__bases__, Audi.__bases__)
```

### Inheritance in Object Oriented Programming - Print Class

As we can see here, the base class of both sub-classes is **Car**. Now, let's see what happens when using **\_\_base\_\_** with the parent class Car:

```
print( Car.__bases__ )Output:
```

### Inheritance in Object Oriented Programming - Print sub-class

Whenever we create a new class in Python 3.x, it is inherited from a built-in basic class called **Object**. In other words, the Object class is the root of all classes.

## Forms of Inheritance

There are broadly five forms of inheritance in oops based on the involvement of parent and child classes.

### Single Inheritance

This is a form of inheritance in which a class inherits only one parent class. This is the simple form of inheritance and hence, is also referred to as **simple inheritance**.

```
class Parent:
    def f1(self):
        print("Function of parent class.")
```

```
class Child(Parent):  
    def f2(self):  
        print("Function of child class.")  
  
object1 = Child()  
object1.f1()  
object1.f2()
```

Output:

## Inheritance in Object Oriented Programming - Single Inheritance

Here object1 is an instantiated object of class Child, which inherits the parent class 'Parent'.

## Multiple Inheritance

Avoid using multiple inheritance in Python. If you need to reuse code from multiple classes, you can use composition.

Multiple inheritances is when a class inherits more than one parent class. The child class, after inheriting properties from various parent classes, has access to all of its objects.

One of the main problems with multiple inheritance is the diamond problem. This occurs when a class inherits from two classes that both inherit from a third class. In this case, it is not clear which implementation of the method from the third class should be used.

When a class inherits from multiple classes, it can be difficult to track which methods and attributes are inherited from which class.

```
class Parent_1:  
    def f1(self):  
        print("Function of parent_1 class.")  
  
class Parent_2:  
    def f2(self):  
        print("Function of parent_2 class.")  
  
class Parent_3:  
    def f3(self):  
        print("function of parent_3 class.")
```



```

class Child(Parent_1, Parent_2, Parent_3):
    def f4(self):
        print("Function of child class.")

object_1 = Child()
object_1.f1()
object_1.f2()
object_1.f3()
object_1.f4()

```

Output:

### Inheritance in Object Oriented Programming - Multiple Inheritance

Here we have one Child class that inherits the properties of three-parent classes Parent\_1, Parent\_2, and Parent\_3. All the classes have different functions, and all of the functions are called using the object of the Child class.

But suppose a child class inherits two classes having the same function:

```

class Parent_1:
    def f1(self):
        print("Function of parent_1 class.")

class Parent_2:
    def f1(self):
        print("Function of parent_2 class.")

class Child(Parent_1, Parent_2):
    def f2(self):
        print("Function of child class.")

```

Here, the classes Parent\_1 and Parent\_2 have the same class methods, f1(). Now, when we create a new object of the child class and call f1() from it since the child class is inheriting both parent classes, what do you think should happen?

```

obj = Child()
obj.f1()

```

Output:

## Inheritance in Object Oriented Programming

So in the above example, why was the function f1() of the class Parent\_2 not inherited?

In multiple inheritances, the child class first searches for the method in its own class. If not found, then it searches in the parent classes depth\_first and left-right order. Since this was an easy example with just two parent classes, we can clearly see that class Parent\_1 was inherited first, so the child class will search the method in Parent\_1 class before searching in class Parent\_2.

But for complicated inheritance oops problems, it gets tough to identify the order. So the actual way of doing this is called **Method Resolution Order (MRO)** in Python. We can find the MRO of any class using the attribute `__mro__`.

```
Child.__mro__
```

Output:

## Inheritance in Object Oriented Programming - MRO

This tells that the Child class first visited the class Parent\_1 and then Parent\_2, so the f1() method of Parent\_1 will be called.

Let's take a bit complicated example in Python:

```
class Parent_1:
    pass

class Parent_2:
    pass

class Parent_3:
    pass

class Child_1(Parent_1,Parent_2):
    pass

class Child_2(Parent_2,Parent_3):
    pass

class Child_3(Child_1,Child_2,Parent_3):
    pass
```

Here, the class Child\_1 inherits two classes – Parent\_1 and Parent\_2. The class Child\_2 is also inheriting two classes – Parent\_2 and Parent\_3. Another class, Child\_3, is inheriting three classes – Child\_1, Child\_2, and Parent\_3.

Now, just by looking at this inheritance, it is quite hard to determine the Method Resolution Order for class Child\_3. So here is the actual use of `__mro__`.

