OPP_01

June 12, 2025

1 The foundations of OOP

In the **procedural approach**, it's possible to distinguish two different and completely separate worlds: **the world of data, and the world of code**. The world of data is populated with variables of different kinds, while the world of code is inhabited by code grouped into modules and functions.

Functions are able to use data, but not vice versa. Furthermore, functions are able to abuse data, i.e., to use the value in an unauthorized manner (e.g., when the sine function gets a bank account balance as a parameter).

We said in the past that data cannot use functions. But is this entirely true? Are there some special kinds of data that can use functions?

Yes, there are - the ones named methods. These are functions which are invoked from within the data, not beside them. If you can see this distinction, you've taken the first step into object programming.

The object approach suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.

Every **class is like a recipe which can be used when you want to create a useful object** (this is where the name of the approach comes from). You may produce as many objects as you need to solve your problem.

Every object has a set of traits (they are called properties or attributes - we'll use both words synonymously) and is able to perform a set of activities (which are called methods).

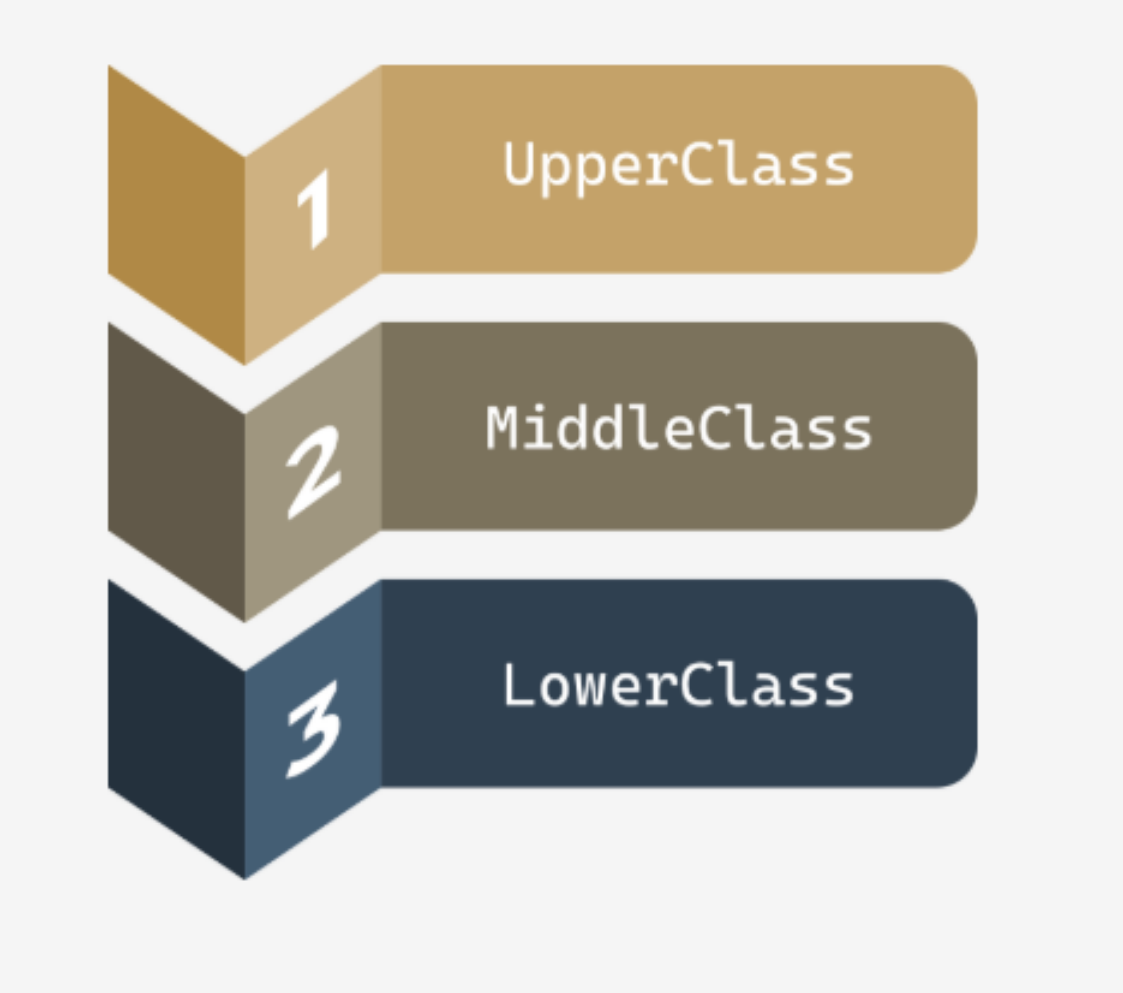
The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools.

Objects are incarnations of ideas expressed in classes, like a cheesecake on your plate is an incarnation of the idea expressed in a recipe printed in an old cookbook.

The objects interact with each other, exchanging data or activating their methods. A properly constructed class (and thus, its objects) are able to protect the sensible data and hide it from unauthorized modifications.

There is no clear border between data and code: they live as one in objects.

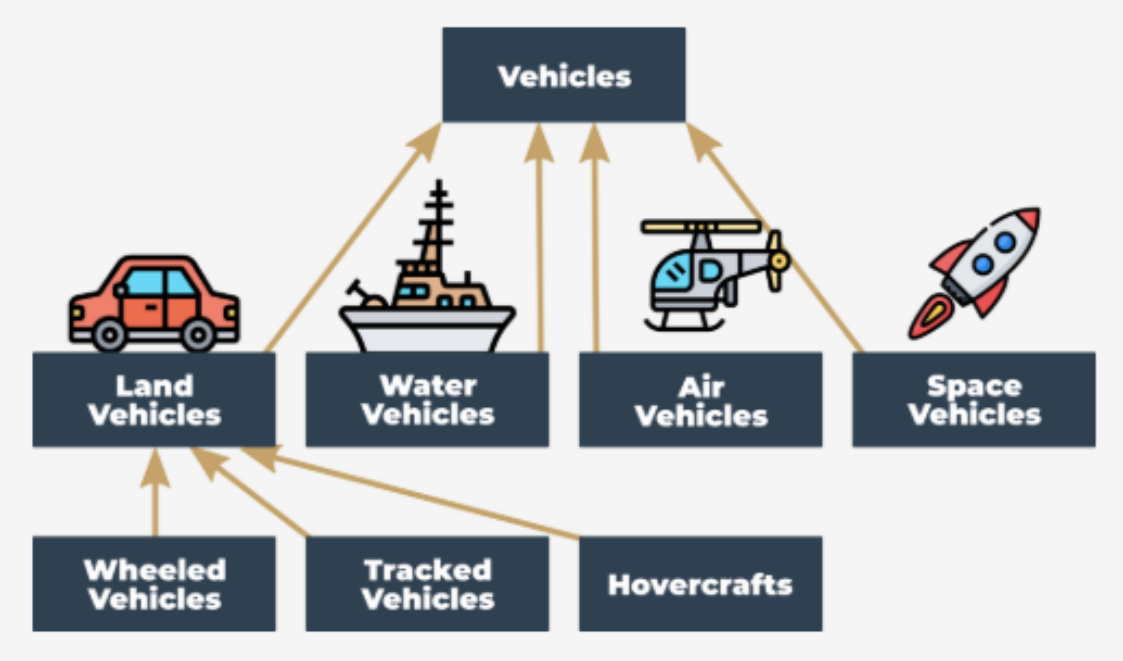
All these concepts are not as abstract as you may at first suspect. On the contrary, they all are taken from real-life experiences, and therefore are extremely useful in computer programming: they don't create artificial life - **they reflect real facts, relationships, and circumstances**.



1.1 Class hierarchies

The word class has many meanings, but not all of them are compatible with the ideas we want to discuss here. The class that we are concerned with is like a category, as a result of precisely defined similarities.

We'll try to point out a few classes which are good examples of this concept.



Let's look for a moment at vehicles. All existing vehicles (and those that don't exist yet) are **related by a single, important feature**: the ability to move. You may argue that a dog moves, too; is a dog a vehicle? No, it isn't. We have to improve the definition, i.e., enrich it with other criteria, distinguishing vehicles from other beings, and creating a stronger connection. Let's take the following circumstances into consideration: vehicles are artificially created entities used for transportation, moved by forces of nature, and directed (driven) by humans.

Based on this definition, a dog is not a vehicle.

The vehicles class is very broad. Too broad. We have to define some more specialized classes, then. The specialized classes are the subclasses. The vehicles class will be a superclass for them all.

Note: **the hierarchy grows from top to bottom, like tree roots, not branches**. The most general, and the widest, class is always at the top (the superclass) while its descendants are located below (the subclasses).

By now, you can probably point out some potential subclasses for the Vehicles superclass. There are many possible classifications. We've chosen subclasses based on the environment, and say that there are (at least) four subclasses:

- land vehicles;
- water vehicles;
- air vehicles;
- space vehicles.

In this example, we'll discuss the first subclass only - land vehicles. If you wish, you can continue with the remaining classes.