```
Child_3.__mro__
```

Output:



image not found or type unknown

We can see that, first, the interpreter searches Child\_3, then Child\_1, followed by Parent\_1, Child\_2, Parent\_2, and Parent\_3, respectively.

## Multi-level Inheritance

For example, a class\_1 is inherited by a class\_2, and this class\_2 also gets inherited by class\_3, and this process goes on. This is known as multi-level inheritance oops. Let's understand with an example:

```
class Parent:
    def f1(self):
        print("Function of parent class.")

class Child_1(Parent):
    def f2(self):
        print("Function of child_1 class.")

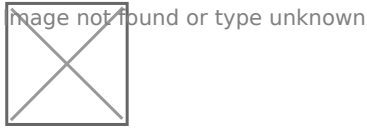
class Child_2(Child_1):
    def f3(self):
        print("Function of child_2 class.")

obj_1 = Child_1()
obj_2 = Child_2()

obj_1.f1()
obj_1.f2()
```

```
print("\n")
obj_2.f1()
obj_2.f2()
obj_2.f3()
```

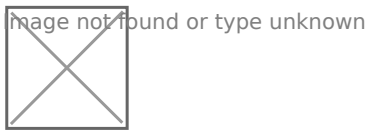
Output:



Here, the class Child\_1 inherits the Parent class, and the class Child\_2 inherits the class Child\_1. In this Child\_1 has access to functions f1() and f2() whereas Child\_2 has access to functions f1(), f2() and f3(). If we try to access the function f3() using the object of class Class\_1, then an error will occur stating:

‘Child\_1’ object has no attribute ‘f3’.

```
obj_1.f3()
```



## Hierarchical Inheritance

In this, various Child classes inherit a single Parent class. The example given in the introduction of the inheritance is an example of Hierarchical inheritance since classes BMW and Audi inherit class Car.

For simplicity, let’s look at another example:

```
class Parent:
    deff1(self):
        print("Function of parent class.")
```

```
class Child_1(Parent):
    deff2(self):
        print("Function of child_1 class.")
```

```
class Child_2(Parent):
```

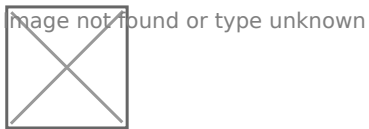
```
deff3(self):
    print("Function of child_2 class.")

obj_1 = Child_1()
obj_2 = Child_2()

obj_1.f1()
obj_1.f2()

print('\n')
obj_2.f1()
obj_2.f3()
```

Output:



Here two child classes inherit the same parent class. The class Child\_1 has access to functions f1() of the parent class and function f2() of itself. Whereas the class Child\_2 has access to functions f1() of the parent class and function f3() of itself.

## Hybrid Inheritance

When there is a combination of more than one form of inheritance, it is known as hybrid inheritance. It will be more clear after this example:

```
class Parent:
    def f1(self):
        print("Function of parent class.")

class Child_1(Parent):
    def f2(self):
        print("Function of child_1 class.")

class Child_2(Parent):
    def f3(self):
        print("Function of child_2 class.")

class Child_3(Child_1, Child_2):
```

```
def f4(self):  
    print("Function of child_3 class.")  
  
obj = Child_3()  
obj.f1()  
obj.f2()  
obj.f3()  
obj.f4()
```

Output:



In this example, two classes, 'Child\_1' and 'Child\_2', are derived from the base class 'Parent' using hierarchical inheritance. Another class, 'Child\_3', is derived from classes 'Child\_1' and 'Child\_2' using multiple inheritances. The class 'Child\_3' is now derived using hybrid inheritance.

# Method Overriding in Inheritance in Python

We do this so we can have our own methods without modifying the base class. If we modify the base class, then that could lead to other problems if other users are expecting certain functionality from the said base class

The concept of overriding is very important in inheritance oops. It gives the special ability to the child/subclasses to provide specific implementation to a method that is already present in their parent classes.

```
class Parent:  
    def f1(self):  
        print("Function of Parent class.")  
  
class Child(Parent):  
    def f1(self):  
        print("Function of Child class.")
```

```
obj = Child()
obj.f1()
```

Output:



Here the function `f1()` of the child class has overridden the function `f1()` of the parent class. Whenever the object of the child class invokes `f1()`, the function of the child class gets executed. However, the object of the parent class can invoke the function `f1()` of the parent class.