Land vehicles may be further divided, depending on the method with which they impact the ground. So, we can enumerate:

- wheeled vehicles;

- tracked vehicles;
- hovercrafts. The hierarchy we've created is illustrated by the figure.

Note the direction of the arrows - they always point to the superclass. The top-level class is an exception - it doesn't have its own superclass.

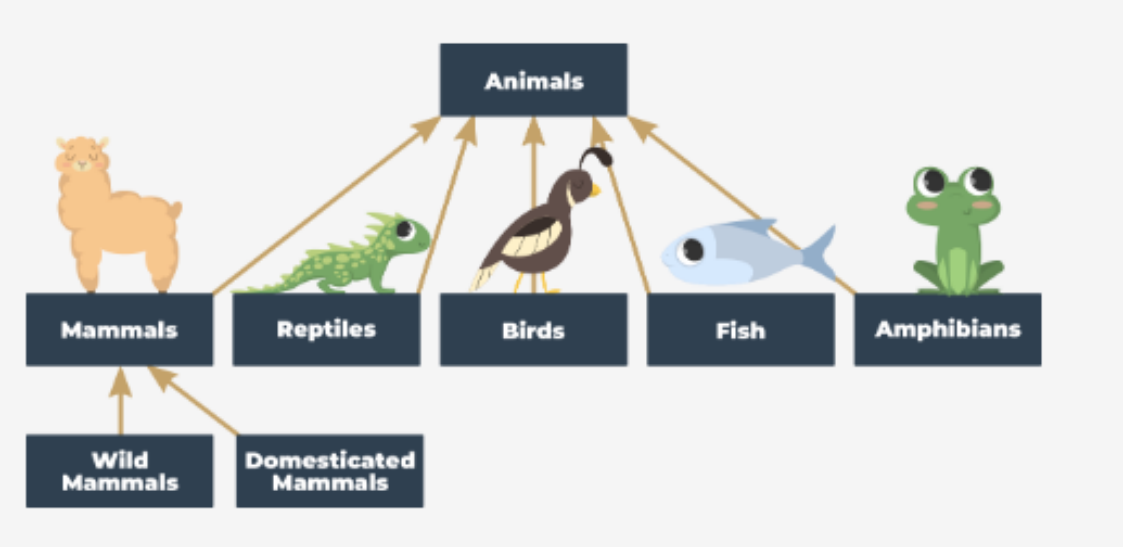
Another example is the hierarchy of the taxonomic kingdom of animals.

We can say that all animals (our top-level class) can be divided into five subclasses:

- mammals;
- reptiles;
- birds;
- fish;
- amphibians. We'll take the first one for further analysis.

We have identified the following subclasses:

- wild mammals;
- domesticated mammals.



1.2 What is an object?

A class (among other definitions) is a **set of objects**. An object is **a being belonging to a class**.

An object is an **incarnation of the requirements, traits, and qualities assigned to a specific class**. This may sound simple, but note the following important circumstances. Classes form a hierarchy.

This may mean that an object belonging to a specific class belongs to all the superclasses at the same time. It may also mean that any object belonging to a superclass may not belong to any of its subclasses.

For example: any personal car is an object belonging to the wheeled vehicles class. It also means that the same car belongs to all superclasses of its home class; therefore, it is a member of the vehicles class, too.

Your dog (or your cat) is an object included in the domesticated mammals class, which explicitly means that it is included in the animals class as well.

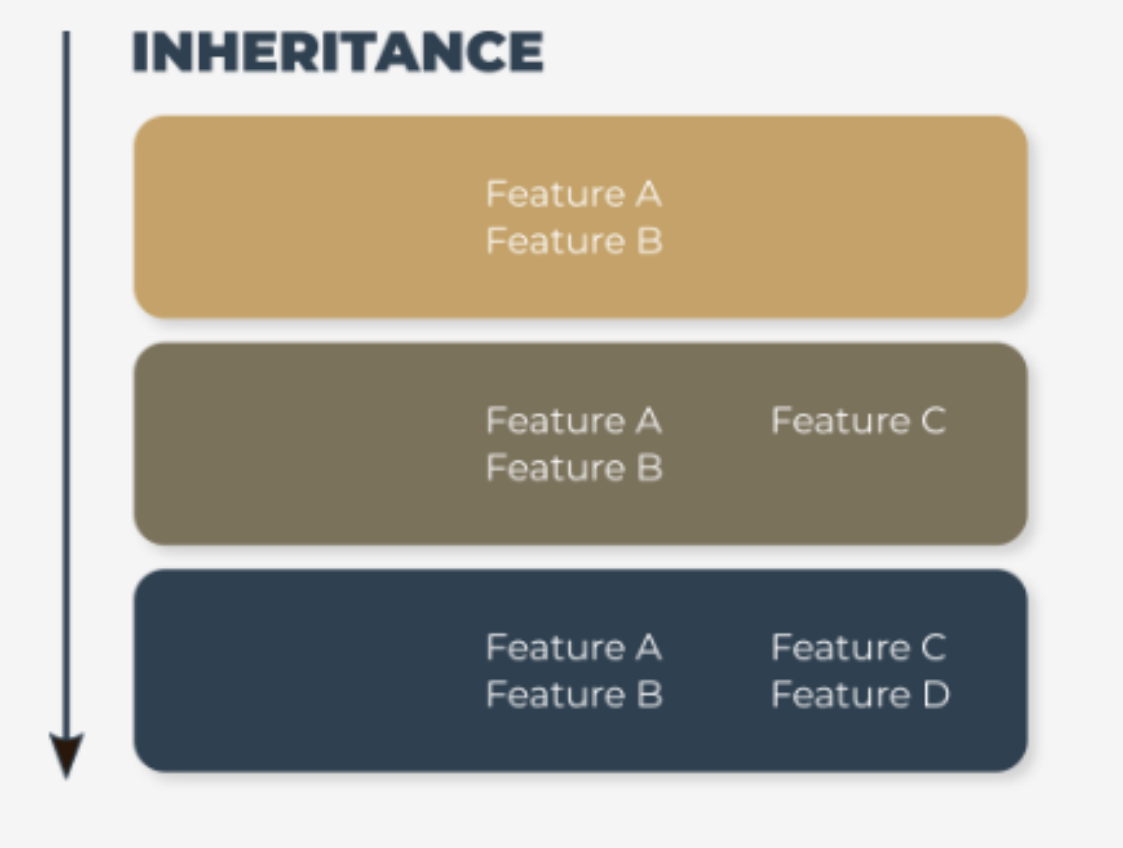
Each subclass is more specialized (or more specific) than its superclass. Conversely, each **superclass is more general** (more abstract) than any of its subclasses.

Note that we've presumed that a class may only have one superclass - this is not always true, but we'll discuss this issue more a bit later.

1.2.1 Inheritance

Let's define one of the fundamental concepts of object programming, named **inheritance**. Any object bound to a specific level of a class hierarchy **inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses**.

The object's home class may define new traits (as well as requirements and qualities) which will be inherited by any of its subclasses.



1.2.2 What does an object have?

The object programming convention assumes that **every existing object may be equipped with three groups of attributes**:

- an object has a **name** that uniquely identifies it within its home namespace (although there may be some anonymous objects, too)
- an object has a **set of individual properties** which make it original, unique, or outstanding (although it's possible that some objects may have no properties at all)
- an object has a **set of abilities to perform specific activities**, able to change the object itself, or some of the other objects. There is a hint (although this doesn't always work) which can help you identify any of the three spheres above. Whenever you describe an object and you use:
- a noun – you probably define the object's name;
- an adjective – you probably define the object's property;
- a verb – you probably define the object's activity. Two sample phrases should serve as a good example:
- A pink Cadillac went quickly.

Object name = Cadillac

Home class = Wheeled vehicles

Property = Color (pink)

Activity = Go (quickly)

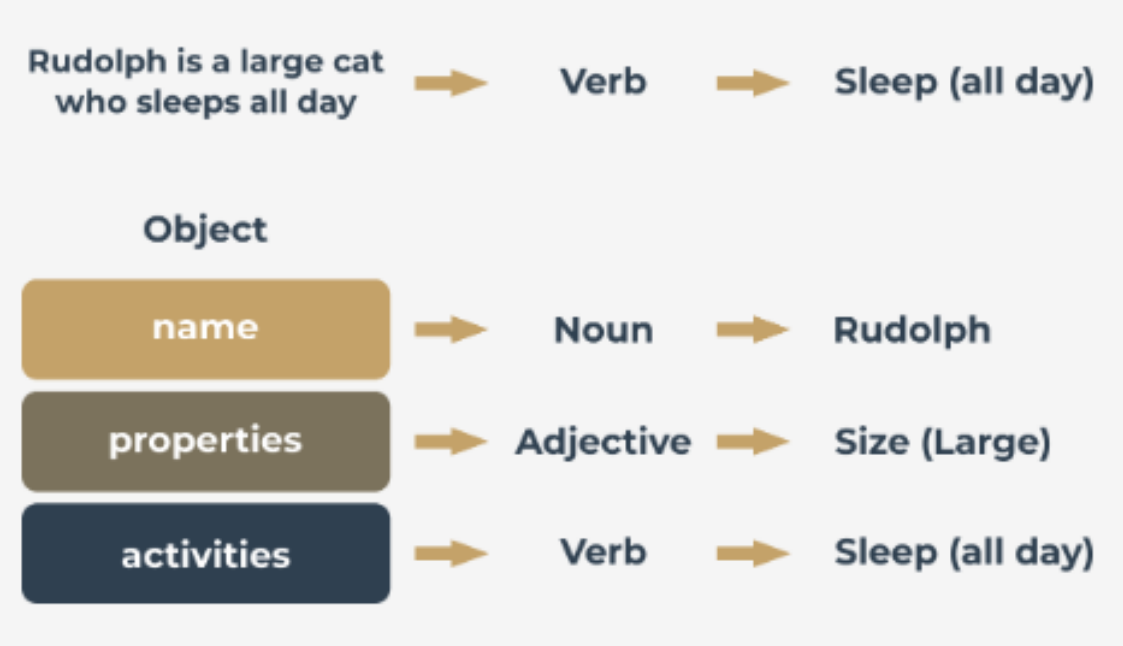
- Rudolph is a large cat who sleeps all day.