```
obj_2 = Parent()
obj_2.f1()
```

Output:



# Super() Function in Python

The `super()` function in Python returns a proxy object that references the parent class using the **super** keyword. This `super()` keyword is basically useful in accessing the overridden methods of the parent class.

The [official documentation](#) of the `super()` function sites two main uses of `super()`:

**In a class hierarchy with single inheritance oops, *super* helps to refer to the parent classes without naming them explicitly, thus making the code more maintainable.**

For example:

```
class Parent:
    def f1(self):
        print("Function of Parent class.")
```

```
class Child(Parent):
    def f1(self):
        super().f1()
        print("Function of Child class.")

obj = Child()
obj.f1()
```

Output:



Here, with the help of `super().f1()`, the `f1()` method of the superclass of the child class, i.e., the parent class, has been called without explicitly naming it.

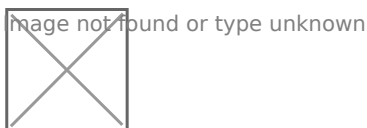
One thing to note here is that the `super()` class can accept two parameters- the first is the name of the subclass, and the second is an object that is an instance of that subclass. Let's see how:

```
class Parent:
    def f1(self):
        print("Function of Parent class.")

class Child(Parent):
    def f1(self):
        super( Child, self ).f1()
        print("Function of Child class.")

obj = Child()
obj.f1()
```

Output:



The first parameter refers to the subclass **Child**, while the second parameter refers to the object of Child, which, in this case, is **self**. You can see the output after using `super()`, and `super( Child, self)`



is the same because, in Python 3, `super( Child, self)` is equivalent to `self()`.

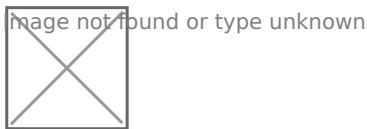
Now let's see one more example using the `__init__` function.

```
class Parent(object):
    def __init__(self, ParentName):
        print(ParentName, 'is derived from another class.')

class Child(Parent):
    def __init__(self, ChildName):
        print(name, 'is a sub-class.')
        super().__init__(ChildName)

obj = Child('Child')
```

Output:



What we have done here is that we called the `__init__` function of the parent class (inside the child class) using `super().__init__( ChildName )`. And as the `__init__` method of the parent class requires one argument, it has been passed as “ChildName”. So after creating the object of the child class, first, the `__init__` function of the child class got executed, and after that, the `__init__` function of the parent class.

**The second use case is to support multiple cooperative inheritances in a dynamic execution environment.**

```
class First():
    def __init__(self):
        print("first")
        super().__init__()

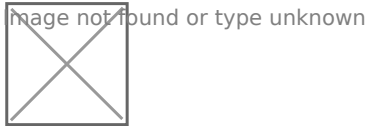
class Second():
    def __init__(self):
        print("second")
        super().__init__()

class Third(Second, First):
```

```
def __init__(self):  
    print("third")  
    super().__init__()
```

```
obj = Third()
```

Output:



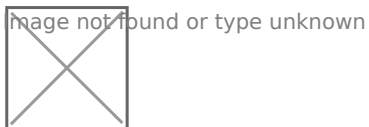
The `super()` call finds the next method in the MRO at each step, which is why `First` and `Second` have to have it, too; otherwise, execution stops at the end of `first().__init__`.

Note that the super-class of both `First` and `Second` is `Object`.

Let's find the MRO of `Third()` as well.

```
Third.__mro__
```

Output:



The order is `Third > Second > First`, and the same is the order of our output.

# Polymorphism

## Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

For example, say we have three classes: `Car`, `Boat`, and `Plane`, and they all have a method called `move()`:

Different classes with the same method:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Drive!")

class Boat:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Sail!")

class Plane:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Fly!")

car1 = Car("Ford", "Mustang")    #Create a Car class
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class
```

```
plane1 = Plane("Boeing", "747")    #Create a Plane class
```

```
for x in (car1, boat1, plane1):  
    x.move()
```

## Inheritance Class Polymorphism

What about classes with child classes with the same name? Can we use polymorphism there?