Object name = Rudolph

Home class = Cat

Property = Size (large)

Activity = Sleep (all day)



1.2.3 Your first class

Object programming is **the art of defining and expanding classes**. A class is a model of a very specific part of reality, reflecting properties and activities found in the real world.

The classes defined at the beginning are too general and imprecise to cover the largest possible number of real cases.

There's no obstacle to defining new, more precise subclasses. They'll inherit everything from their superclass, so the work that went into its creation isn't wasted.

The new class may add new properties and new activities, and therefore may be more useful in specific applications. Obviously, it may be used as a superclass for any number of newly created subclasses.

The process doesn't need to have an end. You can create as many classes as you need.

The class you define has nothing to do with the object: **the existence of a class does not mean that any of the compatible objects will automatically be created**. The class itself isn't able to create an object - you have to create it yourself, and Python allows you to do this.

It's time to define the simplest class and to create an object. Take a look at the example below:

[1]:
```
class TheSimplestClass:
    pass
```

We've defined a class there. The class is rather poor: it has neither properties nor activities. It's **empty**, actually, but that doesn't matter for now. The simpler the class, the better for our purposes.

The definition begins with the keyword`class`. The keyword is followed by an **identifier which will name the class** (note: don't confuse it with the object's name - these are two

different things).

Next, you add a colon (:), as classes, like functions, form their own nested block. The content inside the block define all the class's properties and activities.

The `pass` keyword fills the class with nothing. It doesn't contain any methods or properties.

The newly defined class becomes a tool that is able to create new objects. The tool has to be used explicitly, on demand.

Imagine that you want to create one (exactly one) object of the `TheSimplestClass` class.

To do this, you need to assign a variable to store the newly created object of that class, and create an object at the same time.

You do it in the following way:

[2]:
```
my_first_object = TheSimplestClass()
my_first_object
```

[2]:
```
<__main__.TheSimplestClass at 0x7fb6f0144220>
```

Note:

- the class name tries to pretend that it's a function - can you see this? We'll discuss it soon;
- the newly created object is equipped with everything the class brings; as this class is completely empty, the object is empty, too. The act of creating an object of the selected class is also called an **instantiation** (as the object becomes an **instance of the class**).

The stack - the procedural approach First, you have to decide how to store the values which will arrive onto the stack. We suggest using the simplest of methods, and employing a list for this job. Let's assume that the size of the stack is not limited in any way. Let's also assume that the last element of the list stores the top element.

The stack itself is already created:

```
stack=[]
```

We're ready to define a **function that puts a value onto the stack**. Here are the presuppositions for it:

- the name for the function is push‘;
- the function gets one parameter (this is the value to be put onto the stack)
- the function returns nothing;
- the function appends the parameter's value to the end of the stack;

[3]:
```
stack = []


def push(val):
    stack.append(val)


def pop():
```

```
    val = stack[-1]
    del stack[-1]
    return val


push(3)
push(2)
push(1)

print(pop())
print(pop())
print(pop())
```

```
1
2
3
```

1.2.4 The stack - the procedural approach vs. the object-oriented approach

The procedural stack is ready. Of course, there are some weaknesses, and the implementation could be improved in many ways (harnessing exceptions to work is a good idea), but in general the stack is fully implemented, and you can use it if you need to.

But the more often you use it, the more disadvantages you'll encounter. Here are some of them:

- the essential variable (the stack list) is highly **vulnerable**; anyone can modify it in an uncontrollable way, destroying the stack, in effect; this doesn't mean that it's been done maliciously - on the contrary, it may happen as a result of carelessness, e.g., when somebody confuses variable names; imagine that you have accidentally written something like this:

```
stack[0] = 0
```

The functioning of the stack will be completely disorganized;

- it may also happen that one day you need more than one stack; you'll have to create another list for the stack's storage, and probably other push and pop functions tooit may also happen that one day you need more than one stack; you'll have to create another list for the stack's storage, and probably other `push` and `pop` functions too';
- it may also happen that you need not only `push` and `pop` functions, but also some other conveniences; you could certainly implement them, but try to imagine what would happen if you had dozens of separately implemented stacksit may also happen that you need not only push and pop functions, but also some other conveniences; you could certainly implement them, but try to imagine what would happen if you had dozens of separately implemented stacks.

The objective approach delivers solutions for each of the above problems. Let's name them first:

- the ability to hide (protect) selected values against unauthorized access is called **encapsulation**; the encapsulated values can be neither **accessed nor modified if you want to use them exclusively**;

- when you have a class implementing all the needed stack behaviors, you can produce as many stacks as you want; you needn't copy or replicate any part of the code;
- the ability to enrich the stack with new functions comes from inheritance; you can create a new class (a subclass) which inherits all the existing traits from the superclass, and adds some new ones.

1.2.5 The stack - the object approach

Of course, the main idea remains the same. We'll use a list as the stack's storage. We only have to know how to put the list into the class.

Let's start from the absolute beginning - this is how the objective stack begins:

```
class Stack:
```

Now, we expect two things from it:

- we want the class to have **one property as the stack's storage** - we have to **"install" a list inside each object of the class** (note: each object has to have its own list - the list mustn't be shared among different stacks)
- then, we want **the list to be hidden** from the class users' sight. How is this done?

In contrast to other programming languages, Python has no means of allowing you to declare such a property just like that.

Instead, you need to add a specific statement or instruction. The properties have to be added to the class manually.

How do you guarantee that such an activity takes place every time the new stack is created?

There is a simple way to do it - you have to **equip the class with a specific function** - its specificity is dual:

- it has to be named in a strict way;
- it is invoked implicitly, when the new object is created. Such a function is called a **constructor**, as its general purpose is to construct a new object. The constructor should know everything about the object's structure, and must perform all the needed initializations.

Let's add a very simple constructor to the new class. Take a look at the snippet:

[4]:
```
class Stack:  # Defining the Stack class
    def __init__(self):  # Defining the constructor function.
        print("Hi!")


stack_object = Stack()  # Instantiating the object.
```

```
Hi!
```

And now:

- the constructor's name is always `__init__`;
- it has to have **at least one parameter** (we'll discuss this later); the parameter is used to represent the newly created object - you can use the parameter to manipulate the object, and to enrich it with the needed properties; you'll make use of this soon;
- note: the obligatory parameter is usually named `self` - it's only **a convention, but you should follow it** - it simplifies the process of reading and understanding your code.

Note - there is no trace of invoking the constructor inside the code. It has been invoked implicitly and automatically. Let's make use of that now.

Any change you make inside the constructor that modifies the state of the `self` parameter will be reflected in the newly created object.

This means you can add any property to the object and the property will remain there until the object finishes its life or the property is explicitly removed.

Now let's **add just one property to the new object** – a list for a stack. We'll name it `stack_list`.

[5]:
```
class Stack:
    def __init__(self):
        self.stack_list = []


stack_object = Stack()
print(len(stack_object.stack_list))
```

```
0
```

Note:

- we've used **dot notation**, just like when invoking methods; this is the general convention for accessing an object's properties you need to name the object, put a dot (.) after it, and specify the desired property's name; don't use parentheses! You don't want to invoke a method – you want to **access a property**;
- if you set a property's value for the very first time (like in the constructor), you are creating it; from that moment on, the object has got the property and is ready to use its value;
- we've done something more in the code – we've tried to access the `stack_list` property from outside the class immediately after the object has been created; we want to check the current length of the stack

Take a look - we've added two underscores before the `stack_list` name - nothing more‘

[6]:
```
class Stack:
    def __init__(self):
        self.__stack_list = []


stack_object = Stack()
print(len(stack_object.__stack_list))
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/var/folders/cg/78_rndgn2rg8zjdhvdvq_xrm0000gn/T/ipykernel_35185/1119582015.py␣
 ↪in <module>
      5
      6 stack_object = Stack()
----> 7 print(len(stack_object.__stack_list))

AttributeError: 'Stack' object has no attribute '__stack_list'
```

The change invalidates the program.

Why?