Yes. If we use the example above and make a parent class called `Vehicle`, and make `Car`, `Boat`, `Plane` child classes of `Vehicle`, the child classes inherits the `Vehicle` methods, but can override them:

```
class Vehicle:  
    def __init__(self, brand, model):  
        self.brand = brand  
        self.model = model  
  
    def move(self):  
        print("Move!")  
  
class Car(Vehicle):  
    pass  
  
class Boat(Vehicle):  
    def move(self):  
        print("Sail!")  
  
class Plane(Vehicle):  
    def move(self):  
        print("Fly!")  
  
car1 = Car("Ford", "Mustang") #Create a Car object  
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object  
plane1 = Plane("Boeing", "747") #Create a Plane object  
  
for x in (car1, boat1, plane1):  
    print(x.brand)  
    print(x.model)  
    x.move()
```



# Built-in Methods

## Instance Method vs Static Method

Instance Methods (normal methods without declaring @) specifies with that particular declared instance. For ex. a unique event is specific to a specific calendar instance, and should not exist on other calendar instances.

Static Method doesn't care about any particular instance. For ex, a weekend on a calendar applies to all calendars since it is universally agreed upon how calendars work, regardless of instance

Also look into Class Method if need to consider inheritance.

```
class Calendar:
    def __init__(self):
        self.events = []

    def add_event(self, event):
        self.events.append(event)

    @staticmethod
    def is_weekend(dt):
        pass
```

## Dataclasses

Without using Dataclass,

- If you print, it will be the memory address where the object resides.
  - To print a string, will have to modify the `__str__` dunder method
- A new object with identical information will be a new object

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

person1 = Person("Geralt", "Witcher", 30)
```

```
person2 = Person("Yennefer", "Sorceress", 25)
person3 = Person("Yennefer", "Sorceress", 25)

print(id(person2))
print(id(person3))
print(person1)

print(person3 == person2)
```

"""

OUTPUT:

140244722433808

140244722433712

<\_\_main\_\_.Person object at 0x7f8d44dcbfd0>

False

"""

With Dataclass:

- Print string information instead of object, unless using id()
- A new object will still be a new object, but if information is identical it will return True
- If forgot to write @dataclass...
  - class variables instead of instance variables

```
from dataclasses import dataclass, field
```

```
@dataclass(order=True,frozen=False)
```

```
class Person:
```

```
    sort_index: int = field(init=False, repr=False)
```

```
    name: str
```

```
    job: str
```

```
    age: int
```

```
    strength: int = 100
```

```
    def __post_init__(self):
```

```
        object.__setattr__(self, 'sort_index', self.strength)
```

```
    def __str__(self):
```

```
return f'{self.name}, {self.job} ({self.age})'
```

```
person1 = Person("Geralt", "Witcher", 30, 99)  
person2 = Person("Yennefer", "Sorceress", 25)  
person3 = Person("Yennefer", "Sorceress", 25)
```

```
print(person1)  
print(id(person2))  
print(id(person3))  
print(person3 == person2)  
print(person1 > person2)
```

```
""""  
Geralt, Witcher (30)  
140120080908048  
140120080907856  
True  
False  
""""
```

<https://www.youtube.com/watch?v=CvQ7e6yUtnw&t=389s>



# Class Creation: Getter/Setter vs Property

Getter and Setter Pattern comes from OOP languages such as C++ and Java in order to adhere to

- Encapsulation
  - the bundling of data/attributes and methods.
  - restricting access to implementation details and violating state invariants

In Python it is considered unpythonic. Instead, use @property decorator instead

## Class without Getter and Setter

```
# Basic method of setting and getting attributes in Python
# If this class had no methods, and only an __init__, then this would be fine.
# This has no Encapsulation because it directly accesses the attributes

class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

# Create a new object
human = Celsius()

# Set the temperature
human.temperature = 37

# Get the temperature attribute
print(human.temperature)
```

```
# Get the to_fahrenheit method
print(human.to_fahrenheit())
```

```
>> 37
```

```
>> 98.600000000000001
```

## Getter and Setter Pattern

- Class Attributes...
  - are variables where you can access through the instance, class, or both.
  - holds an internal state.
  - Two ways to access class attributes:
    1. Directly (breaks encapsulation)
      - If accessed directly, they become part of Public API
      - Only use if sure that no behavior (methods) will ever be attached to the variables (@dataclass?)
      - Changing implementation will be problematic:
        - Converting stored attribute -> computed attribute. Want to be able to store attributes instead of recomputing each time
      - No internal implementation
    2. Methods (ideal)
      - Two methods to respect encapsulation
        - Getter: Allows Access
        - Setter: Allows Setting or Mutation
- Implementation
  1. Making attributes Non-public
    - use underscore in variable name ( \_name )
  2. Writing Getter and Setter Methods for each attribute
    - In a class, `__init__` is the attributes (variables)

```
# Making Getters and Setter methods
```

```
# PRO: Adheres to Encapsulation since get/set methods is used to change the attributes instead of direct access
```

```
# PRO: Adheres to a faux Information Hiding since converting temperature to _temperature, but no
private/protected vars in Python
```

```
# CON: Has more code to change
```

```
class Celsius:
```

```
    def __init__(self, temperature=0):
        self.set_temperature(temperature)
```

```

def to_fahrenheit(self):
    return (self.get_temperature() * 1.8) + 32

# getter method
def get_temperature(self):
    return self._temperature

# setter method
def set_temperature(self, value):
    if value < -273.15:
        raise ValueError("Temperature below -273.15 is not possible.")
    self._temperature = value

# Create a new object, set_temperature() internally called by __init__
human = Celsius(37)

# Get the temperature attribute via a getter
print(human.get_temperature())

# Get the to_fahrenheit method, get_temperature() called by the method itself
print(human.to_fahrenheit())

# new constraint implementation
human.set_temperature(-300)

# Get the to_fahrenheit method
print(human.to_fahrenheit())

>> 37
>> 98.60000000000001
>> Traceback (most recent call last):
>> File "<string>", line 30, in <module>
>> File "<string>", line 16, in set_temperature
>> ValueError: Temperature below -273.15 is not possible.

```

## Property Class

```

# using property class
class Celsius:

```

```

def __init__(self, temperature=0):
    self.temperature = temperature

def to_fahrenheit(self):
    return (self.temperature * 1.8) + 32

# getter
def get_temperature(self):
    print("Getting value...")
    return self._temperature

# setter
def set_temperature(self, value):
    print("Setting value...")
    if value < -273.15:
        raise ValueError("Temperature below -273.15 is not possible")
    self._temperature = value

# creating a property object
temperature = property(get_temperature, set_temperature)

human = Celsius(37)

print(human.temperature)

print(human.to_fahrenheit())

human.temperature = -300

```

The reason is that when an object is created, the `__init__()` method gets called. This method has the line `self.temperature = temperature`. This expression automatically calls `set_temperature()`.

Similarly, any access like `c.temperature` automatically calls `get_temperature()`. This is what property does.

By using `property`, we can see that no modification is required in the implementation of the value constraint. Thus, our implementation is backward compatible.

# The @property Decorator

In Python, `property()` is a built-in function that creates and returns a `property` object. The syntax of this `function` is:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Here,

- `fget` is function to get value of the attribute
- `fset` is function to set value of the attribute
- `fdel` is function to delete the attribute
- `doc` is a string (like a comment)

As seen from the implementation, these function arguments are optional.

A property object has three methods, `getter()`, `setter()`, and `deleter()` to specify `fget`, `fset` and `fdel` at a later point. This means, the line:

```
temperature = property(get_temperature, set_temperature)
```

can be broken down as:

```
# make empty property
temperature = property()

# assign fget
temperature = temperature.getter(get_temperature)

# assign fset
temperature = temperature.setter(set_temperature)
```

These two pieces of code are equivalent.

Programmers familiar with [Python Decorators](#) can recognize that the above construct can be implemented as decorators.

We can even not define the names `get_temperature` and `set_temperature` as they are unnecessary and pollute the class `namespace`.

For this, we reuse the `temperature` name while defining our getter and setter functions. Let's look at how to implement this as a decorator:

```

# Using @property decorator
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        print("Getting value...")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value

# create an object
human = Celsius(37)

print(human.temperature)

print(human.to_fahrenheit())

coldest_thing = Celsius(-300)

>>Setting value...
>>Getting value...
>>37
>>Getting value...
>>98.600000000000001.
>>Setting value...
>>Traceback (most recent call last):
>> File "", line 29, in
>> File "", line 4, in __init__
>> File "", line 18, in temperature

```

>>ValueError: Temperature below -273 is not possible