When any class component has a **name starting with two underscores** (___), **it becomes private** - this means that it can be accessed only from within the class.

You cannot see it from the outside world. This is how Python implements the encapsulation concept.

An AttributeError exception should be raised

Now it's time for the two functions (methods) implementing the `push` and `pop` operations. Python assumes that a function of this kind (a class activity) should be'**immersed inside the class body** - just like a constructor.

We want to invoke these functions to `push` and `pop` values. This means that they should both be accessible to every class's user (in contrast to the previously constructed list, which is hidden from the ordinary class's users).

Such a component is called **public**, so you **can't begin its name with two (or more) underscores**. There is one more requirement - **the name must have no more than one trailing underscore**. As no trailing underscores at all fully meets the requirement, you can assume that the name is acceptable.

The functions themselves are simple. Take a look:

[7]:
```
class Stack:
    def __init__(self):
        self.__stack_list = []


    def push(self, val):
        self.__stack_list.append(val)


    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val
```

```
stack_object = Stack()

stack_object.push(3)
stack_object.push(2)
stack_object.push(1)

print(stack_object.pop())
print(stack_object.pop())
print(stack_object.pop())
```

```
1
2
3
```

However, there's something really strange in the code. The functions look familiar, but they have more parameters than their procedural counterparts.

Here, both functions have a parameter named `self` at the first position of the parameters list.

Is it needed? Yes, it is.

All methods have to have this parameter. It plays the same role as the first constructor parameter.

It allows the method to access entities (properties and activities/methods) carried out by the actual object. You cannot omit it. Every time Python invokes a method, it implicitly sends the current object as the first argument.

This means that a **method is obligated to have at least one parameter, which is used by Python itself** - you don't have any influence on it.

If your method needs no parameters at all, this one must be specified anyway. If it's designed to process just one parameter, you have to specify two, and the first one's role is still the same.

There is one more thing that requires explanation - the way in which methods are invoked from within the `__stack_list` variable.

Fortunately, it's much simpler than it looks:

- the first stage delivers the object as a whole → `self`;
- next, you need to get to the `__stack_list list` → `self.__stack_list`;
- with `__stack_list` ready to be used, you can perform the third and last step →`self.__stack_list.append(val)`.

Having such a class opens up some new possibilities. For example, you can now have more than one stack behaving in the same way. Each stack will have its own copy of private data, but will utilize the same set of methods.

This is exactly what we want for this example.

[8]:
```
class Stack:
    def __init__(self):
```

```
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


stack_object_1 = Stack()
stack_object_2 = Stack()

stack_object_1.push(3)
stack_object_2.push(stack_object_1.pop())

print(stack_object_2.pop())
```

```
3
```

There are **two stacks created from the same base class**. They work **independently**. You can make more of them if you want to.

Analyze the snippet below - we've created three objects of the class Stack. Next, we've juggled them up. Try to predict the value outputted to the screen.

[9]:
```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


little_stack = Stack()
another_stack = Stack()
funny_stack = Stack()

little_stack.push(1)
another_stack.push(little_stack.pop() + 1)
funny_stack.push(another_stack.pop() - 2)
```

```
print(funny_stack.pop())
```

```
0
```

Now let's go a little further. Let's **add a new class for handling stacks**.

The new class should be able to **evaluate the sum of all the elements currently stored on the stack**.

We don't want to modify the previously defined stack. It's already good enough in its applications, and we don't want it changed in any way. We want a new stack with new capabilities. In other words, we want to construct a subclass of the already existing Stack class.

The first step is easy: just **define a new subclass pointing to the class which will be used as the superclass**.

This is what it looks like:

[10]:
```
class AddingStack(Stack):
    pass
```

The class doesn't define any new component yet, but that doesn't mean that it's empty. **It gets all the components defined by its superclass - the name of the superclass** is written before the colon directly after the new class name.

This is what we want from the new stack:

- we want the `push` method not only to push the value onto the stack but also to add the value to the `sum` variable;
- we want the `pop` function not only to pop the value off the stack but also to subtract the value from the `sum` variable.

Firstly, let's add a new variable to the class. It'll be a **private variable**, like the stack list. We don't want anybody to manipulate the `sum` value.

As you already know, adding a new property to the class is done by the constructor. You already know how to do that, but there is something really intriguing inside the constructor. Take a look:

[17]:
```
class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0
```

The second line of the constructor's body creates a property named `__sum` - it will store the total of all the stack's values.

But the line before it looks different. What does it do? Is it really necessary? Yes, it is.

Contrary to many other languages, Python forces you to **explicitly invoke a superclass's constructor**. Omitting this point will have harmful effects - the object will be deprived of the `__stack_list` list. Such a stack will not function properly.

This is the only time you can invoke any of the available constructors explicitly - it can be done inside the subclass's constructor.

Note the syntax:

- you specify the superclass's name (this is the class whose constructor you want to run)
- you put a dot (.) after it;
- you specify the name of the constructor;
- you have to point to the object (the class's instance) which has to be initialized by the constructor - this is why you have to specify the argument and use the `self` variable here; note: invoking any method (including constructors) from outside the class never requires you to put the `self` argument at the argument's list - invoking a method from within the class demands explicit usage of the `self` argument, and it has to be put first on the list.

Note: it's generally a recommended practice to invoke the superclass's constructor before any other initializations you want to perform inside the subclass. This is the rule we have followed in the snippet.

[15]:
```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

    def push(self, val):
        self.__sum += val
        Stack.push(self, val)
```

Secondly, let's add two methods. But let us ask you: is it really adding? We have these methods in the superclass already. Can we do something like that?

Yes, we can. It means that we're going to **change the functionality of the methods, not their names**. We can say more precisely that the interface (the way in which the objects are handled) of the class remains the same when changing the implementation at the same time.

Let's start with the implementation of the `push` function. This is what we expect from it:

- to add the value to the `__sum` variable to add the value to the ___sum variable;
- to push the value onto the stack.

Note: the second activity has already been implemented inside the superclass - so we can use that. Furthermore, we have to use it, as there's no other way to access the `__stackList` variable.

This is how the ‘push method looks in the subclass:

[18]:
```
def push(self, val):
    self.__sum += val
    Stack.push(self, val)
```

Note the way we’ve invoked the previous implementation of the `push` method (the one available in the superclass):

- we have to specify the superclass’s name; this is necessary in order to clearly indicate the class containing the method, to avoid confusing it with any other function of the same namewe have to specify the superclass’s name; this is necessary in order to clearly indicate the class containing the method, to avoid confusing it with any other function of the same name‘;
- we have to specify the target object and to pass it as the first argument (it’s not implicitly added to the invocation in this context.)

We say that the `push` method has been‘**overridden** - the same name as in the superclass now represents a different functionality.

This is the new pop function:

[19]:
```
def pop(self):
    val = Stack.pop(self)
    self.__sum -= val
    return val
```

So far, we’ve defined the `__sum` variable, but we haven’t provided a method to get its value. It seems to be hidden. How can we reveal it and do it in a way that still protects it from modifications?

We have to define a new method. We’ll name it `get_sum`. Its only task will be to **return the `__sum` value**.

Here it is:

[20]:
```
def get_sum(self):
        return self.__sum
```

[13]:
```
## Complete code

class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
```

```
        return val


class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

    def get_sum(self):
        return self.__sum

    def push(self, val):
        self.__sum += val
        Stack.push(self, val)

    def pop(self):
        val = Stack.pop(self)
        self.__sum -= val
        return val


stack_object = AddingStack()

for i in range(5):
    stack_object.push(i)
print(stack_object.get_sum())

for i in range(5):
    print(stack_object.pop())
```

```
10
4
3
2
1
0
```

As you can see, we add five subsequent values onto the stack, print their sum, and take them all off the stack.

1.2.6 Scenario

We've showed you recently how to extend Stack possibilities by defining a new class (i.e., a subclass) which retains all inherited traits and adds some new ones.

Your task is to extend the Stack class behavior in such a way so that the class is able to count all the elements that are pushed and popped (we assume that counting pops is enough). Use the Stack class we've provided in the editor.

Follow the hints:

- introduce a property designed to count pop operations and name it in a way which guarantees hiding it;
- initialize it to zero inside the constructor;
- provide a method which returns the value currently assigned to the counter (name it get_counter()).
- Complete the code in the editor. Run it to check whether your code outputs 100.

[14]:
```
class Stack:
    def __init__(self):
        self.__stk = []

    def push(self, val):
        self.__stk.append(val)

    def pop(self):
        val = self.__stk[-1]
        del self.__stk[-1]
        return val


class CountingStack(Stack):
    def __init__(self):
        super().__init__()           # Inicializa la pila base
        self.__counter = 0           # Atributo privado que cuenta pops

    def get_counter(self):
        return self.__counter        # Devuelve el número de pops

    def pop(self):
        val = super().pop()          # Usa el pop de la clase base
        self.__counter += 1          # Incrementa el contador
        return val


# Prueba del comportamiento
stk = CountingStack()
for i in range(100):
    stk.push(i)
    stk.pop()

print(stk.get_counter())  # Resultado esperado: 100
```

```
100
```

[ ]: