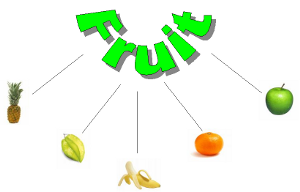
**General Introduction**



Though Python is an object-oriented language without fuss or quibble, we have so far intentionally avoided the treatment of object-oriented programming (OOP) in the previous chapters of our Python tutorial. We skipped OOP, because we are convinced that it is easier and more fun to start learning Python without having to know about all the details of object-oriented programming.

Even though we have avoided OOP, it has nevertheless always been present in the exercises and examples of our course. We used objects and methods from classes without properly explaining their OOP background. In this chapter, we will catch up on what has been missing so far. We will provide an introduction into the principles of object oriented programming in general and into the specifics of the OOP approach of Python. OOP is one of the most powerful tools of Python, but nevertheless you don't have to use it, i.e. you can write powerful and efficient programs without it as well.

Though many computer scientists and programmers consider OOP to be a modern programming paradigm, the roots go back to 1960s. The first programming language to use objects was Simula 67. As the name implies, Simula 67 was introduced in the year 1967. A major breakthrough for object-oriented programming came with the programming language Smalltalk in the 1970s.

You will learn to know the four major principles of object-orientation and the way Python deals with them in the next section of this tutorial on object-oriented programming:



* Encapsulation
* Data Abstraction
* Polymorphism
* Inheritance

Before we start the section about the way OOP is used in Python, we want to give you a general idea about object-oriented programming. For this purpose, we would like to draw your attention to a public library. Let's think about a huge one, like the "British Library" in London or the "New York Public Library" in New York. If it helps, you can imagine the libraries in Paris, Berlin, Ottawa or Toronto\* as well. Each of these contain an organized collection of books, periodicals, newspapers, audiobooks, films and so on.

Generally, there are two opposed ways of keeping the stock in a library. You can use a "closed access" method that is the stock is not displayed on open shelves. In this system, trained staff brings the books and other publications to the users on demand. Another way of running a library is open-access shelving, also known as "open shelves". "Open" means open to all users of the library not only specially trained staff. In this case the books are openly displayed. Imperative languages like C could be seen as open-access shelving libraries. The user can do everything. It's up to the user to find the books and to put them back at the right shelf. Even though this is great for the user, it might lead to serious problems in the long run. For example some books will be misplaced, so it's hard to find them again. As you may have guessed already, "closed access" can be compared to object oriented programming. The analogy can be seen like this: The books and other publications, which a library offers, are like the data in an object-oriented program. Access to the books is restricted like access to the data is restricted in OOP. Getting or returning a book is only possible via the staff. The staff functions like the methods in OOP, which control the access to the data. So, the data, - often called attributes, - in such a program can be seen as being hidden and protected by a shell, and it can only be accessed by special functions, usually called methods in the OOP context. Putting the data behind a "shell" is called *Encapsulation*. So a library can be regarded as a class and a book is an instance or an object of this class. Generally speaking, an object is defined by a class. A class is a formal description of how an object is designed, i.e. which attributes and methods it has. These objects are called instances as well. The expressions are in most cases used synonymously. A class should not be confused with an object.

**OOP in Python**



Even though we haven't talked about classes and object orientation in previous chapters, we have worked with classes all the time. In fact, everything is a class in Python. Guido van Rossum has designed the language according to the principle "first-class everything". He wrote: "One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth." (Blog, The History of Python, February 27, 2009) In other words, "everything" is treated the same way, everything is a class: functions and methods are values just like lists, integers or floats. Each of these are instances of their corresponding classes.

x = 42

type(x)

Output: :

int

y = 4.34

type(y)

Output: :

float

**def** f(x):

**return** x + 1

type(f)

Output: :

function

**import** **math**

type(math)

Output: :

module

One of the many integrated classes in Python is the list class, which we have quite often used in our exercises and examples. The list class provides a wealth of methods to build lists, to access and change elements, or to remove elements:

x = [3,6,9]

y = [45, "abc"]

print(x[1])

6

x[1] = 99

x.append(42)

last = y.pop()

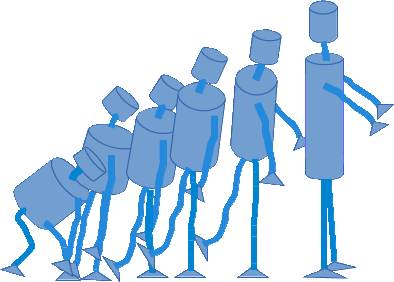
print(last)

abc

The variables x and y of the previous example denote two instances of the list class. In simplified terms, we have said so far that "x and y are lists". We will use the terms "object" and "instance" synonymously in the following chapters, as it is often done\*\*.

pop and append of the previous example are methods of the list class. pop returns the top (or you might think of it as the "rightest") element of the list and removes this element from the list. We will not explain how Python has implemented lists internally. We don't need this information, because the list class provides us with all the necessary methods to access the data indirectly. That is, the encapsulation details are encapsulated. We will learn about encapsulation later.

**A Minimal Class in Python**



We will design and use a robot class in Python as an example to demonstrate the most important terms and ideas of object orientation. We will start with the simplest class in Python.

class Robot:

pass

We can realize the fundamental syntactical structure of a class in Python: A class consists of two parts: the header and the body. The header usually consists of just one line of code. It begins with the keyword "class" followed by a blank and an arbitrary name for the class. The class name is "Robot" in our case. The class name is followed by a listing of other class names, which are classes from which the defined class inherits. These classes are called superclasses, base classes or sometimes parent classes. If you look at our example, you will see that this listing of superclasses is not obligatory. You don't have to bother about inheritance and superclasses for now. We will introduce them later.

The body of a class consists of an indented block of statements. In our case a single statement, the "pass" statement.

A class object is created, when the definition is left normally, i.e. via the end. This is basically a wrapper around the contents of the namespace created by the class definition.

It's hard to believe, especially for C++ or Java programmers, but we have already defined a complete class with just three words and two lines of code. We are capable of using this class as well:

**class** **Robot**:

**pass**

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot()

y = Robot()

y2 = y

print(y == y2)

print(y == x)

True

False

We have created two different robots x and y in our example. Besides this, we have created a reference y2 to y, i.e. y2 is an alias name for y.

**Attributes**

Those who have learned already another object-oriented language, must have realized that the terms attributes and properties are usually used synonymously. It may even be used in the definition of an attribute, like Wikipedia does: "In computing, an attribute is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance of such."

Even in normal English usage the words "attribute" and "property" can be used in some cases as synonyms. Both can have the meaning "An attribute, feature, quality, or characteristic of something or someone". Usually an "attribute" is used to denote a specific ability or characteristic which something or someone has, like black hair, no hair, or a quick perception, or "her quickness to grasp new tasks". So, think a while about your outstanding attributes. What about your "outstanding properties"? Great, if one of your strong points is your ability to quickly understand and adapt to new situations! Otherwise, you would not be learning Python!

Let's get back to Python: We will learn later that properties and attributes are essentially different things in Python. This subsection of our tutorial is about attributes in Python. So far our robots have no attributes. Not even a name, like it is customary for ordinary robots, isn't it? So, let's implement a name attribute. "type designation", "build year" etc. are easily conceivable as further attributes as well\*\*\*.

Attributes are created inside a class definition, as we will soon learn. We can dynamically create arbitrary new attributes for existing instances of a class. We do this by joining an arbitrary name to the instance name, separated by a dot ".". In the following example, we demonstrate this by creating an attribute for the name and the year built:

**class** **Robot**:

**pass**

x = Robot()

y = Robot()

x.name = "Marvin"

x.build\_year = "1979"

y.name = "Caliban"

y.build\_year = "1993"

print(x.name)

Marvin

print(y.build\_year)

1993

As we have said before: This is not the way to properly create instance attributes. We introduced this example, because we think that it may help to make the following explanations easier to understand.

If you want to know, what's happening internally: The instances possess dictionaries \_\_dict\_\_, which they use to store their attributes and their corresponding values:

x.\_\_dict\_\_

Output: :

{'name': 'Marvin', 'build\_year': '1979'}

y.\_\_dict\_\_

Output: :

{'name': 'Caliban', 'build\_year': '1993'}

Attributes can be bound to class names as well. In this case, each instance will possess this name as well. Watch out, what happens, if you assign the same name to an instance:

**class** **Robot**(object):

**pass**

x = Robot()

Robot.brand = "Kuka"

x.brand

Output: :

'Kuka'

x.brand = "Thales"

Robot.brand

Output: :

'Kuka'

y = Robot()

y.brand

Output: :

'Kuka'

Robot.brand = "Thales"

y.brand

Output: :

'Thales'

x.brand

Output: :

'Thales'

If you look at the \_\_dict\_\_ dictionaries, you can see what's happening.

x.\_\_dict\_\_

Output: :

{'brand': 'Thales'}

y.\_\_dict\_\_

Output: :

{}

Robot.\_\_dict\_\_

Output: :

mappingproxy({'\_\_module\_\_': '\_\_main\_\_',

'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Robot' objects>,

'\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Robot' objects>,

'\_\_doc\_\_': None,

'brand': 'Thales'})

If you try to access y.brand, Python checks first, if "brand" is a key of the y. \_\_dict\_\_ dictionary. If it is not, Python checks, if "brand" is a key of the Robot. \_\_dict\_\_. If so, the value can be retrieved.

If an attribute name is not in included in either of the dictionary, the attribute name is not defined. If you try to access a non-existing attribute, you will raise an AttributeError:

x.energy

**---------------------------------------------------------------------------**

**AttributeError** Traceback (most recent call last)

**<ipython-input-28-82fa0f11497d>** in <module>

**----> 1** x**.**energy

**AttributeError**: 'Robot' object has no attribute 'energy'

By using the function getattr, you can prevent this exception, if you provide a default value as the third argument:

getattr(x, 'energy', 100)

Output: :

100

Binding attributes to objects is a general concept in Python. Even function names can be attributed. You can bind an attribute to a function name in the same way, we have done so far to other instances of classes:

**def** f(x):

**return** 42

f.x = 42

print(f.x)

42

This can be used as a replacement for the static function variables of C and C++, which are not possible in Python. We use a counter attribute in the following example:

**def** f(x):

f.counter = getattr(f, "counter", 0) + 1

**return** "Monty Python"

**for** i **in** range(10):

f(i)

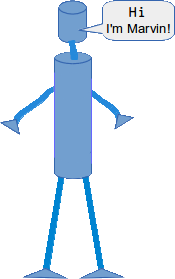
print(f.counter)

10

Some uncertainty may arise at this point. It is possible to assign attributes to most class instances, but this has nothing to do with defining classes. We will see soon how to assign attributes when we define a class.

To properly create instances of classes, we also need methods. You will learn in the following subsection of our Python tutorial, how you can define methods.

**Methods**



Methods in Python are essentially functions in accordance with Guido's saying "first-class everything".

Let's define a function "hi", which takes an object "obj" as an argument and assumes that this object has an attribute "name". We will also define our basic Robot class again:

**def** hi(obj):

print("Hi, I am " + obj.name + "!")

**class** **Robot**:

**pass**

x = Robot()

x.name = "Marvin"

hi(x)

Hi, I am Marvin!

We will now bind the function „hi“ to a class attribute „say\_hi“!

**def** hi(obj):

print("Hi, I am " + obj.name)

**class** **Robot**:

say\_hi = hi

x = Robot()

x.name = "Marvin"

Robot.say\_hi(x)

Hi, I am Marvin

"say\_hi" is called a method. Usually, it will be called like this:

x.say\_hi()

It is possible to define methods like this, but you shouldn't do it.

The proper way to do it:

* Instead of defining a function outside of a class definition and binding it to a class attribute, we define a method directly inside (indented) of a class definition.
* A method is "just" a function which is defined inside a class.
* The first parameter is used a reference to the calling instance.
* This parameter is usually called self.
* Self corresponds to the Robot object x.

We have seen that a method differs from a function only in two aspects:

* It belongs to a class, and it is defined within a class
* The first parameter in the definition of a method has to be a reference to the instance, which called the method. This parameter is usually called "self".

As a matter of fact, "self" is not a Python keyword. It's just a naming convention! So C++ or Java programmers are free to call it "this", but this way they are risking that others might have greater difficulties in understanding their code!

Most other object-oriented programming languages pass the reference to the object (self) as a hidden parameter to the methods.

You saw before that the calls Robot.say\_hi(x)". and "x.say\_hi()" are equivalent. "x.say\_hi()" can be seen as an "abbreviated" form, i.e. Python automatically binds it to the instance name. Besides this "x.say\_hi()" is the usual way to call methods in Python and in other object oriented languages.

For a Class C, an instance x of C and a method m of C the following three method calls are equivalent:

* type(x).m(x, ...)
* C.m(x, ...)
* x.m(...)

Before you proceed with the following text, you may mull over the previous example for awhile. Can you figure out, what is wrong in the design?

There is more than one thing about this code, which may disturb you, but the essential problem at the moment is the fact that we create a robot and that after the creation, we shouldn't forget about naming it! If we forget it, say\_hi will raise an error.

We need a mechanism to initialize an instance right after its creation. This is the \_\_init\_\_-method, which we cover in the next section.

**The \_\_init\_\_ Method**

We want to define the attributes of an instance right after its creation. \_\_init\_\_ is a method which is immediately and automatically called after an instance has been created. This name is fixed and it is not possible to chose another name. \_\_init\_\_ is one of the so-called magic methods,we will get to know it with some more details later. The \_\_init\_\_ method is used to initialize an instance. There is no explicit constructor or destructor method in Python, as they are known in C++ and Java. The \_\_init\_\_ method can be anywhere in a class definition, but it is usually the first method of a class, i.e. it follows right after the class header.

**class** **A**:

**def** \_\_init\_\_(self):

print("\_\_init\_\_ has been executed!")

x = A()

\_\_init\_\_ has been executed!

We add an \_\_init\_\_-method to our robot class:

**class** **Robot**:

**def** \_\_init\_\_(self, name=**None**):

self.name = name

**def** say\_hi(self):

**if** self.name:

print("Hi, I am " + self.name)

**else**:

print("Hi, I am a robot without a name")

x = Robot()

x.say\_hi()

y = Robot("Marvin")

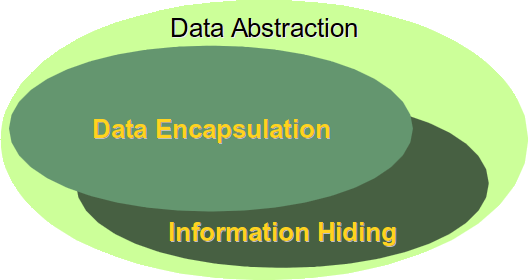
y.say\_hi()

Hi, I am a robot without a name

Hi, I am Marvin

**Data Abstraction, Data Encapsulation, and Information Hiding**

**Definitions of Terms**



Data Abstraction, Data Encapsulation and Information Hiding are often synonymously used in books and tutorials on OOP. However, there is a difference. Encapsulation is seen as the bundling of data with the methods that operate on that data. Information hiding on the other hand is the principle that some internal information or data is "hidden", so that it can't be accidentally changed. Data encapsulation via methods doesn't necessarily mean that the data is hidden. You might be capable of accessing and seeing the data anyway, but using the methods is recommended. Finally, data abstraction is present, if both data hiding and data encapsulation is used. In other words, data abstraction is the broader term:

*Data Abstraction = Data Encapsulation + Data Hiding*

Encapsulation is often accomplished by providing two kinds of methods for attributes: The methods for retrieving or accessing the values of attributes are called getter methods. Getter methods do not change the values of attributes, they just return the values. The methods used for changing the values of attributes are called setter methods.

We will define now a Robot class with a Getter and a Setter for the name attribute. We will call them get\_name and set\_name accordingly.

**class** **Robot**:

**def** \_\_init\_\_(self, name=**None**):

self.name = name

**def** say\_hi(self):

**if** self.name:

print("Hi, I am " + self.name)

**else**:

print("Hi, I am a robot without a name")

**def** set\_name(self, name):

self.name = name

**def** get\_name(self):

**return** self.name

x = Robot()

x.set\_name("Henry")

x.say\_hi()

y = Robot()

y.set\_name(x.get\_name())

print(y.get\_name())

Hi, I am Henry

Henry

Before you go on, you can do a little exercise. You can add an additional attribute "build\_year" with Getters and Setters to the Robot class.

**class** **Robot**:

**def** \_\_init\_\_(self,

name=**None**,

build\_year=**None**):

self.name = name

self.build\_year = build\_year

**def** say\_hi(self):

**if** self.name:

print("Hi, I am " + self.name)

**else**:

print("Hi, I am a robot without a name")

**if** self.build\_year:

print("I was built in " + str(self.build\_year))

**else**:

print("It's not known, when I was created!")

**def** set\_name(self, name):

self.name = name

**def** get\_name(self):

**return** self.name

**def** set\_build\_year(self, by):

self.build\_year = by

**def** get\_build\_year(self):

**return** self.build\_year

x = Robot("Henry", 2008)

y = Robot()

y.set\_name("Marvin")

x.say\_hi()

y.say\_hi()

Hi, I am Henry

I was built in 2008

Hi, I am Marvin

It's not known, when I was created!

There is still something wrong with our Robot class. The Zen of Python says: "There should be one-- and preferably only one --obvious way to do it." Our Robot class provides us with two ways to access or to change the "name" or the "build\_year" attribute. This can be prevented by using private attributes, which we will explain later.

**\_\_str\_\_- and \_\_repr\_\_-Methods**

We will have a short break in our treatise on data abstraction for a quick side-trip. We want to introduce two important magic methods "\_\_str\_\_" and "\_\_repr\_\_", which we will need in future examples. In the course of this tutorial, we have already encountered the \_\_str\_\_ method. We had seen that we can depict various data as string by using the str function, which uses "magically" the internal \_\_str\_\_ method of the corresponding data type. \_\_repr\_\_ is similar. It also produces a string representation.

l = ["Python", "Java", "C++", "Perl"]

print(l)

['Python', 'Java', 'C++', 'Perl']

str(l)

Output: :

"['Python', 'Java', 'C++', 'Perl']"

repr(l)

Output: :

"['Python', 'Java', 'C++', 'Perl']"

d = {"a":3497, "b":8011, "c":8300}

print(d)

{'a': 3497, 'b': 8011, 'c': 8300}

str(d)

Output: :

"{'a': 3497, 'b': 8011, 'c': 8300}"

repr(d)

Output: :

"{'a': 3497, 'b': 8011, 'c': 8300}"

x = 587.78

str(x)

Output: :

'587.78'

repr(x)

Output: :

'587.78'

If you apply str or repr to an object, Python is looking for a corresponding method \_\_str\_\_ or \_\_repr\_\_ in the class definition of the object. If the method does exist, it will be called. In the following example, we define a class A, having neither a \_\_str\_\_ nor a \_\_repr\_\_ method. We want to see, what happens, if we use print directly on an instance of this class, or if we apply str or repr to this instance:

**class** **A**:

**pass**

a = A()

print(a)

<\_\_main\_\_.A object at 0x0000016B162A2688>

print(repr(a))

<\_\_main\_\_.A object at 0x0000016B162A2688>

print(str(a))

<\_\_main\_\_.A object at 0x0000016B162A2688>

a

Output: :

<\_\_main\_\_.A at 0x16b162a2688>

As both methods are not available, Python uses the default output for our object "a".

If a class has a \_\_str\_\_ method, the method will be used for an instance x of that class, if either the function str is applied to it or if it is used in a print function. \_\_str\_\_ will not be used, if repr is called, or if we try to output the value directly in an interactive Python shell:

**class** **A**:

**def** \_\_str\_\_(self):

**return** "42"

a = A()

print(repr(a))

<\_\_main\_\_.A object at 0x0000016B162A4108>

print(str(a))

42

a

Output: :

<\_\_main\_\_.A at 0x16b162a4108>

Otherwise, if a class has only the \_\_repr\_\_ method and no \_\_str\_\_ method, \_\_repr\_\_ will be applied in the situations, where \_\_str\_\_ would be applied, if it were available:

**class** **A**:

**def** \_\_repr\_\_(self):

**return** "42"

a = A()

print(repr(a))

print(str(a))

a

42

42

Output: :

42

A frequently asked question is when to use \_\_repr\_\_ and when \_\_str\_\_. \_\_str\_\_ is always the right choice, if the output should be for the end user or in other words, if it should be nicely printed. \_\_repr\_\_ on the other hand is used for the internal representation of an object. The output of \_\_repr\_\_ should be - if feasible - a string which can be parsed by the python interpreter. The result of this parsing is in an equal object. That is, the following should be true for an object "o":

o == eval(repr(o))

This is shown in the following interactive Python session:

l = [3,8,9]

s = repr(l)

s

Output: :

'[3, 8, 9]'

l == eval(s)

Output: :

True

l == eval(str(l))

Output: :

True

We show in the following example with the datetime module that eval can only be applied on the strings created by repr:

**import** **datetime**

today = datetime.datetime.now()

str\_s = str(today)

eval(str\_s)

Traceback **(most recent call last)**:

File "C:\Users\melis\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py", line 3326, in run\_code

exec(code\_obj, self.user\_global\_ns, self.user\_ns)

**File "<ipython-input-57-52b036e82a64>", line 4, in <module>**

**eval(str\_s)**

**File "<string>", line 1**

**2020-06-11 17:48:26.391635**

**^**

**SyntaxError:** invalid token

repr\_s = repr(today)

t = eval(repr\_s)

type(t)

Output: :

datetime.datetime

We can see that eval(repr\_s) returns again a datetime.datetime object. The string created by str can't be turned into a datetime.datetime object by parsing it.

We will extend our robot class with a repr method. We dropped the other methods to keep this example simple:

**class** **Robot**:

**def** \_\_init\_\_(self, name, build\_year):

self.name = name

self.build\_year = build\_year

**def** \_\_repr\_\_(self):

**return** "Robot('" + self.name + "', " + str(self.build\_year) + ")"

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979)

x\_str = str(x)

print(x\_str)

print("Type of x\_str: ", type(x\_str))

new = eval(x\_str)

print(new)

print("Type of new:", type(new))

Robot('Marvin', 1979)

Type of x\_str: <class 'str'>

Robot('Marvin', 1979)

Type of new: <class '\_\_main\_\_.Robot'>

x\_str has the value Robot('Marvin', 1979). eval(x\_str) converts it again into a Robot instance.

Now it's time to extend our class with a user friendly \_\_str\_\_ method:

**class** **Robot**:

**def** \_\_init\_\_(self, name, build\_year):

self.name = name

self.build\_year = build\_year

**def** \_\_repr\_\_(self):

**return** "Robot('" + self.name + "', " + str(self.build\_year) + ")"

**def** \_\_str\_\_(self):

**return** "Name: " + self.name + ", Build Year: " + str(self.build\_year)

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979)

x\_str = str(x)

print(x\_str)

print("Type of x\_str: ", type(x\_str))

new = eval(x\_str)

print(new)

print("Type of new:", type(new))

Name: Marvin, Build Year: 1979

Type of x\_str: <class 'str'>

Traceback **(most recent call last)**:

File "C:\Users\melis\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py", line 3326, in run\_code

exec(code\_obj, self.user\_global\_ns, self.user\_ns)

**File "<ipython-input-60-c8082321922a>", line 19, in <module>**

**new = eval(x\_str)**

**File "<string>", line 1**

**Name: Marvin, Build Year: 1979**

**^**

**SyntaxError:** invalid syntax

When we start this program, we can see that it is not possible to convert our string x\_str, created via str(x), into a Robot object anymore.

We show in the following program that x\_repr can still be turned into a Robot object:

**class** **Robot**:

**def** \_\_init\_\_(self, name, build\_year):

self.name = name

self.build\_year = build\_year

**def** \_\_repr\_\_(self):

**return** "Robot(**\"**" + self.name + "**\"**," + str(self.build\_year) + ")"

**def** \_\_str\_\_(self):

**return** "Name: " + self.name + ", Build Year: " + str(self.build\_year)

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979)

x\_repr = repr(x)

print(x\_repr, type(x\_repr))

new = eval(x\_repr)

print(new)

print("Type of new:", type(new))

Robot("Marvin",1979) <class 'str'>

Name: Marvin, Build Year: 1979

Type of new: <class '\_\_main\_\_.Robot'>

**Public, - Protected-, and Private Attributes**



Who doesn't know those trigger-happy farmers from films. Shooting as soon as somebody enters their property. This "somebody" has of course neglected the "no trespassing" sign, indicating that the land is private property. Maybe he hasn't seen the sign, maybe the sign is hard to be seen? Imagine a jogger, running the same course five times a week for more than a year, but than he receives a $50 fine for trespassing in the Winchester Fells. Trespassing is a criminal offence in Massachusetts. He was innocent anyway, because the signage was inadequate in the area*\*\**.

Even though no trespassing signs and strict laws do protect the private property, some surround their property with fences to keep off unwanted "visitors". Should the fence keep the dog in the yard or the burglar on the street? Choose your fence: Wood panel fencing, post-and-rail fencing, chain-link fencing with or without barbed wire and so on.

We have a similar situation in the design of object-oriented programming languages. The first decision to take is how to protect the data which should be private. The second decision is what to do if trespassing, i.e. accessing or changing private data, occurs. Of course, the private data may be protected in a way that it can't be accessed under any circumstances. This is hardly possible in practice, as we know from the old saying "Where there's a will, there's a way"!



Some owners allow a restricted access to their property. Joggers or hikers may find signs like "Enter at your own risk". A third kind of property might be public property like streets or parks, where it is perfectly legal to be.

We have the same classification again in object-oriented programming:

* Private attributes should only be used by the owner, i.e. inside of the class definition itself.
* Protected (restricted) Attributes may be used, but at your own risk. Essentially, they should only be used under certain conditions.
* Public Attributes can and should be freely used.

Python uses a special naming scheme for attributes to control the accessibility of the attributes. So far, we have used attribute names, which can be freely used inside or outside of a class definition, as we have seen. This corresponds to public attributes of course. There are two ways to restrict the access to class attributes:

* First, we can prefix an attribute name with a leading underscore "\_". This marks the attribute as protected. It tells users of the class not to use this attribute unless, they write a subclass. We will learn about inheritance and subclassing in the next chapter of our tutorial.
* Second, we can prefix an attribute name with two leading underscores "\_\_". The attribute is now inaccessible and invisible from outside. It's neither possible to read nor write to those attributes except inside the class definition itself**\***.

To summarize the attribute types:

|  |  |  |
| --- | --- | --- |
| **Naming** | **Type** | **Meaning** |
| name | Public | These attributes can be freely used inside or outside a class definition. |
| \_name | Protected | Protected attributes should not be used outside the class definition, unless inside a subclass definition. |
| \_\_name | Private | This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside the class definition itself. |

We want to demonstrate the behaviour of these attribute types with an example class:

**class** **A**():

**def** \_\_init\_\_(self):

self.\_\_priv = "I am private"

self.\_prot = "I am protected"

self.pub = "I am public"

We store this class (attribute\_tests.py) and test its behaviour in the following interactive Python shell:

**from** **attribute\_tests** **import** A

x = A()

x.pub

Output: :

'I am public'

x.pub = x.pub + " and my value can be changed"

x.pub

Output: :

'I am public and my value can be changed'

x.\_prot

Output: :

'I am protected'

x.\_\_priv

**---------------------------------------------------------------------------**

**AttributeError** Traceback (most recent call last)

**<ipython-input-6-f75b36b98afa>** in <module>

**----> 1** x**.**\_\_priv

**AttributeError**: 'A' object has no attribute '\_\_priv'

The error message is very interesting. One might have expected a message like "\_\_priv is private". We get the message "AttributeError: 'A' object has no attribute \_\_priv instead, which looks like a "lie". There is such an attribute, but we are told that there isn't. This is perfect information hiding. Telling a user that an attribute name is private, means that we make some information visible, i.e. the existence or non-existence of a private variable.

Our next task is rewriting our Robot class. Though we have Getter and Setter methods for the name and the build\_year, we can access the attributes directly as well, because we have defined them as public attributes. Data Encapsulation means, that we should only be able to access private attributes via getters and setters.

We have to replace each occurrence of self.name and self.build\_year by self.\_\_name and self.\_\_build\_year.

The listing of our revised class:

**class** **Robot**:

**def** \_\_init\_\_(self, name=**None**, build\_year=2000):

self.\_\_name = name

self.\_\_build\_year = build\_year

**def** say\_hi(self):

**if** self.\_\_name:

print("Hi, I am " + self.\_\_name)

**else**:

print("Hi, I am a robot without a name")

**def** set\_name(self, name):

self.\_\_name = name

**def** get\_name(self):

**return** self.\_\_name

**def** set\_build\_year(self, by):

self.\_\_build\_year = by

**def** get\_build\_year(self):

**return** self.\_\_build\_year

**def** \_\_repr\_\_(self):

**return** "Robot('" + self.\_\_name + "', " + str(self.\_\_build\_year) + ")"

**def** \_\_str\_\_(self):

**return** "Name: " + self.\_\_name + ", Build Year: " + str(self.\_\_build\_year)

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979)

y = Robot("Caliban", 1943)

**for** rob **in** [x, y]:

rob.say\_hi()

**if** rob.get\_name() == "Caliban":

rob.set\_build\_year(1993)

print("I was built in the year " + str(rob.get\_build\_year()) + "!")

Hi, I am Marvin

I was built in the year 1979!

Hi, I am Caliban

I was built in the year 1993!

Every private attribute of our class has a getter and a setter. There are IDEs for object-oriented programming languages, who automatically provide getters and setters for every private attribute as soon as an attribute is created.

This may look like the following class:

class A():

def \_\_init\_\_(self, x, y):

self.\_\_x = x

self.\_\_y = y

def GetX(self):

return self.\_\_x

def GetY(self):

return self.\_\_y

def SetX(self, x):

self.\_\_x = x

def SetY(self, y):

self.\_\_y = y

There are at least two good reasons against such an approach. First of all not every private attribute needs to be accessed from outside. Second, we will create non-pythonic code this way, as you will learn soon.

**Destructor**

What we said about constructors holds true for destructors as well. There is no "real" destructor, but something similar, i.e. the method \_\_del\_\_. It is called when the instance is about to be destroyed and if there is no other reference to this instance. If a base class has a \_\_del\_\_() method, the derived class's \_\_del\_\_() method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

The following script is an example with \_\_init\_\_ and \_\_del\_\_:

**class** **Robot**():

**def** \_\_init\_\_(self, name):

print(name + " has been created!")

**def** \_\_del\_\_(self):

print ("Robot has been destroyed")

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Tik-Tok")

y = Robot("Jenkins")

z = x

print("Deleting x")

**del** x

print("Deleting z")

**del** z

**del** y

Tik-Tok has been created!

Jenkins has been created!

Deleting x

Deleting z

Robot has been destroyed

Robot has been destroyed

The usage of the \_\_del\_\_ method is very problematic. If we change the previous code to personalize the deletion of a robot, we create an error:

**class** **Robot**():

**def** \_\_init\_\_(self, name):

print(name + " has been created!")

**def** \_\_del\_\_(self):

print (self.name + " says bye-bye!")

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Tik-Tok")

y = Robot("Jenkins")

z = x

print("Deleting x")

**del** x

print("Deleting z")

**del** z

**del** y

Tik-Tok has been created!

Jenkins has been created!

Deleting x

Deleting z

Exception ignored in: <function Robot.\_\_del\_\_ at 0x0000024A13CDC8B8>

Traceback (most recent call last):

File "<ipython-input-18-7dedd1b7f17b>", line 7, in \_\_del\_\_

AttributeError: 'Robot' object has no attribute 'name'

Exception ignored in: <function Robot.\_\_del\_\_ at 0x0000024A13CDC8B8>

Traceback (most recent call last):

File "<ipython-input-18-7dedd1b7f17b>", line 7, in \_\_del\_\_

AttributeError: 'Robot' object has no attribute 'name'

We are accessing an attribute which doesn't exist anymore. We will learn later, why this is the case.

Footnotes: + The picture on the right side is taken in the Library of the Court of Appeal for Ontario, located downtown Toronto in historic Osgoode Hall

++ "Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.) Every object has an identity, a type and a value." (excerpt from the official Python Language Reference)

+++ "attribute" stems from the Latin verb "attribuere" which means "to associate with"

++++ <http://www.wickedlocal.com/x937072506/tJogger-ticketed-for-trespassing-in-the-Winchester-Fells-kicks-back>

+++++ There is a way to access a private attribute directly. In our example, we can do it like this: x.\_Robot\_\_build\_year You shouldn't do this under any circumstances!

Next Chapter: [Class and Instance Attributes](https://www.python-course.eu/python3_class_and_instance_attributes.php)

**Class and Instance Attributes**

**Class Attributes**



Instance attributes are owned by the specific instances of a class. That is, for two different instances, the instance attributes are usually different. You should by now be familiar with this concept which we introduced in our previous chapter.

We can also define attributes at the class level. Class attributes are attributes which are owned by the class itself. They will be shared by all the instances of the class. Therefore they have the same value for every instance. We define class attributes outside all the methods, usually they are placed at the top, right below the class header.

In the following interactive Python session, we can see that the class attribute "a" is the same for all instances, in our examples "x" and "y". Besides this, we see that we can access a class attribute via an instance or via the class name:

**class** **A**:

a = "I am a class attribute!"

x = A()

y = A()

x.a

Output: :

'I am a class attribute!'

y.a

Output: :

'I am a class attribute!'

A.a

Output: :

'I am a class attribute!'

But be careful, if you want to change a class attribute, you have to do it with the notation ClassName.AttributeName. Otherwise, you will create a new instance variable. We demonstrate this in the following example:

**class** **A**:

a = "I am a class attribute!"

x = A()

y = A()

x.a = "This creates a new instance attribute for x!"

y.a

Output: :

'I am a class attribute!'

A.a

Output: :

'I am a class attribute!'

A.a = "This is changing the class attribute 'a'!"

A.a

Output: :

"This is changing the class attribute 'a'!"

y.a

Output: :

"This is changing the class attribute 'a'!"

x.a

*# but x.a is still the previously created instance variable*

Output: :

'This creates a new instance attribute for x!'

Python's class attributes and object attributes are stored in separate dictionaries, as we can see here:

x.\_\_dict\_\_

Output: :

{'a': 'This creates a new instance attribute for x!'}

y.\_\_dict\_\_

Output: :

{}

A.\_\_dict\_\_

Output: :

mappingproxy({'\_\_module\_\_': '\_\_main\_\_',

'a': "This is changing the class attribute 'a'!",

'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'A' objects>,

'\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'A' objects>,

'\_\_doc\_\_': None})

x.\_\_class\_\_.\_\_dict\_\_

Output: :

mappingproxy({'\_\_module\_\_': '\_\_main\_\_',

'a': "This is changing the class attribute 'a'!",

'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'A' objects>,

'\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'A' objects>,

'\_\_doc\_\_': None})

**Example with Class Attributes**

Isaac Asimov devised and introduced the so-called "Three Laws of Robotics" in 1942. The appeared in his story "Runaround". His three laws have been picked up by many science fiction writers. As we have started manufacturing robots in Python, it's high time to make sure that they obey Asimovs three laws. As they are the same for every instance, i.e. robot, we will create a class attribute Three\_Laws. This attribute is a tuple with the three laws.

**class** **Robot**:

Three\_Laws = (

*"""A robot may not injure a human being or, through inaction, allow a human being to come to harm."""*,

*"""A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.,"""*,

*"""A robot must protect its own existence as long as such protection does not conflict with the First or Second Law."""*

)

**def** \_\_init\_\_(self, name, build\_year):

self.name = name

self.build\_year = build\_year

*# other methods as usual*

As we mentioned before, we can access a class attribute via instance or via the class name. You can see in the following that we don't need an instance:

**from** **robot\_asimov** **import** Robot

**for** number, text **in** enumerate(Robot.Three\_Laws):

print(str(number+1) + ":**\n**" + text)

1:

A robot may not injure a human being or, through inaction, allow a human being to come to harm.

2:

A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.,

3:

A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

In the following example, we demonstrate, how you can count instance with class attributes. All we have to do is

* to create a class attribute, which we call "counter" in our example
* to increment this attribute by 1 every time a new instance is created
* to decrement the attribute by 1 every time an instance is destroyed

**class** **C**:

counter = 0

**def** \_\_init\_\_(self):

type(self).counter += 1

**def** \_\_del\_\_(self):

type(self).counter -= 1

**if** \_\_name\_\_ == "\_\_main\_\_":

x = C()

print("Number of instances: : " + str(C.counter))

y = C()

print("Number of instances: : " + str(C.counter))

**del** x

print("Number of instances: : " + str(C.counter))

**del** y

print("Number of instances: : " + str(C.counter))

Number of instances: : 1

Number of instances: : 2

Number of instances: : 1

Number of instances: : 0

Principially, we could have written C.counter instead of type(self).counter, because type(self) will be evaluated to "C" anyway. However we will understand later, that type(self) makes sense, if we use such a class as a superclass.

**Static Methods**

We used class attributes as public attributes in the previous section. Of course, we can make public attributes private as well. We can do this by adding the double underscore again. If we do so, we need a possibility to access and change these private class attributes. We could use instance methods for this purpose:

**class** **Robot**:

\_\_counter = 0

**def** \_\_init\_\_(self):

type(self).\_\_counter += 1

**def** RobotInstances(self):

**return** Robot.\_\_counter

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot()

print(x.RobotInstances())

y = Robot()

print(x.RobotInstances())

1

2

This is not a good idea for two reasons: First of all, because the number of robots has nothing to do with a single robot instance and secondly because we can't inquire the number of robots before we create an instance. If we try to invoke the method with the class name Robot.RobotInstances(), we get an error message, because it needs an instance as an argument:

Robot.RobotInstances()

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-35-f53600e3296e>** in <module>

**----> 1** Robot**.**RobotInstances**()**

**TypeError**: RobotInstances() missing 1 required positional argument: 'self'

The next idea, which still doesn't solve our problem, is omitting the parameter "self":

**class** **Robot**:

\_\_counter = 0

**def** \_\_init\_\_(self):

type(self).\_\_counter += 1

**def** RobotInstances():

**return** Robot.\_\_counter

Now it's possible to access the method via the class name, but we can't call it via an instance:

**from** **static\_methods2** **import** Robot

Robot.RobotInstances()

Output: :

0

x = Robot()

x.RobotInstances()

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-40-4d5e11c3474a>** in <module>

1 x **=** Robot**()**

**----> 2** x**.**RobotInstances**()**

**TypeError**: RobotInstances() takes 0 positional arguments but 1 was given

The call "x.RobotInstances()" is treated as an instance method call and an instance method needs a reference to the instance as the first parameter.

So, what do we want? We want a method, which we can call via the class name or via the instance name without the necessity of passing a reference to an instance to it.

The solution lies in static methods, which don't need a reference to an instance. It's easy to turn a method into a static method. All we have to do is to add a line with "@staticmethod" directly in front of the method header. It's the decorator syntax.

You can see in the following example that we can now use our method RobotInstances the way we want:

**class** **Robot**:

\_\_counter = 0

**def** \_\_init\_\_(self):

type(self).\_\_counter += 1

@staticmethod

**def** RobotInstances():

**return** Robot.\_\_counter

**if** \_\_name\_\_ == "\_\_main\_\_":

print(Robot.RobotInstances())

x = Robot()

print(x.RobotInstances())

y = Robot()

print(x.RobotInstances())

print(Robot.RobotInstances())

0

1

2

2

**Class Methods**

Static methods shouldn't be confused with class methods. Like static methods class methods are not bound to instances, but unlike static methods class methods are bound to a class. The first parameter of a class method is a reference to a class, i.e. a class object. They can be called via an instance or the class name.

**class** **Robot**:

\_\_counter = 0

**def** \_\_init\_\_(self):

type(self).\_\_counter += 1

@classmethod

**def** RobotInstances(cls):

**return** cls, Robot.\_\_counter

**if** \_\_name\_\_ == "\_\_main\_\_":

print(Robot.RobotInstances())

x = Robot()

print(x.RobotInstances())

y = Robot()

print(x.RobotInstances())

print(Robot.RobotInstances())

(<class '\_\_main\_\_.Robot'>, 0)

(<class '\_\_main\_\_.Robot'>, 1)

(<class '\_\_main\_\_.Robot'>, 2)

(<class '\_\_main\_\_.Robot'>, 2)

The use cases of class methods:

* They are used in the definition of the so-called factory methods, which we will not cover here.
* They are often used, where we have static methods, which have to call other static methods. To do this, we would have to hard code the class name, if we had to use static methods. This is a problem, if we are in a use case, where we have inherited classes.

The following program contains a fraction class, which is still not complete. If you work with fractions, you need to be capable of reducing fractions, e.g. the fraction 8/24 can be reduced to 1/3. We can reduce a fraction to lowest terms by dividing both the numerator and denominator by the Greatest Common Divisor (GCD).

We have defined a static gcd function to calculate the greatest common divisor of two numbers. the greatest common divisor (gcd) of two or more integers (at least one of which is not zero), is the largest positive integer that divides the numbers without a remainder. For example, the 'GCD of 8 and 24 is 8. The static method "gcd" is called by our class method "reduce" with "cls.gcd(n1, n2)". "CLS" is a reference to "fraction".

**class** **fraction**(object):

**def** \_\_init\_\_(self, n, d):

self.numerator, self.denominator = fraction.reduce(n, d)

@staticmethod

**def** gcd(a,b):

**while** b != 0:

a, b = b, a%**b**

**return** a

@classmethod

**def** reduce(cls, n1, n2):

g = cls.gcd(n1, n2)

**return** (n1 // g, n2 // g)

**def** \_\_str\_\_(self):

**return** str(self.numerator)+'/'+str(self.denominator)

Using this class:

**from** **fraction1** **import** fraction

x = fraction(8,24)

print(x)

1/3

**Class Methods vs. Static Methods and Instance Methods**

Our last example will demonstrate the usefulness of class methods in inheritance. We define a class "Pet" with a method "about". This method should give some general class information. The class Cats will be inherited in a subclass "Dog" and "Cat". The method "about" will be inherited as well. We will demonstrate that we will encounter problems, if we define the method "about" as a normal instance method or as a static method. We will start by defining "about" as an instance method:

**class** **Pet**:

\_class\_info = "pet animals"

**def** about(self):

print("This class is about " + self.\_class\_info + "!")

**class** **Dog**(Pet):

\_class\_info = "man's best friends"

**class** **Cat**(Pet):

\_class\_info = "all kinds of cats"

p = Pet()

p.about()

d = Dog()

d.about()

c = Cat()

c.about()

This class is about pet animals!

This class is about man's best friends!

This class is about all kinds of cats!

This may look alright at first at first glance. On second thought we recognize the awful design. We had to create instances of the Pet, Dog and Cat classes to be able to ask what the class is about. It would be a lot better, if we could just write "Pet.about()", "Dog.about()" and "Cat.about()" to get the previous result. We cannot do this. We will have to write "Pet.about(p)", "Dog.about(d)" and "Cat.about(c)" instead.

Now, we will define the method "about" as a "staticmethod" to show the disadvantage of this approach. As we have learned previously in our tutorial, a staticmethod does not have a first parameter with a reference to an object. So about will have no parameters at all. Due to this, we are now capable of calling "about" without the necessity of passing an instance as a parameter, i.e. "Pet.about()", "Dog.about()" and "Cat.about()". Yet, a problem lurks in the definition of "about". The only way to access the class info \_class\_info is putting a class name in front. We arbitrarily put in "Pet". We could have put there "Cat" or "Dog" as well. No matter what we do, the solution will not be what we want:

**class** **Pet**:

\_class\_info = "pet animals"

@staticmethod

**def** about():

print("This class is about " + Pet.\_class\_info + "!")

**class** **Dog**(Pet):

\_class\_info = "man's best friends"

**class** **Cat**(Pet):

\_class\_info = "all kinds of cats"

Pet.about()

Dog.about()

Cat.about()

This class is about pet animals!

This class is about pet animals!

This class is about pet animals!

In other words, we have no way of differenciating between the class Pet and its subclasses Dog and Cat. The problem is that the method "about" does not know that it has been called via the Pet, the Dog or the Cat class.

A classmethod is the solution to all our problems. We will decorate "about" with a classmethod decorator instead of a staticmethod decorator:

**class** **Pet**:

\_class\_info = "pet animals"

@classmethod

**def** about(cls):

print("This class is about " + cls.\_class\_info + "!")

**class** **Dog**(Pet):

\_class\_info = "man's best friends"

**class** **Cat**(Pet):

\_class\_info = "all kinds of cats"

Pet.about()

Dog.about()

Cat.about()

This class is about pet animals!

This class is about man's best friends!

This class is about all kinds of cats!

**Properties vs. Getters and Setters**

**Properties**



Getters(also known as 'accessors') and setters (aka. 'mutators') are used in many object oriented programming languages to ensure the principle of data encapsulation. Data encapsulation - as we have learnt in our introduction on [Object Oriented Programming](https://www.python-course.eu/python3_object_oriented_programming.php) of our tutorial - is seen as the bundling of data with the methods that operate on them. These methods are of course the *getter* for retrieving the data and the *setter* for changing the data. According to this principle, the attributes of a class are made private to hide and protect them from the other codes.

Unfortunately, it is widespread belief that a proper Python class should encapsulate private attributes by using getters and setters. As soon as one of these programmers introduces a new attribute, he or she will make it a private variable and creates "automatically" a getter and a setter for this attributes. Such programmers may even use an editor or an IDE, which automatically creates getters and setters for all private attributes. These tools even warn the programmer if she or he uses a public attribute! Java programmers will wrinkle their brows, screw up their noses, or even scream with horror when they read the following: The Pythonic way to introduce attributes is to make them public.

We will explain this later. First, we demonstrate in the following example, how we can design a class in a Javaesque way with getters and setters to encapsulate the private attribute self.\_\_x:

**class** **P**:

**def** \_\_init\_\_(self,x):

self.\_\_x = x

**def** get\_x(self):

**return** self.\_\_x

**def** set\_x(self, x):

self.\_\_x = x

We can see in the following demo session how to work with this class and the methods:

**from** **mutators** **import** P

p1 = P(42)

p2 = P(4711)

p1.get\_x()

Output: :

42

p1.set\_x(47)

p1.set\_x(p1.get\_x()+p2.get\_x())

p1.get\_x()

Output: :

4758

What do you think about the expression "p1.set\_x(p1.get\_x()+p2.get\_x())"? It's ugly, isn't it? It's a lot easier to write an expression like the following, if we had a public attribute x:

p1.x = p1.x + p2.x

Such an assignment is easier to write and above all easier to read than the Javaesque expression.

Let's rewrite the class P in a Pythonic way. No getter, no setter and instead of the private attribute self.\_\_x we use a public one:

**class** **P**:

**def** \_\_init\_\_(self,x):

self.x = x

Beautiful, isn't it? Just three lines of code, if we don't count the blank line!

**from** **p** **import** P

p1 = P(42)

p2 = P(4711)

p1.x

Output: :

42

p1.x = 47

p1.x = p1.x + p2.x

p1.x

Output: :

4758

"But, but, but, but, but ... ", we can hear them howling and screaming, "But there is NO data ENCAPSULATION!" Yes, in this case there is no data encapsulation. We don't need it in this case. The only thing get\_x and set\_x in our starting example did was "getting the data through" without doing anything.

But what happens if we want to change the implementation in the future? This is a serious argument. Let's assume we want to change the implementation like this: The attribute x can have values between 0 and 1000. If a value larger than 1000 is assigned, x should be set to 1000. Correspondingly, x should be set to 0, if the value is less than 0.

It is easy to change our first P class to cover this problem. We change the set\_x method accordingly:

**class** **P**:

**def** \_\_init\_\_(self,x):

self.set\_x(x)

**def** get\_x(self):

**return** self.\_\_x

**def** set\_x(self, x):

**if** x < 0:

self.\_\_x = 0

**elif** x > 1000:

self.\_\_x = 1000

**else**:

self.\_\_x = x

The following Python session shows that it works the way we want it to work:

**from** **mutators1** **import** P

p1 = P(1001)

p1.get\_x()

Output: :

1000

p2 = P(15)

p2.get\_x()

Output: :

15

p3 = P(-1)

p3.get\_x()

Output: :

0

But there is a catch: Let's assume we designed our class with the public attribute and no methods. People have already used it a lot and they have written code like this:

**from** **p** **import** P

p1 = P(42)

p1.x = 1001

p1.x

Output: :

1001

Our new class means breaking the interface. The attribute x is not available anymore. That's why in Java e.g. people are recommended to use only private attributes with getters and setters, so that they can change the implementation without having to change the interface.

But Python offers a solution to this problem. The solution is called *properties*!

The class with a property looks like this:

**class** **P**:

**def** \_\_init\_\_(self,x):

self.x = x

@property

**def** x(self):

**return** self.\_\_x

@x.setter

**def** x(self, x):

**if** x < 0:

self.\_\_x = 0

**elif** x > 1000:

self.\_\_x = 1000

**else**:

self.\_\_x = x

A method which is used for getting a value is decorated with "@property", i.e. we put this line directly in front of the header. The method which has to function as the setter is decorated with "@x.setter". If the function had been called "f", we would have to decorate it with "@f.setter". Two things are noteworthy: We just put the code line "self.x = x" in the \_\_init\_\_ method and the property method x is used to check the limits of the values. The second interesting thing is that we wrote "two" methods with the same name and a different number of parameters "def x(self)" and "def x(self,x)". We have learned in a previous chapter of our course that this is not possible. It works here due to the decorating:

**from** **p2** **import** P

p1 = P(1001)

p1.x

Output: :

1000

p1.x = -12

p1.x

Output: :

0

Alternatively, we could have used a different syntax without decorators to define the property. As you can see, the code is definitely less elegant and we have to make sure that we use the getter function in the \_\_init\_\_ method again:

**class** **P**:

**def** \_\_init\_\_(self,x):

self.set\_x(x)

**def** get\_x(self):

**return** self.\_\_x

**def** set\_x(self, x):

**if** x < 0:

self.\_\_x = 0

**elif** x > 1000:

self.\_\_x = 1000

**else**:

self.\_\_x = x

x = property(get\_x, set\_x)

There is still another problem in the most recent version. We have now two ways to access or change the value of x: Either by using "p1.x = 42" or by "p1.set\_x(42)". This way we are violating one of the fundamentals of Python: "There should be one-- and preferably only one --obvious way to do it." (see [Zen of Python](https://www.python-course.eu/python3_history_and_philosophy.php))

We can easily fix this problem by turning the getter and the setter methods into private methods, which can't be accessed anymore by the users of our class P:

**class** **P**:

**def** \_\_init\_\_(self,x):

self.\_\_set\_x(x)

**def** \_\_get\_x(self):

**return** self.\_\_x

**def** \_\_set\_x(self, x):

**if** x < 0:

self.\_\_x = 0

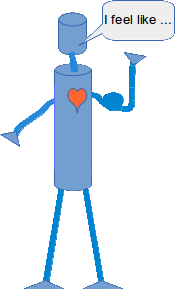
**elif** x > 1000:

self.\_\_x = 1000

**else**:

self.\_\_x = x

x = property(\_\_get\_x, \_\_set\_x)



Even though we fixed this problem by using a private getter and setter, the version with the decorator "@property" is the Pythonic way to do it!

From what we have written so far, and what can be seen in other books and tutorials as well, we could easily get the impression that there is a one-to-one connection between properties (or mutator methods) and the attributes, i.e. that each attribute has or should have its own property (or getter-setter-pair) and the other way around. Even in other object oriented languages than Python, it's usually not a good idea to implement a class like that. The main reason is that many attributes are only internally needed and creating interfaces for the user of the class increases unnecessarily the usability of the class. The possible user of a class shouldn't be "drowned" with umpteen - of mainly unnecessary - methods or properties!

The following example shows a class, which has internal attributes, which can't be accessed from outside. These are the private attributes self.\_\_potential \_physical and self.\_\_potential\_psychic. Furthermore we show that a property can be deduced from the values of more than one attribute. The property "condition" of our example returns the condition of the robot in a descriptive string. The condition depends on the sum of the values of the psychic and the physical conditions of the robot.

**class** **Robot**:

**def** \_\_init\_\_(self, name, build\_year, lk = 0.5, lp = 0.5 ):

self.name = name

self.build\_year = build\_year

self.\_\_potential\_physical = lk

self.\_\_potential\_psychic = lp

@property

**def** condition(self):

s = self.\_\_potential\_physical + self.\_\_potential\_psychic

**if** s <= -1:

**return** "I feel miserable!"

**elif** s <= 0:

**return** "I feel bad!"

**elif** s <= 0.5:

**return** "Could be worse!"

**elif** s <= 1:

**return** "Seems to be okay!"

**else**:

**return** "Great!"

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Robot("Marvin", 1979, 0.2, 0.4 )

y = Robot("Caliban", 1993, -0.4, 0.3)

print(x.condition)

print(y.condition)

Seems to be okay!

I feel bad!

**Public instead of Private Attributes**

Let's summarize the usage of private and public attributes, getters and setters, and properties: Let's assume that we are designing a new class and we pondering about an instance or class attribute "OurAtt", which we need for the design of our class. We have to observe the following issues:

* Will the value of "OurAtt" be needed by the possible users of our class?
* If not, we can or should make it a private attribute.
* If it has to be accessed, we make it accessible as a public attribute
* We will define it as a private attribute with the corresponding property, if and only if we have to do some checks or transformation of the data. (As an example, you can have a look again at our class P, where the attribute has to be in the interval between 0 and 1000, which is ensured by the property "x")
* Alternatively, you could use a getter and a setter, but using a property is the Pythonic way to deal with it!

Let's assume we defined "OurAtt" as a public attribute. Our class has been successfully used by other users for quite a while. Now comes the point which frightens some traditional OOPistas out of their wits: Imagine "OurAtt" has been used as an integer. Now, our class has to ensure that "OurAtt" has to be a value between 0 and 1000? Without property, this is really a horrible scenario! Due to properties it's easy: We create a property version of "OurAtt".

**class** **OurClass**:

**def** \_\_init\_\_(self, a):

self.OurAtt = a

x = OurClass(10)

print(x.OurAtt)

10



**class** **OurClass**:

**def** \_\_init\_\_(self, a):

self.OurAtt = a

@property

**def** OurAtt(self):

**return** self.\_\_OurAtt

@OurAtt.setter

**def** OurAtt(self, val):

**if** val < 0:

self.\_\_OurAtt = 0

**elif** val > 1000:

self.\_\_OurAtt = 1000

**else**:

self.\_\_OurAtt = val

x = OurClass(10)

print(x.OurAtt)

10

This is great, isn't it? You can start with the simplest implementation imaginable, and you are free to later migrate to a property version without having to change the interface! So properties are not just a replacement for getters and setters!

Something else you might have already noticed: For the users of a class, properties are syntactically identical to ordinary attributes.

## Deeper into Properties



In the previous chapter of our tutorial, we learned how to create and use properties in a class. The main objective was to understand them as a way to get rid of explicit getters and setters and have a simple class interface. This is usually enough to know for most programmers and for practical use cases and they will not need more.

If you want to know more about how 'property' works, you can go one step further with us. By doing this, you can improve your coding skills and get a deeper insight and understanding of Python. We will have a look at the way the "property" decorator could be implemented in Python code. (It is implemented in C code in reality!) By doing this, the way of working will be clearer. Everything is based on the descriptor protocol, which we will explain later.

We define a class with the name 'our\_property' so that it will not be mistaken for the existing 'property' class. This class can be used like the 'real' property class.

**class** **our\_property**:

*""" emulation of the property class*

*for educational purposes """*

**def** \_\_init\_\_(self,

fget=**None**,

fset=**None**,

fdel=**None**,

doc=**None**):

*"""Attributes of 'our\_decorator'*

*fget*

*function to be used for getting*

*an attribute value*

*fset*

*function to be used for setting*

*an attribute value*

*fdel*

*function to be used for deleting*

*an attribute*

*doc*

*the docstring*

*"""*

self.fget = fget

self.fset = fset

self.fdel = fdel

**if** doc **is** **None** **and** fget **is** **not** **None**:

doc = fget.\_\_doc\_\_

self.\_\_doc\_\_ = doc

**def** \_\_get\_\_(self, obj, objtype=**None**):

**if** obj **is** **None**:

**return** self

**if** self.fget **is** **None**:

**raise** **AttributeError**("unreadable attribute")

**return** self.fget(obj)

**def** \_\_set\_\_(self, obj, value):

**if** self.fset **is** **None**:

**raise** **AttributeError**("can't set attribute")

self.fset(obj, value)

**def** \_\_delete\_\_(self, obj):

**if** self.fdel **is** **None**:

**raise** **AttributeError**("can't delete attribute")

self.fdel(obj)

**def** getter(self, fget):

**return** type(self)(fget, self.fset, self.fdel, self.\_\_doc\_\_)

**def** setter(self, fset):

**return** type(self)(self.fget, fset, self.fdel, self.\_\_doc\_\_)

**def** deleter(self, fdel):

**return** type(self)(self.fget, self.fset, fdel, self.\_\_doc\_\_)

We need another class to use the previously defined class and to demonstrate how the property class decorator works. To continue the tradition of the previous chapters of our Python tutorial we will again write a Robot class. We will define a property in this example class to demonstrate the usage of our previously defined property class or better 'our\_decorator' class. When you run the code, you can see \_\_init\_\_ of 'our\_property' will be called 'fget' set to a reference to the 'getter' function of 'city'. The attribute 'city' is an instance of the 'our\_property' class. The 'our'property' class provides another decorator 'setter' for the setter functionality. We apply this with '@city.setter'

**class** **Robot**:

**def** \_\_init\_\_(self, city):

self.city = city

@our\_property

**def** city(self):

print("The Property 'city' will be returned now:")

**return** self.\_\_city

@city.setter

**def** city(self, city):

print("'city' will be set")

self.\_\_city = city

'Robot.city' is an instance of the 'our\_property' class as we can see in the following:

type(Robot.city)

Output: :

\_\_main\_\_.our\_property

If you change the line '@our\_property' to '@property' the program will behave totally the same, but it will be using the original Python class 'property' and not our 'our\_property' class. We will create instances of the Robot class in the following Python code:

print("Instantiating a Root and setting 'city' to 'Berlin'")

robo = Robot("Berlin")

print("The value is: ", robo.city)

print("Our robot moves now to Frankfurt:")

robo.city = "Frankfurt"

print("The value is: ", robo.city)

Instantiating a Root and setting 'city' to 'Berlin'

'city' will be set

The Property 'city' will be returned now:

The value is: Berlin

Our robot moves now to Frankfurt:

'city' will be set

The Property 'city' will be returned now:

The value is: Frankfurt

Let's make our property implementation a little bit more talkative with some print functions to see what is going on. We also change the name to 'chatty\_property' for obvious reasons:

**class** **chatty\_property**:

*""" emulation of the property class*

*for educational purposes """*

**def** \_\_init\_\_(self,

fget=**None**,

fset=**None**,

fdel=**None**,

doc=**None**):

self.fget = fget

self.fset = fset

self.fdel = fdel

print("**\n**\_\_init\_\_ called with:)")

print(f"fget=**{fget}**, fset=**{fset}**, fdel=**{fdel}**, doc=**{doc}**")

**if** doc **is** **None** **and** fget **is** **not** **None**:

print(f"doc set to docstring of **{fget.\_\_name\_\_}** method")

doc = fget.\_\_doc\_\_

self.\_\_doc\_\_ = doc

**def** \_\_get\_\_(self, obj, objtype=**None**):

**if** obj **is** **None**:

**return** self

**if** self.fget **is** **None**:

**raise** **AttributeError**("unreadable attribute")

**return** self.fget(obj)

**def** \_\_set\_\_(self, obj, value):

**if** self.fset **is** **None**:

**raise** **AttributeError**("can't set attribute")

self.fset(obj, value)

**def** \_\_delete\_\_(self, obj):

**if** self.fdel **is** **None**:

**raise** **AttributeError**("can't delete attribute")

self.fdel(obj)

**def** getter(self, fget):

**return** type(self)(fget, self.fset, self.fdel, self.\_\_doc\_\_)

**def** setter(self, fset):

print(type(self))

**return** type(self)(self.fget, fset, self.fdel, self.\_\_doc\_\_)

**def** deleter(self, fdel):

**return** type(self)(self.fget, self.fset, fdel, self.\_\_doc\_\_)

**class** **Robot**:

**def** \_\_init\_\_(self, city):

self.city = city

@chatty\_property

**def** city(self):

*""" city attribute of Robot """*

print("The Property 'city' will be returned now:")

**return** self.\_\_city

@city.setter

**def** city(self, city):

print("'city' will be set")

self.\_\_city = city

\_\_init\_\_ called with:)

fget=<function Robot.city at 0x000002825C9420D8>, fset=None, fdel=None, doc=None

doc set to docstring of city method

<class '\_\_main\_\_.chatty\_property'>

\_\_init\_\_ called with:)

fget=<function Robot.city at 0x000002825C9420D8>, fset=<function Robot.city at 0x000002825C9425E8>, fdel=None, doc= city attribute of Robot

robo = Robot("Berlin")

'city' will be set

Previous Chapter: [Properties vs. getters and setters](https://www.python-course.eu/python3_properties.php)  
Next Chapter: [Descriptors](https://www.python-course.eu/python3_descriptors.php)

**Descriptors**



In the previous two chapters of our Python tutorial, we learned how to use [Python properties](https://www.python-course.eu/python3_properties.php) and even how to implement a custom-made property class. In this chapter you will learn the details of descriptors.

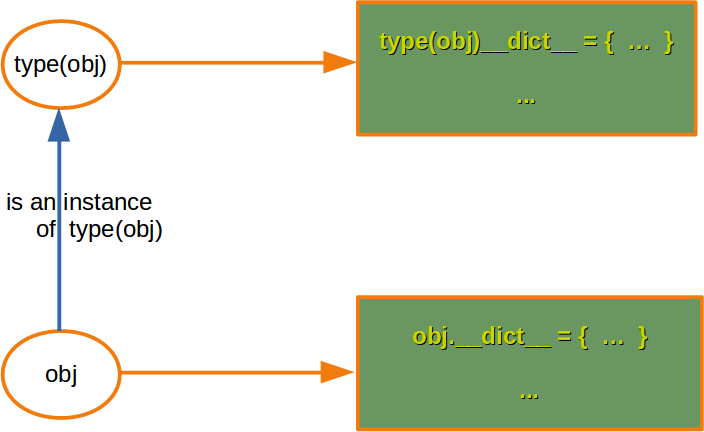
Descriptors were introduced to Python in version 2.2. That time the "What's New in Python2.2" mentioned: "The one big idea underlying the new class model is that an API for describing the attributes of an object using descriptors has been formalized. Descriptors specify the value of an attribute, stating whether it’s a method or a field. With the descriptor API, static methods and class methods become possible, as well as more exotic constructs."

A descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are \_\_get\_\_(), \_\_set\_\_(), and \_\_delete\_\_().

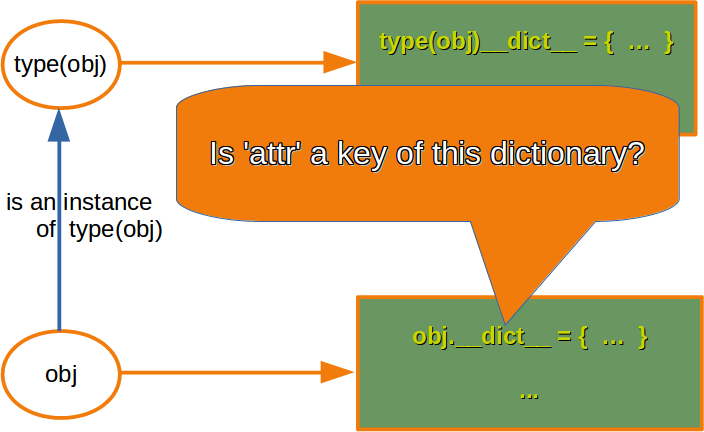
If any of those methods are defined for an object, it is said to be a descriptor.

Their purpose consists in providing programmers with the ability to add managed attributes to classes. The descriptors are introduced to get, set or delete attributes from the object's \_\_dict\_\_ dictionary via the above mentioned methods. Accessing a class attribute will start the lookup chain.

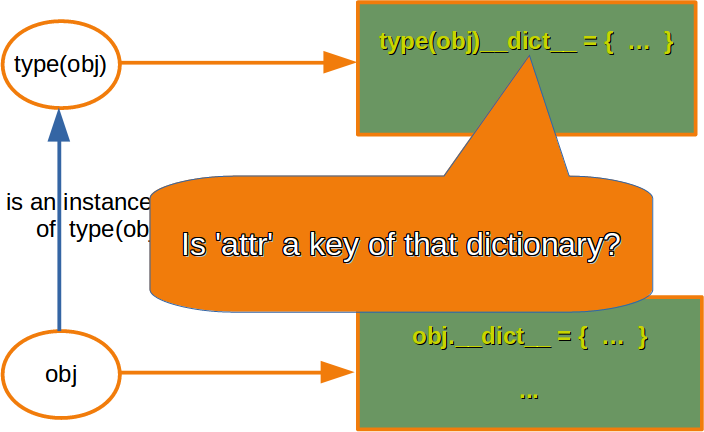
Let a closer look at what is happening. Assuming we have an object obj: What happens if we try to access an attribute (or property) ap? "Accesssing" the attribute means to "get" the value, so the attribute is used for example in a print function or inside of an expression. Both the obj and the class belong to type(obj) contain a dictionary attribute \_\_dict\_\_. This situation is viuslaized in the following diagram:



obj.ap has a lookup chain starting with obj.\_\_dict\_\_['ap'], i.e. checks if obj.ap is a key of the dictionary obj.\_\_dict\_\_['ap'].



If ap is not a key of obj.\_\_dict\_\_, it will try to lookup type(obj).\_\_dict\_\_['ap'].



If obj is not contained in this dictionary either, it will continue checking through the base classes of type(ap) excluding metaclasses.

We demonstrate this in an example:

**class** **A**:

ca\_A = "class attribute of A"

**def** \_\_init\_\_(self):

self.ia\_A = "instance attribute of A instance"

**class** **B**(A):

ca\_B = "class attribute of B"

**def** \_\_init\_\_(self):

super().\_\_init\_\_()

self.ia\_B = "instance attribute of A instance"

x = B()

print(x.ia\_B)

print(x.ca\_B)

print(x.ia\_A)

print(x.ca\_A)

instance attribute of A instance

class attribute of B

instance attribute of A instance

class attribute of A

If we call print(x.non\_existing) we get the following exception:

---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-4-119192d61d5e> in <module>

----> 1 print(x.non\_existing)

AttributeError: 'B' object has no attribute 'non\_existing'

If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined.

Descriptors provide a powerful, general purpose protocol, which is the underlying mechanism behind properties, methods, static methods, class methods, and super(). Descriptors were needed to implement the so-called new style classes introduced in version 2.2. The "new style classes" are the default nowadays.

**Descriptor Protocol**

The general descriptor protocol consists of three methods:

descr.\_\_get\_\_(self, obj, type=None) -> value

descr.\_\_set\_\_(self, obj, value) -> None

descr.\_\_delete\_\_(self, obj) -> None

If you define one or more of these methods, you will create a descriptor. We distinguish between data descriptors and non-data descriptors:

**non-data descriptor**

If we define only the \_\_get\_\_() method, we create a non-data descriptor, which are mostly used for methods.

**data descriptor**

If an object defines \_\_set\_\_() or \_\_delete\_\_(), it is considered a data descriptor. To make a read-only data descriptor, define both \_\_get\_\_() and \_\_set\_\_() with the \_\_set\_\_() raising an AttributeError when called. Defining the \_\_set\_\_() method with an exception raising placeholder is enough to make it a data descriptor.

We finally come now to our simple descriptor example in the following code:

**class** **SimpleDescriptor**(object):

*"""*

*A simple data descriptor that can set and return values*

*"""*

**def** \_\_init\_\_(self, initval=**None**):

print("\_\_init\_\_ of SimpleDecorator called with initval: ", initval)

self.\_\_set\_\_(self, initval)

**def** \_\_get\_\_(self, instance, owner):

print(instance, owner)

print('Getting (Retrieving) self.val: ', self.val)

**return** self.val

**def** \_\_set\_\_(self, instance, value):

print('Setting self.val to ', value)

self.val = value

**class** **MyClass**(object):

x = SimpleDescriptor("green")

m = MyClass()

print(m.x)

m.x = "yellow"

print(m.x)

\_\_init\_\_ of SimpleDecorator called with initval: green

Setting self.val to green

<\_\_main\_\_.MyClass object at 0x7efe93ff4a20> <class '\_\_main\_\_.MyClass'>

Getting (Retrieving) self.val: green

green

Setting self.val to yellow

<\_\_main\_\_.MyClass object at 0x7efe93ff4a20> <class '\_\_main\_\_.MyClass'>

Getting (Retrieving) self.val: yellow

yellow

The third parameter owner of \_\_get\_\_ is always the owner class and provides users with an option to do something with the class that was used to call the descriptor. Usually, i.e. if the descriptor is called through an object obj, the object type can be deduced by calling type(obj). The situation is different, if the descriptor is invoked through a class. In this case it is None and it wouldn't be possible to access the class unless the third argument is given. The second parameter instance is the instance that the attribute was accessed through, or None when the attribute is accessed through the owner.

Let's have a look at the \_\_dict\_\_ dictionaries of both the instances and the class:

'x' is a class attribute in the previous class. You may have asked yourself, if we could also use this mechanism in the \_\_init\_\_ method to define instance attribute. This is not possible. The methods \_\_get\_\_(), \_\_set\_\_(), and \_\_delete\_\_() will only apply if an instance of the class containing the method (a so-called descriptor class) appears in an owner class (the descriptor must be in either the owner’s class dictionary or in the class dictionary for one of its parents). In the examples above, the attribute 'x' is in the owner \_\_dict\_\_ of the owner class MyClass, as we can see below:

print(m.\_\_dict\_\_)

print(MyClass.\_\_dict\_\_)

print(SimpleDescriptor.\_\_dict\_\_)

{}

{'\_\_module\_\_': '\_\_main\_\_', 'x': <\_\_main\_\_.SimpleDescriptor object at 0x7efe93ff49e8>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'MyClass' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'MyClass' objects>, '\_\_doc\_\_': None}

{'\_\_module\_\_': '\_\_main\_\_', '\_\_doc\_\_': '\n A simple data descriptor that can set and return values\n ', '\_\_init\_\_': <function SimpleDescriptor.\_\_init\_\_ at 0x7efe93fc8d08>, '\_\_get\_\_': <function SimpleDescriptor.\_\_get\_\_ at 0x7efe93fc8840>, '\_\_set\_\_': <function SimpleDescriptor.\_\_set\_\_ at 0x7efe93fc88c8>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'SimpleDescriptor' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'SimpleDescriptor' objects>}

print(MyClass.\_\_dict\_\_['x'])

<\_\_main\_\_.SimpleDescriptor object at 0x7efe93ff49e8>

It is possible to call a descriptor directly by its method name, e.g. d.\_\_get\_\_(obj).

Alternatively, it is more common for a descriptor to be invoked automatically upon attribute access. For example, obj.d looks up d in the dictionary \_\_dict\_\_ of obj. If d defines the method \_\_get\_\_(), then d.\_\_get\_\_(obj) is invoked according to the precedence rules listed below.

It makes a difference if obj is an object or a class:

* For objects, the method to control the invocation is in object.\_\_getattribute\_\_() which transforms b.x into the call type(b).\_\_dict\_\_['x'].\_\_get\_\_(b, type(b)). The implementation works through a precedence chain that gives data descriptors priority over instance variables, instance variables priority over non-data descriptors, and assigns lowest priority to \_\_getattr\_\_() if provided.
* For classes, the corresponing method is in the type class, i.e. type.\_\_getattribute\_\_() which transforms B.x into B.\_\_dict\_\_['x'].\_\_get\_\_(None, B).

\_\_getattribute\_\_ is not implemented in Python but in C. The following Python code is a simulation of the logic in Python. We can see that the descriptors are called by the \_\_getattribute\_\_ implementations.

**def** \_\_getattribute\_\_(self, key):

"Emulate type\_getattro() in Objects/typeobject.c"

v = type.\_\_getattribute\_\_(self, key)

**if** hasattr(v, '\_\_get\_\_'):

**return** v.\_\_get\_\_(**None**, self)

**return** v

m.\_\_getattribute\_\_("x")

<\_\_main\_\_.MyClass object at 0x7efe93ff4a20> <class '\_\_main\_\_.MyClass'>

Getting (Retrieving) self.val: yellow

Output: :

'yellow'

The object returned by super() also has a custom \_\_getattribute\_\_() method for invoking descriptors. The attribute lookup super(B, obj).m searches obj.\_\_class\_\_.\_\_mro\_\_ for the base class A immediately following B and then returns A.\_\_dict\_\_['m'].\_\_get\_\_(obj, B). If not a descriptor, m is returned unchanged. If not in the dictionary, m reverts to a search using object.\_\_getattribute\_\_().

The details above show that the mechanism for descriptors is embedded in the \_\_getattribute\_\_() methods for object, type, and super(). Classes inherit this machinery when they derive from object or if they have a meta-class providing similar functionality. This means also that one can turn-off automatic descriptor calls by overriding \_\_getattribute\_\_().

**from** **weakref** **import** WeakKeyDictionary

**class** **Voter**:

required\_age = 18 *# in Germany*

**def** \_\_init\_\_(self):

self.age = WeakKeyDictionary()

**def** \_\_get\_\_(self, instance\_obj, objtype):

**return** self.age.get(instance\_obj)

**def** \_\_set\_\_(self, instance, new\_age):

**if** new\_age < Voter.required\_age:

msg = '**{name}** is not old enough to vote in Germany'

**raise** **Exception**(msg.format(name=instance.name))

self.age[instance] = new\_age

print('**{name}** can vote in Germany'.format(

name=instance.name))

**def** \_\_delete\_\_(self, instance):

**del** self.age[instance]

**class** **Person**:

voter\_age = Voter()

**def** \_\_init\_\_(self, name, age):

self.name = name

self.voter\_age = age

p1 = Person('Ben', 23)

p2 = Person('Emilia', 22)

p2.voter\_age

Ben can vote in Germany

Emilia can vote in Germany

Output: :

22

A pure sample implementation of a property() class (<https://docs.python.org/3/howto/descriptor.html#properties>)

**class** **Property**:

"Emulate PyProperty\_Type() in Objects/descrobject.c"

**def** \_\_init\_\_(self, fget=**None**, fset=**None**, fdel=**None**, doc=**None**):

self.fget = fget

self.fset = fset

self.fdel = fdel

**if** doc **is** **None** **and** fget **is** **not** **None**:

doc = fget.\_\_doc\_\_

self.\_\_doc\_\_ = doc

**def** \_\_get\_\_(self, obj, objtype=**None**):

**if** obj **is** **None**:

**return** self

**if** self.fget **is** **None**:

**raise** **AttributeError**("unreadable attribute")

**return** self.fget(obj)

**def** \_\_set\_\_(self, obj, value):

**if** self.fset **is** **None**:

**raise** **AttributeError**("can't set attribute")

self.fset(obj, value)

**def** \_\_delete\_\_(self, obj):

**if** self.fdel **is** **None**:

**raise** **AttributeError**("can't delete attribute")

self.fdel(obj)

**def** getter(self, fget):

**return** type(self)(fget, self.fset, self.fdel, self.\_\_doc\_\_)

**def** setter(self, fset):

**return** type(self)(self.fget, fset, self.fdel, self.\_\_doc\_\_)

**def** deleter(self, fdel):

**return** type(self)(self.fget, self.fset, fdel, self.\_\_doc\_\_)

A simple class using our Property implementation.

**class** **A**:

**def** \_\_init\_\_(self, prop):

self.prop = prop

@Property

**def** prop(self):

print("The Property 'prop' will be returned now:")

**return** self.\_\_prop

@prop.setter

**def** prop(self, prop):

print("prop will be set")

self.\_\_prop = prop

Using our class A:

print("Initializing the Property 'prop' with the value 'Python'")

x = A("Python")

print("The value is: ", x.prop)

print("Reassigning the Property 'prop' to 'Python descriptors'")

x.prop = "Python descriptors"

print("The value is: ", x.prop)

Initializing the Property 'prop' with the value 'Python'

prop will be set

The Property 'prop' will be returned now:

The value is: Python

Reassigning the Property 'prop' to 'Python descriptors'

prop will be set

The Property 'prop' will be returned now:

The value is: Python descriptors

Let's make our Property implementation a little bit more talkative with some print functions to see what is going on:

**class** **Property**:

"Emulate PyProperty\_Type() in Objects/descrobject.c"

**def** \_\_init\_\_(self, fget=**None**, fset=**None**, fdel=**None**, doc=**None**):

print("**\n**\_\_init\_\_ of Property called with:")

print("fget=" + str(fget) + " fset=" + str(fset) + \

" fdel=" + str(fdel) + " doc=" + str(doc))

self.fget = fget

self.fset = fset

self.fdel = fdel

**if** doc **is** **None** **and** fget **is** **not** **None**:

doc = fget.\_\_doc\_\_

self.\_\_doc\_\_ = doc

**def** \_\_get\_\_(self, obj, objtype=**None**):

print("**\n**Property.\_\_get\_\_ has been called!")

**if** obj **is** **None**:

**return** self

**if** self.fget **is** **None**:

**raise** **AttributeError**("unreadable attribute")

**return** self.fget(obj)

**def** \_\_set\_\_(self, obj, value):

print("**\n**Property.\_\_set\_\_ has been called!")

**if** self.fset **is** **None**:

**raise** **AttributeError**("can't set attribute")

self.fset(obj, value)

**def** \_\_delete\_\_(self, obj):

print("**\n**Property.\_\_delete\_\_ has been called!")

**if** self.fdel **is** **None**:

**raise** **AttributeError**("can't delete attribute")

self.fdel(obj)

**def** getter(self, fget):

print("**\n**Property.getter has been called!")

**return** type(self)(fget, self.fset, self.fdel, self.\_\_doc\_\_)

**def** setter(self, fset):

print("**\n**Property.setter has been called!")

**return** type(self)(self.fget, fset, self.fdel, self.\_\_doc\_\_)

**def** deleter(self, fdel):

print("**\n**Property.deleter has been called!")

**return** type(self)(self.fget, self.fset, fdel, self.\_\_doc\_\_)

**class** **A**:

**def** \_\_init\_\_(self, prop):

self.prop = prop

@Property

**def** prop(self):

*""" This will be the doc string of the property """*

print("The Property 'prop' will be returned now:")

**return** self.\_\_prop

@prop.setter

**def** prop(self, prop):

print("prop will be set")

self.\_\_prop = prop

**def** prop2\_getter(self):

**return** self.\_\_prop2

**def** prop2\_setter(self, prop2):

self.\_\_prop2 = prop2

prop2 = Property(prop2\_getter, prop2\_setter)

print("Initializing the Property 'prop' with the value 'Python'")

x = A("Python")

print("The value is: ", x.prop)

print("Reassigning the Property 'prop' to 'Python descriptors'")

x.prop = "Python descriptors"

print("The value is: ", x.prop)

print(A.prop.getter(x))

**def** new\_prop\_setter(self, prop):

**if** prop=="foo":

self.\_\_prop = "foobar"

**else**:

self.\_\_prop = prop

A.prop.setter

\_\_init\_\_ of Property called with:

fget=<function A.prop at 0x7efe93fff840> fset=None fdel=None doc=None

Property.setter has been called!

\_\_init\_\_ of Property called with:

fget=<function A.prop at 0x7efe93fff840> fset=<function A.prop at 0x7efe9370f840> fdel=None doc= This will be the doc string of the property

\_\_init\_\_ of Property called with:

fget=<function A.prop2\_getter at 0x7efe93715048> fset=<function A.prop2\_setter at 0x7efe93715158> fdel=None doc=None

Initializing the Property 'prop' with the value 'Python'

Property.\_\_set\_\_ has been called!

prop will be set

Property.\_\_get\_\_ has been called!

The Property 'prop' will be returned now:

The value is: Python

Reassigning the Property 'prop' to 'Python descriptors'

Property.\_\_set\_\_ has been called!

prop will be set

Property.\_\_get\_\_ has been called!

The Property 'prop' will be returned now:

The value is: Python descriptors

Property.\_\_get\_\_ has been called!

Property.getter has been called!

\_\_init\_\_ of Property called with:

fget=<\_\_main\_\_.A object at 0x7efe9371b080> fset=<function A.prop at 0x7efe9370f840> fdel=None doc= This will be the doc string of the property

<\_\_main\_\_.Property object at 0x7efe93ff4358>

Property.\_\_get\_\_ has been called!

Output: :

<bound method Property.setter of <\_\_main\_\_.Property object at 0x7efe93729f98>>

**class** **Robot**:

**def** \_\_init\_\_(self, name="Marvin", city="Freiburg"):

self.name = name

self.city = city

@Property

**def** name(self):

**return** self.\_\_name

@name.setter

**def** name(self, name):

**if** name == "hello":

self.\_\_name = "hi"

**else**:

self.\_\_name = name

x = Robot("Marvin")

print(x.name)

x.name = "Eddie"

print(x.name)

\_\_init\_\_ of Property called with:

fget=<function Robot.name at 0x7efe93715ea0> fset=None fdel=None doc=None

Property.setter has been called!

\_\_init\_\_ of Property called with:

fget=<function Robot.name at 0x7efe93715ea0> fset=<function Robot.name at 0x7efe93715400> fdel=None doc=None

Property.\_\_set\_\_ has been called!

Property.\_\_get\_\_ has been called!

Marvin

Property.\_\_set\_\_ has been called!

Property.\_\_get\_\_ has been called!

Eddie

**class** **A**:

**def** a(func):

**def** helper(self, x):

**return** 4 \* func(self, x)

**return** helper

@a

**def** b(self, x):

**return** x + 1

a = A()

a.b(4)

Output: :

20

A lot of people ask, if it is possible to automatically create descriptors at runtime. This is possible as we show in the following example. On the other hand, this example is not very useful, because the getters and setters have no real functionality:

**class** **DynPropertyClass**(object):

**def** add\_property(self, attribute):

*""" add a property to the class """*

**def** get\_attribute(self):

*""" The value for attribute 'attribute' will be retrieved """*

**return** getattr(self, "\_" + type(x).\_\_name\_\_ + "\_\_" + attribute)

**def** set\_attribute(self, value):

*""" The value for attribute 'attribute' will be retrieved """*

*#setter = lambda self, value: self.setProperty(attribute, value)*

setattr(self, "\_" + type(x).\_\_name\_\_ + "\_\_" + attribute, value)

*#construct property attribute and add it to the class*

setattr(type(self), attribute, property(fget=get\_attribute,

fset=set\_attribute,

doc="Auto‑generated method"))

x = DynPropertyClass()

x.add\_property('name')

x.add\_property('city')

x.name = "Henry"

x.name

x.city = "Hamburg"

print(x.name, x.city)

print(x.\_\_dict\_\_)

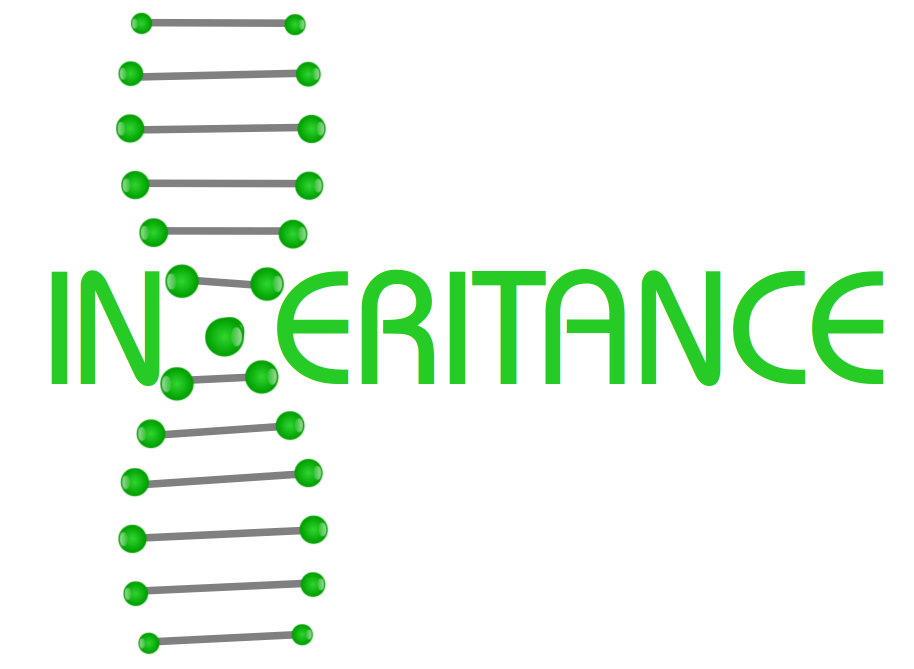
Henry Hamburg

{'\_DynPropertyClass\_\_name': 'Henry', '\_DynPropertyClass\_\_city': 'Hamburg'}

Previous Chapter: [Implementing a Property Decorator](https://www.python-course.eu/python3_implementing_a_property_decorator.php)  
Next Chapter: [Inheritance](https://www.python-course.eu/python3_inheritance.php)

## Inheritance

### Introduction and Definitions



No object-oriented programming language would be worthy to look at or use, if it didn't support inheritance. Inheritance was invented in 1969 for Simula. Python not only supports inheritance but multiple inheritance as well. Generally speaking, inheritance is the mechanism of deriving new classes from existing ones. By doing this, we get a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance (a "child object") acquires all, - though there are exceptions in some programming languages, - of the properties and behaviors of the parent object.

Inheritance allows programmers to create classes that are built upon existing classes, and this enables a class created through inheritance to inherit the attributes and methods of the parent class. This means that inheritance supports code reusability. The methods or generally speaking the software inherited by a subclass is considered to be reused in the subclass. The relationships of objects or classes through inheritance give rise to a directed graph.

The class from which a class inherits is called the parent or superclass. A class which inherits from a superclass is called a subclass, also called heir class or child class. Superclasses are sometimes called ancestors as well. There exists a hierarchical relationship between classes. It's similar to relationships or categorizations that we know from real life. Think about vehicles, for example. Bikes, cars, buses and trucks are vehicles. Pick-ups, vans, sports cars, convertibles and estate cars are all cars and by being cars they are vehicles as well. We could implement a vehicle class in Python, which might have methods like accelerate and brake. Cars, Buses and Trucks and Bikes can be implemented as subclasses which will inherit these methods from vehicle.

Classification of v
ehicles

### Syntax of Inheritance in Python

The syntax for a subclass definition looks like this:

class DerivedClassName(BaseClassName):

pass

Instead of the pass statement, there will be methods and attributes like in all other classes. The name BaseClassName must be defined in a scope containing the derived class definition.

Now we are ready for a simple inheritance example with Python code.

### Simple Inheritance Example

We will stick with our beloved robots or better Robot class from the previous chapters of our Python tutorial to show how the principle of inheritance works. We will define a class PhysicianRobot, which inherits from Robot.

**class** **Robot**:

**def** \_\_init\_\_(self, name):

self.name = name

**def** say\_hi(self):

print("Hi, I am " + self.name)

**class** **PhysicianRobot**(Robot):

**pass**

x = Robot("Marvin")

y = PhysicianRobot("James")

print(x, type(x))

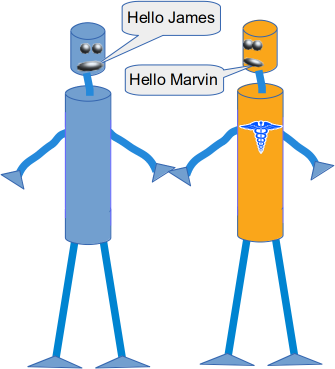
print(y, type(y))

y.say\_hi()

<\_\_main\_\_.Robot object at 0x7fd0080b3ba8> <class '\_\_main\_\_.Robot'>

<\_\_main\_\_.PhysicianRobot object at 0x7fd0080b3b70> <class '\_\_main\_\_.PhysicianRobot'>

Hi, I am James

  
  
If you look at the code of our PhysicianRobot class, you can see that we haven't defined any attributes or methods in this class. As the class PhysicianRobot is a subclass of Robot, it inherits, in this case, both the method \_\_init\_\_ and say\_hi. Inheriting these methods means that we can use them as if they were defined in the PhysicianRobot class. When we create an instance of PhysicianRobot, the \_\_init\_\_ function will also create a name attribute. We can apply the say\_hi method to the PhysisicianRobot object y, as we can see in the output from the code above.

### Difference between type and isinstance

You should also pay attention to the following facts, which we pointed out in other sections of our Python tutorial as well. People frequently ask where the difference between checking the type via the type function or the function isinstance is lies. The difference can be seen in the following code. We see that isinstance returns True if we compare an object either with the class it belongs to or with the superclass. Whereas the equality operator only returns True, if we compare an object with its own class.

x = Robot("Marvin")

y = PhysicianRobot("James")

print(isinstance(x, Robot), isinstance(y, Robot))

print(isinstance(x, PhysicianRobot))

print(isinstance(y, PhysicianRobot))

print(type(y) == Robot, type(y) == PhysicianRobot)

True True

False

True

False True

This is even true for arbitrary ancestors of the class in the inheritance line:

**class** **A**:

**pass**

**class** **B**(A):

**pass**

**class** **C**(B):

**pass**

x = C()

print(isinstance(x, A))

True

Now it should be clear, why [PEP 8](https://legacy.python.org/dev/peps/pep-0008/#programming-recommendations), the official Style Guide for Python code, says: "Object type comparisons should always use isinstance() instead of comparing types directly."

### Overriding

Let us get back to our new PhysicianRobot class. Imagine now that an instance of a PhysicianRobot should say hi in a different way. In this case, we have to redefine the method say\_hi inside of the subclass PhysicianRobot:

**class** **Robot**:

**def** \_\_init\_\_(self, name):

self.name = name

**def** say\_hi(self):

print("Hi, I am " + self.name)

**class** **PhysicianRobot**(Robot):

**def** say\_hi(self):

print("Everything will be okay! ")

print(self.name + " takes care of you!")

y = PhysicianRobot("James")

y.say\_hi()

Everything will be okay!

James takes care of you!

What we have done in the previous example is called overriding. A method of a parent class gets overridden by simply defining a method with the same name in the child class.

If a method is overridden in a class, the original method can still be accessed, but we have to do it by calling the method directly with the class name, i.e. Robot.say\_hi(y). We demonstrate this in the following code:

y = PhysicianRobot("Doc James")

y.say\_hi()

print("... and now the 'traditional' robot way of saying hi :-)")

Robot.say\_hi(y)

Everything will be okay!

Doc James takes care of you!

... and now the 'traditional' robot way of saying hi :-)

Hi, I am Doc James

We have seen that an inherited class can inherit and override methods from the superclass. Besides this a subclass often needs additional methods with additional functionalities, which do not exist in the superclass. An instance of the PhysicianRobot class will need for example the method heal so that the physician can do a proper job. We will also add an attribute health\_level to the Robot class, which can take a value between 0 and 1. The robots will 'come to live' with a random value between 0 and 1. If the health\_level of a Robot is below 0.8, it will need a doctor. We write a method needs\_a\_doctor which returns True if the value is below 0.8 and False otherwise. The 'healing' in the heal method is done by setting the health\_level to a random value between the old health\_level and 1. This value is calculated by the uniform function of the random module.

**import** **random**

**class** **Robot**:

**def** \_\_init\_\_(self, name):

self.name = name

self.health\_level = random.random()

**def** say\_hi(self):

print("Hi, I am " + self.name)

**def** needs\_a\_doctor(self):

**if** self.health\_level < 0.8:

**return** **True**

**else**:

**return** **False**

**class** **PhysicianRobot**(Robot):

**def** say\_hi(self):

print("Everything will be okay! ")

print(self.name + " takes care of you!")

**def** heal(self, robo):

robo.health\_level = random.uniform(robo.health\_level, 1)

print(robo.name + " has been healed by " + self.name + "!")

doc = PhysicianRobot("Dr. Frankenstein")

rob\_list = []

**for** i **in** range(5):

x = Robot("Marvin" + str(i))

**if** x.needs\_a\_doctor():

print("health\_level of " + x.name + " before healing: ", x.health\_level)

doc.heal(x)

print("health\_level of " + x.name + " after healing: ", x.health\_level)

rob\_list.append((x.name, x.health\_level))

print(rob\_list)

health\_level of Marvin0 before healing: 0.5562005305000016

Marvin0 has been healed by Dr. Frankenstein!

health\_level of Marvin0 after healing: 0.7807651150204282

health\_level of Marvin1 before healing: 0.40571527448692757

Marvin1 has been healed by Dr. Frankenstein!

health\_level of Marvin1 after healing: 0.4160992532325318

health\_level of Marvin2 before healing: 0.3786957462635925

Marvin2 has been healed by Dr. Frankenstein!

health\_level of Marvin2 after healing: 0.5474124864506639

health\_level of Marvin3 before healing: 0.6384666796845331

Marvin3 has been healed by Dr. Frankenstein!

health\_level of Marvin3 after healing: 0.6986491928780778

health\_level of Marvin4 before healing: 0.5983126049766974

Marvin4 has been healed by Dr. Frankenstein!

health\_level of Marvin4 after healing: 0.6988801787833587

[('Marvin0', 0.7807651150204282), ('Marvin1', 0.4160992532325318), ('Marvin2', 0.5474124864506639), ('Marvin3', 0.6986491928780778), ('Marvin4', 0.6988801787833587)]

When we override a method, we sometimes want to reuse the method of the parent class and at some new stuff. To demonstrate this, we will write a new version of the PhysicianRobot. say\_hi should return the text from the Robot class version plus the text " and I am a physician!"

**class** **PhysicianRobot**(Robot):

**def** say\_hi(self):

Robot.say\_hi(self)

print("and I am a physician!")

doc = PhysicianRobot("Dr. Frankenstein")

doc.say\_hi()

Hi, I am Dr. Frankenstein

and I am a physician!

We don't want to write redundant code and therefore we called Robot.say\_hi(self). We could also use the super function:

**class** **PhysicianRobot**(Robot):

**def** say\_hi(self):

super().say\_hi()

print("and I am a physician!")

doc = PhysicianRobot("Dr. Frankenstein")

doc.say\_hi()

Hi, I am Dr. Frankenstein

and I am a physician!

super is not realls necessary in this case. One could argue that it makes the code more maintainable, because we could change the name of the parent class, but this is seldom done anyway in existing classes. The real benefit of super shows when we use it with multiple inheritance.

### Distinction between Overwriting, Overloading and Overriding

#### Overwriting

If we overwrite a function, the original function will be gone. The function will be redefined. This process has nothing to do with object orientation or inheritance.

**def** f(x):

**return** x + 42

print(f(3))

*# f will be overwritten (or redefined) in the following:*

**def** f(x):

**return** x + 43

print(f(3))

45

46

#### Overloading

This subchapter will be only interesting for C++ and Java programmers who want to know how overloading can be accomplished in Python. Those who do not know about overloading will not miss it!

In the context of object-oriented programming, you might have heard about "overloading" as well. Even though "overloading" is not directly connected to OOP. Overloading is the ability to define a function with the same name multiple times. The definitions are different concerning the number of parameters and types of the parameters. It's the ability of one function to perform different tasks, depending on the number of parameters or the types of the parameters. We cannot overload functions like this in Python, but it is not necessary either.

This course is, however, not about C++ and we have so far avoided using any C++ code. We want to make an exception now, so that you can see, how overloading works in C++.

#include

#include

using namespace std;

int successor(int number) {

return number + 1;

}

double successor(double number) {

return number + 1;

}

int main() {

cout << successor(10) << endl;

cout << successor(10.3) << endl;

return 0;

}

We defined the successor function twice: One time for int and the other time with float as a Parameter. In Python the function can be defined like this, as you will know for sure:

In [ ]:

**def** successor(x):

**return** x + 1

As x is only a reference to an object, the Python function successor can be called with every object, even though it will create exceptions with many types. But it will work with int and float values!

Having a function with a different number of parameters is another way of function overloading. The following C++ program shows such an example. The function f can be called with either one or two integer arguments:

#include

using namespace std;

int f(int n);

int f(int n, int m);

int main() {

cout << "f(3): " << f(3) << endl;

cout << "f(3, 4): " << f(3, 4) << endl;

return 0;

}

int f(int n) {

return n + 42;

}

int f(int n, int m) {

return n + m + 42;

}

This doesn't work in Python, as we can see in the following example. The second definition of f with two parameters redefines or overrides the first definition with one argument. Overriding means that the first definition is not available anymore.

**def** f(n):

**return** n + 42

**def** f(n,m):

**return** n + m + 42

print(f(3, 4))

49

If you call f with only one parameter, you will raise an exception:

f(3)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

in

----> 1 f(3)

TypeError: f() missing 1 required positional argument: 'm'

Yet, it is possible to simulate the overloading behaviour of C++ in Python in this case with a default parameter:

**def** f(n, m=**None**):

**if** m:

**return** n + m +42

**else**:

**return** n + 42

print(f(3), f(1, 3))

45 46

The \* operator can be used as a more general approach for a family of functions with 1, 2, 3, or even more parameters:

**def** f(\*x):

**if** len(x) == 1:

**return** x[0] + 42

**elif** len(x) == 2:

**return** x[0] - x[1] + 5

**else**:

**return** 2 \* x[0] + x[1] + 42

print(f(3), f(1, 2), f(3, 2, 1))

45 4 50

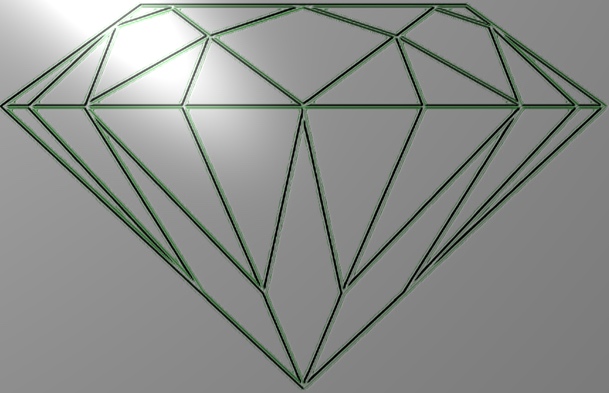
#### Overriding

Overriding is already explained above!

Previous Chapter: [Descriptors](https://www.python-course.eu/python3_descriptors.php)  
Next Chapter: [Multiple Inheritance](https://www.python-course.eu/python3_multiple_inheritance.php)

**Multiple Inheritance**

**Introduction**



In the previous chapter of our tutorial, we have covered inheritance, or more specific "single inheritance". As we have seen, a class inherits in this case from one class. Multiple inheritance on the other hand is a feature in which a class can inherit attributes and methods from more than one parent class. The critics point out that multiple inheritance comes along with a high level of complexity and ambiguity in situations such as the diamond problem. We will address this problem later in this chapter.

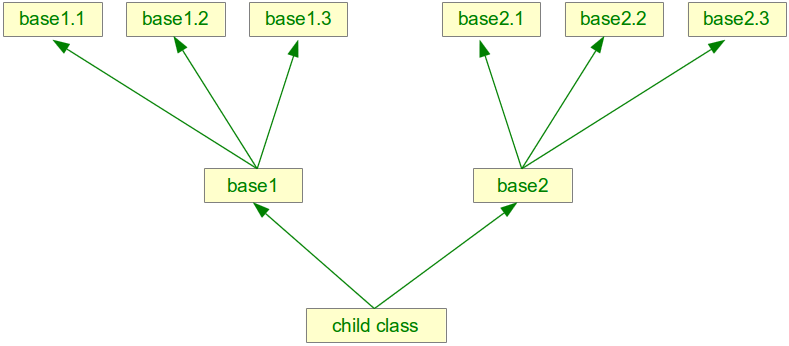
The widespread prejudice that multiple inheritance is something "dangerous" or "bad" is mostly nourished by programming languages with poorly implemented multiple inheritance mechanisms and above all by improper usage of it. Java doesn't even support multiple inheritance, while C++ supports it. Python has a sophisticated and well-designed approach to multiple inheritance.

A class definition, where a child class SubClassName inherits from the parent classes BaseClass1, BaseClass2, BaseClass3, and so on, looks like this:

class SubclassName(BaseClass1, BaseClass2, BaseClass3, ...):

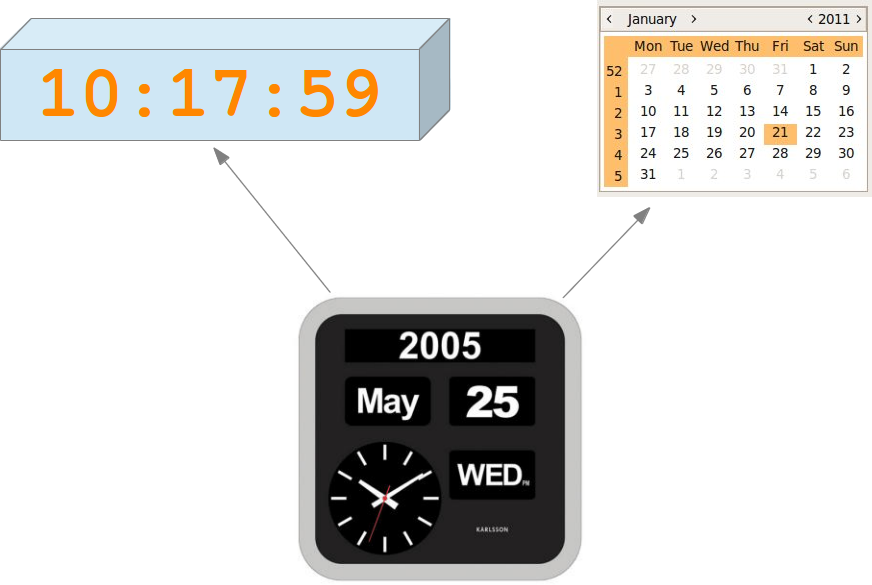
pass

It's clear that all the superclasses BaseClass1, BaseClass2, BaseClass3, ... can inherit from other superclasses as well. What we get is an inheritance tree.



**Example: CalendarClock**

We want to introduce the principles of multiple inheritance with an example. For this purpose, we will implement to independent classes: a "Clock" and a "Calendar" class. After this, we will introduce a class "CalendarClock", which is, as the name implies, a combination of "Clock" and "Calendar". CalendarClock inherits both from "Clock" and "Calendar".



The class Clock simulates the tick-tack of a clock. An instance of this class contains the time, which is stored in the attributes self.hours, self.minutes and self.seconds. Principally, we could have written the \_\_init\_\_ method and the set method like this:

def \_\_init\_\_(self,hours=0, minutes=0, seconds=0):

self.\_hours = hours

self.\_\_minutes = minutes

self.\_\_seconds = seconds

def set(self,hours, minutes, seconds=0):

self.\_hours = hours

self.\_\_minutes = minutes

self.\_\_seconds = seconds

We decided against this implementation, because we added additional code for checking the plausibility of the time data into the set method. We call the set method from the \_\_init\_\_ method as well, because we want to circumvent redundant code. The complete Clock class:

*"""*

*The class Clock is used to simulate a clock.*

*"""*

**class** **Clock**(object):

**def** \_\_init\_\_(self, hours, minutes, seconds):

*"""*

*The paramaters hours, minutes and seconds have to be*

*integers and must satisfy the following equations:*

*0 <= h < 24*

*0 <= m < 60*

*0 <= s < 60*

*"""*

self.set\_Clock(hours, minutes, seconds)

**def** set\_Clock(self, hours, minutes, seconds):

*"""*

*The parameters hours, minutes and seconds have to be*

*integers and must satisfy the following equations:*

*0 <= h < 24*

*0 <= m < 60*

*0 <= s < 60*

*"""*

**if** type(hours) == int **and** 0 <= hours **and** hours < 24:

self.\_hours = hours

**else**:

**raise** **TypeError**("Hours have to be integers between 0 and 23!")

**if** type(minutes) == int **and** 0 <= minutes **and** minutes < 60:

self.\_\_minutes = minutes

**else**:

**raise** **TypeError**("Minutes have to be integers between 0 and 59!")

**if** type(seconds) == int **and** 0 <= seconds **and** seconds < 60:

self.\_\_seconds = seconds

**else**:

**raise** **TypeError**("Seconds have to be integers between 0 and 59!")

**def** \_\_str\_\_(self):

**return** "**{0:02d}**:**{1:02d}**:**{2:02d}**".format(self.\_hours,

self.\_\_minutes,

self.\_\_seconds)

**def** tick(self):

*"""*

*This method lets the clock "tick", this means that the*

*internal time will be advanced by one second.*

*Examples:*

*>>> x = Clock(12,59,59)*

*>>> print(x)*

*12:59:59*

*>>> x.tick()*

*>>> print(x)*

*13:00:00*

*>>> x.tick()*

*>>> print(x)*

*13:00:01*

*"""*

**if** self.\_\_seconds == 59:

self.\_\_seconds = 0

**if** self.\_\_minutes == 59:

self.\_\_minutes = 0

**if** self.\_hours == 23:

self.\_hours = 0

**else**:

self.\_hours += 1

**else**:

self.\_\_minutes += 1

**else**:

self.\_\_seconds += 1

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Clock(23,59,59)

print(x)

x.tick()

print(x)

y = str(x)

print(type(y))

23:59:59

00:00:00

<class 'str'>

Let's check our exception handling by inputting floats and strings as input. We also check, what happens, if we exceed the limits of the expected values:

x = Clock(7.7, 45, 17)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-2-f91279ca09c6> in <module>

----> 1 x = Clock(7.7,45,17)

<ipython-input-1-ffb3e30af1a8> in \_\_init\_\_(self, hours, minutes, seconds)

**14** """

**15**

---> 16 self.set\_Clock(hours, minutes, seconds)

**17**

**18** def set\_Clock(self, hours, minutes, seconds):

<ipython-input-1-ffb3e30af1a8> in set\_Clock(self, hours, minutes, seconds)

**28** self.\_hours = hours

**29** else:

---> 30 raise TypeError("Hours have to be integers between 0 and 23!")

**31** if type(minutes) == int and 0 <= minutes and minutes < 60:

**32** self.\_\_minutes = minutes

TypeError: Hours have to be integers between 0 and 23!

x = Clock(24, 45, 17)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-4-36d0f83822b6> in <module>

----> 1 x = Clock(24, 45, 17)

<ipython-input-1-ffb3e30af1a8> in \_\_init\_\_(self, hours, minutes, seconds)

**14** """

**15**

---> 16 self.set\_Clock(hours, minutes, seconds)

**17**

**18** def set\_Clock(self, hours, minutes, seconds):

<ipython-input-1-ffb3e30af1a8> in set\_Clock(self, hours, minutes, seconds)

**28** self.\_hours = hours

**29** else:

---> 30 raise TypeError("Hours have to be integers between 0 and 23!")

**31** if type(minutes) == int and 0 <= minutes and minutes < 60:

**32** self.\_\_minutes = minutes

TypeError: Hours have to be integers between 0 and 23!

x = Clock(23, 60, 17)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-5-6c03e29741a6> in <module>

----> 1 x = Clock(23,60,17)

<ipython-input-1-ffb3e30af1a8> in \_\_init\_\_(self, hours, minutes, seconds)

**14** """

**15**

---> 16 self.set\_Clock(hours, minutes, seconds)

**17**

**18** def set\_Clock(self, hours, minutes, seconds):

<ipython-input-1-ffb3e30af1a8> in set\_Clock(self, hours, minutes, seconds)

**32** self.\_\_minutes = minutes

**33** else:

---> 34 raise TypeError("Minutes have to be integers between 0 and 59!")

**35** if type(seconds) == int and 0 <= seconds and seconds < 60:

**36** self.\_\_seconds = seconds

TypeError: Minutes have to be integers between 0 and 59!

x = Clock("23", "60", "17")

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-6-9d4de9fcfcdc> in <module>

----> 1 x = Clock("23", "60", "17")

<ipython-input-1-ffb3e30af1a8> in \_\_init\_\_(self, hours, minutes, seconds)

**14** """

**15**

---> 16 self.set\_Clock(hours, minutes, seconds)

**17**

**18** def set\_Clock(self, hours, minutes, seconds):

<ipython-input-1-ffb3e30af1a8> in set\_Clock(self, hours, minutes, seconds)

**28** self.\_hours = hours

**29** else:

---> 30 raise TypeError("Hours have to be integers between 0 and 23!")

**31** if type(minutes) == int and 0 <= minutes and minutes < 60:

**32** self.\_\_minutes = minutes

TypeError: Hours have to be integers between 0 and 23!

x = Clock(23, 17)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-7-dcf065c8d655> in <module>

----> 1 x = Clock(23, 17)

TypeError: \_\_init\_\_() missing 1 required positional argument: 'seconds'

We will now create a class "Calendar", which has lots of similarities to the previously defined Clock class. Instead of "tick" we have an "advance" method, which advances the date by one day, whenever it is called. Adding a day to a date is quite tricky. We have to check, if the date is the last day in a month and the number of days in the months vary. As if this isn't bad enough, we have February and the leap year problem.

The rules for calculating a leap year are the following:

* If a year is divisible by 400, it is a leap year.
* If a year is not divisible by 400 but by 100, it is not a leap year.
* A year number which is divisible by 4 but not by 100, it is a leap year.
* All other year numbers are common years, i.e. no leap years.

As a little useful gimmick, we added a possibility to output a date either in British or in American (Canadian) style.

*"""*

*The class Calendar implements a calendar.*

*"""*

**class** **Calendar**(object):

months = (31,28,31,30,31,30,31,31,30,31,30,31)

date\_style = "British"

@staticmethod

**def** leapyear(year):

*"""*

*The method leapyear returns True if the parameter year*

*is a leap year, False otherwise*

*"""*

**if** **not** year % 4 == 0:

**return** **False**

**elif** **not** year % 100 == 0:

**return** **True**

**elif** **not** year % 400 == 0:

**return** **False**

**else**:

**return** **True**

**def** \_\_init\_\_(self, d, m, y):

*"""*

*d, m, y have to be integer values and year has to be*

*a four digit year number*

*"""*

self.set\_Calendar(d,m,y)

**def** set\_Calendar(self, d, m, y):

*"""*

*d, m, y have to be integer values and year has to be*

*a four digit year number*

*"""*

**if** type(d) == int **and** type(m) == int **and** type(y) == int:

self.\_\_days = d

self.\_\_months = m

self.\_\_years = y

**else**:

**raise** **TypeError**("d, m, y have to be integers!")

**def** \_\_str\_\_(self):

**if** Calendar.date\_style == "British":

**return** "**{0:02d}**/**{1:02d}**/**{2:4d}**".format(self.\_\_days,

self.\_\_months,

self.\_\_years)

**else**:

*# assuming American style*

**return** "**{0:02d}**/**{1:02d}**/**{2:4d}**".format(self.\_\_months,

self.\_\_days,

self.\_\_years)

**def** advance(self):

*"""*

*This method advances to the next date.*

*"""*

max\_days = Calendar.months[self.\_\_months-1]

**if** self.\_\_months == 2 **and** Calendar.leapyear(self.\_\_years):

max\_days += 1

**if** self.\_\_days == max\_days:

self.\_\_days= 1

**if** self.\_\_months == 12:

self.\_\_months = 1

self.\_\_years += 1

**else**:

self.\_\_months += 1

**else**:

self.\_\_days += 1

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Calendar(31,12,2012)

print(x, end=" ")

x.advance()

print("after applying advance: ", x)

print("2012 was a leapyear:")

x = Calendar(28,2,2012)

print(x, end=" ")

x.advance()

print("after applying advance: ", x)

x = Calendar(28,2,2013)

print(x, end=" ")

x.advance()

print("after applying advance: ", x)

print("1900 no leapyear: number divisible by 100 but not by 400: ")

x = Calendar(28,2,1900)

print(x, end=" ")

x.advance()

print("after applying advance: ", x)

print("2000 was a leapyear, because number divisibe by 400: ")

x = Calendar(28,2,2000)

print(x, end=" ")

x.advance()

print("after applying advance: ", x)

print("Switching to American date style: ")

Calendar.date\_style = "American"

print("after applying advance: ", x)

31/12/2012 after applying advance: 01/01/2013

2012 was a leapyear:

28/02/2012 after applying advance: 29/02/2012

28/02/2013 after applying advance: 01/03/2013

1900 no leapyear: number divisible by 100 but not by 400:

28/02/1900 after applying advance: 01/03/1900

2000 was a leapyear, because number divisibe by 400:

28/02/2000 after applying advance: 29/02/2000

Switching to American date style:

after applying advance: 02/29/2000

At last, we will introduce our multiple inheritance example. We are now capable of implementing the originally intended class CalendarClock, which will inherit from both Clock and Calendar. The method "tick" of Clock will have to be overridden. However, the new tick method of CalendarClock has to call the tick method of Clock: Clock.tick(self)

*"""*

*Module, which implements the class CalendarClock.*

*"""*

**from** **clock** **import** Clock

**from** **calendar** **import** Calendar

**class** **CalendarClock**(Clock, Calendar):

*"""*

*The class CalendarClock implements a clock with integrated*

*calendar. It's a case of multiple inheritance, as it inherits*

*both from Clock and Calendar*

*"""*

**def** \_\_init\_\_(self, day, month, year, hour, minute, second):

Clock.\_\_init\_\_(self,hour, minute, second)

Calendar.\_\_init\_\_(self, day, month, year)

**def** tick(self):

*"""*

*advance the clock by one second*

*"""*

previous\_hour = self.\_hours

Clock.tick(self)

**if** (self.\_hours < previous\_hour):

self.advance()

**def** \_\_str\_\_(self):

**return** Calendar.\_\_str\_\_(self) + ", " + Clock.\_\_str\_\_(self)

**if** \_\_name\_\_ == "\_\_main\_\_":

x = CalendarClock(31, 12, 2013, 23, 59, 59)

print("One tick from ",x, end=" ")

x.tick()

print("to ", x)

x = CalendarClock(28, 2, 1900, 23, 59, 59)

print("One tick from ",x, end=" ")

x.tick()

print("to ", x)

x = CalendarClock(28, 2, 2000, 23, 59, 59)

print("One tick from ",x, end=" ")

x.tick()

print("to ", x)

x = CalendarClock(7, 2, 2013, 13, 55, 40)

print("One tick from ",x, end=" ")

x.tick()

print("to ", x)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-1-6b533ccbb057> in <module>

**33**

**34** if \_\_name\_\_ == "\_\_main\_\_":

---> 35 x = CalendarClock(31, 12, 2013, 23, 59, 59)

**36** print("One tick from ",x, end=" ")

**37** x.tick()

<ipython-input-1-6b533ccbb057> in \_\_init\_\_(self, day, month, year, hour, minute, second)

**16** def \_\_init\_\_(self, day, month, year, hour, minute, second):

**17** Clock.\_\_init\_\_(self,hour, minute, second)

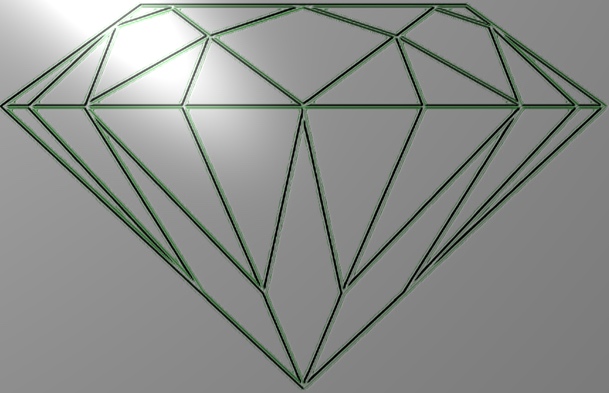
---> 18 Calendar.\_\_init\_\_(self, day, month, year)

**19**

**20**

TypeError: \_\_init\_\_() takes from 1 to 2 positional arguments but 4 were given

**The Diamond Problem or the ,,deadly diamond of death''**



The "diamond problem" (sometimes referred as the "deadly diamond of death") is the generally used term for an ambiguity that arises when two classes B and C inherit from a superclass A, and another class D inherits from both B and C. If there is a method "m" in A that B or C (or even both of them) has overridden, and furthermore, if it does not override this method, then the question is which version of the method does D inherit? It could be the one from A, B or C.

Let's look at Python. The first Diamond Problem configuration is like this: Both B and C override the method m of A:

**class** **A**:

**def** m(self):

print("m of A called")

**class** **B**(A):

**def** m(self):

print("m of B called")

**class** **C**(A):

**def** m(self):

print("m of C called")

**class** **D**(B,C):

**pass**

If you call the method m on an instance x of D, i.e. x.m(), we will get the output "m of B called". If we transpose the order of the classes in the class header of D in "class D(C,B):", we will get the output "m of C called".

The case in which m will be overridden only in one of the classes B or C, e.g. in C:

**class** **A**:

**def** m(self):

print("m of A called")

**class** **B**(A):

**pass**

**class** **C**(A):

**def** m(self):

print("m of C called")

**class** **D**(B,C):

**pass**

x = D()

x.m()

m of C called

Principially, two possibilities are imaginable: "m of C" or "m of A" could be used

We call this script with Python2.7 (python) and with Python3 (python3) to see what's happening:

$ python diamond1.py

m of A called

$ python3 diamond1.py

m of C called

Only for those who are interested in Python version2: To have the same inheritance behaviour in Python2 as in Python3, every class has to inherit from the class "object". Our class A doesn't inherit from object, so we get a so-called old-style class, if we call the script with python2. Multiple inheritance with old-style classes is governed by two rules: depth-first and then left-to-right. If you change the header line of A into "class A(object):", we will have the same behaviour in both Python versions.

**super and MRO**

We have seen in our previous implementation of the diamond problem, how Python "solves" the problem, i.e. in which order the base classes are browsed through. The order is defined by the so-called "Method Resolution Order" or in short MRO\*.

We will extend our previous example, so that every class defines its own method m:

**class** **A**:

**def** m(self):

print("m of A called")

**class** **B**(A):

**def** m(self):

print("m of B called")

**class** **C**(A):

**def** m(self):

print("m of C called")

**class** **D**(B,C):

**def** m(self):

print("m of D called")

Let's apply the method m on an instance of D. We can see that only the code of the method m of D will be executed. We can also explicitly call the methods m of the other classes via the class name, as we demonstrate in the following interactive Python session:

**from** **super1** **import** A,B,C,D

x = D()

B.m(x)

m of B called

C.m(x)

m of C called

A.m(x)

m of A called

Now let's assume that the method m of D should execute the code of m of B, C and A as well, when it is called. We could implement it like this:

**class** **D**(B,C):

**def** m(self):

print("m of D called")

B.m(self)

C.m(self)

A.m(self)

The output is what we have been looking for:

**from** **mro** **import** D

x = D()

x.m()

m of D called

m of B called

m of C called

m of A called

But it turns out once more that things are more complicated than they seem. How can we cope with the situation, if both m of B and m of C will have to call m of A as well. In this case, we have to take away the call A.m(self) from m in D. The code might look like this, but there is still a bug lurking in it:

**class** **A**:

**def** m(self):

print("m of A called")

**class** **B**(A):

**def** m(self):

print("m of B called")

A.m(self)

**class** **C**(A):

**def** m(self):

print("m of C called")

A.m(self)

**class** **D**(B,C):

**def** m(self):

print("m of D called")

B.m(self)

C.m(self)

The bug is that the method m of A will be called twice:

**from** **super3** **import** D

x = D()

x.m()

m of D called

m of B called

m of A called

m of C called

m of A called

One way to solve this problem - admittedly not a Pythonic one - consists in splitting the methods m of B and C in two methods. The first method, called \_m consists of the specific code for B and C and the other method is still called m, but consists now of a call self.\_m() and a call A.m(self). The code of the method m of D consists now of the specific code of D 'print("m of D called")', and the calls B.\_m(self), C.\_m(self) and A.m(self):

**class** **A**:

**def** m(self):

print("m of A called")

**class** **B**(A):

**def** \_m(self):

print("m of B called")

**def** m(self):

self.\_m()

A.m(self)

**class** **C**(A):

**def** \_m(self):

print("m of C called")

**def** m(self):

self.\_m()

A.m(self)

**class** **D**(B,C):

**def** m(self):

print("m of D called")

B.\_m(self)

C.\_m(self)

A.m(self)

Our problem is solved, but - as we have already mentioned - not in a pythonic way:

**from** **super4** **import** D

x = D()

x.m()

m of D called

m of B called

m of C called

m of A called

The optimal way to solve the problem, which is the "super" pythonic way, would be calling the super function:

**class** **A**:

**def** m(self):

print("m of A called")

**class** **B**(A):

**def** m(self):

print("m of B called")

super().m()

**class** **C**(A):

**def** m(self):

print("m of C called")

super().m()

**class** **D**(B,C):

**def** m(self):

print("m of D called")

super().m()

It also solves our problem, but in a beautiful design as well:

x = D()

x.m()

m of D called

m of B called

m of C called

m of A called

The super function is often used when instances are initialized with the \_\_init\_\_ method:

**class** **A**:

**def** \_\_init\_\_(self):

print("A.\_\_init\_\_")

**class** **B**(A):

**def** \_\_init\_\_(self):

print("B.\_\_init\_\_")

super().\_\_init\_\_()

**class** **C**(A):

**def** \_\_init\_\_(self):

print("C.\_\_init\_\_")

super().\_\_init\_\_()

**class** **D**(B,C):

**def** \_\_init\_\_(self):

print("D.\_\_init\_\_")

super().\_\_init\_\_()

We demonstrate the way of working in the following interactive session:

d = D()

D.\_\_init\_\_

B.\_\_init\_\_

C.\_\_init\_\_

A.\_\_init\_\_

c = C()

C.\_\_init\_\_

A.\_\_init\_\_

b = B()

B.\_\_init\_\_

A.\_\_init\_\_

a = A()

A.\_\_init\_\_

The question arises about how the super functions makes decisions. How does it decide which class has to be used? As we have already mentioned, it uses the so-called method resolution order(MRO). It is based on the "C3 superclass linearisation" algorithm. This is called a linearisation, because the tree structure is broken down into a linear order. The mro method can be used to create this list:

D.mro()

Output: :

[\_\_main\_\_.D, \_\_main\_\_.B, \_\_main\_\_.C, \_\_main\_\_.A, object]

B.mro()

Output: :

[\_\_main\_\_.B, \_\_main\_\_.A, object]

A.mro()

Output: :

[\_\_main\_\_.A, object]

**Polymorphism**



Polymorphism is construed from two Greek words. "Poly" stands for "much" or "many" and "morph" means shape or form. Polymorphism is the state or condition of being polymorphous, or if we use the translations of the components "the ability to be in many shapes or forms. Polymorphism is a term used in many scientific areas. In crystallography it defines the state, if something crystallizes into two or more chemically identical but crystallographically distinct forms. Biologists know polymorphism as the existence of an organism in several form or colour varieties. The Romans even had a god, called Morpheus, who is able to take any human form: Morheus appears in Ovid's metamorphoses and is the son of Somnus, the god of sleep. You can admire Morpheus and Iris in the picture on the right side.

So, before we fall asleep, we get back to Python and to what polymorphism means in the context of programming languages. Polymorphism in Computer Science is the ability to present the same interface for differing underlying forms. We can have in some programming languages polymorphic functions or methods, for example. Polymorphic functions or methods can be applied to arguments of different types, and they can behave differently depending on the type of the arguments to which they are applied. We can also define the same function name with a varying number of parameter.

Let's have a look at the following Python function:

**def** f(x, y):

print("values: ", x, y)

f(42, 43)

f(42, 43.7)

f(42.3, 43)

f(42.0, 43.9)

values: 42 43

values: 42 43.7

values: 42.3 43

values: 42.0 43.9

We can call this function with various types, as demonstrated in the example. In typed programming languages like Java or C++, we would have to overload f to implement the various type combinations.

Our example could be implemented like this in C++:

#include

using namespace std;

void f(int x, int y ) {

cout << "values: " << x << ", " << x << endl;

}

void f(int x, double y ) {

cout << "values: " << x << ", " << x << endl;

}

void f(double x, int y ) {

cout << "values: " << x << ", " << x << endl;

}

void f(double x, double y ) {

cout << "values: " << x << ", " << x << endl;

}

int main()

{

f(42, 43);

f(42, 43.7);

f(42.3,43);

f(42.0, 43.9);

}

Python is implicitly polymorphic. We can apply our previously defined function f even to lists, strings or other types, which can be printed:

**def** f(x,y):

print("values: ", x, y)

f([3,5,6],(3,5))

values: [3, 5, 6] (3, 5)

f("A String", ("A tuple", "with Strings"))

values: A String ('A tuple', 'with Strings')

f({2,3,9}, {"a":3.4,"b":7.8, "c":9.04})

values: {9, 2, 3} {'a': 3.4, 'b': 7.8, 'c': 9.04}

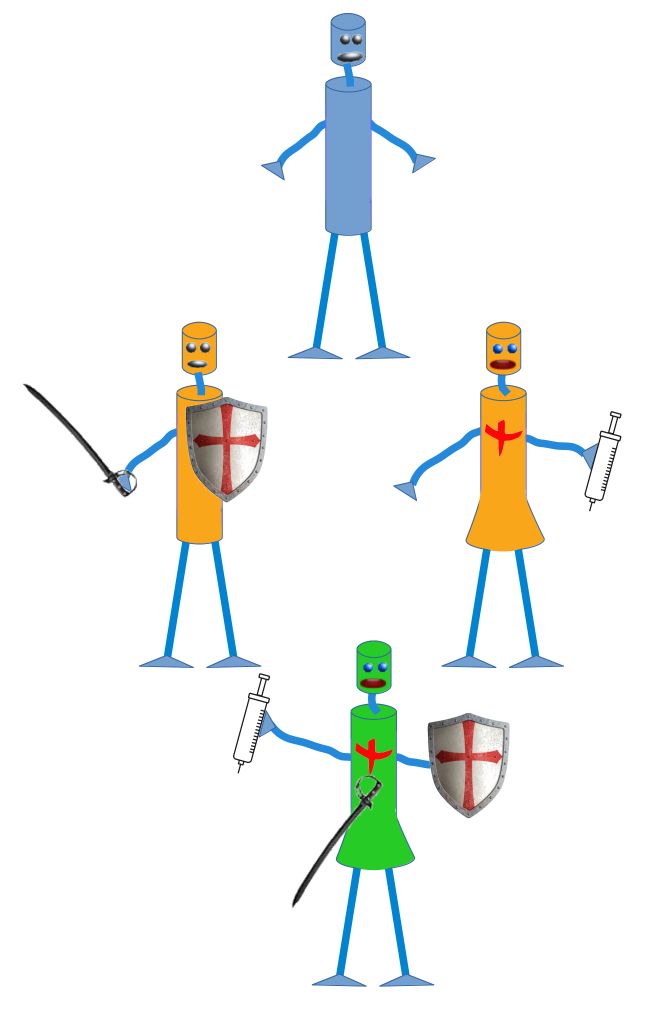
**Footnotes**

\* Python has used since 2.3 the ,,C3 superclass linearisation''-algorithm to determine the MRO. -->

Previous Chapter: [Inheritance](https://www.python-course.eu/python3_inheritance.php)  
Next Chapter: [Multiple Inheritance Example](https://www.python-course.eu/python3_multiple_inheritance_example.php)

## Multiple Inheritance: Example

### Robot Classes



This chapter of our tutorial is meant to deepen the understanding of multiple inheritance that the reader has built up in our previous chapter. We will provide a further extentive example for this important object oriented principle of the programming language Python. We will use a variation of our Robot class as the superclass. We will also summarize some other important aspects of object orientation with Python like properties. We will also work out the differences between overwriting, overloading and overriding.

This example has grown during my onsite Python training classes, because I urgently needed simple and easy to understand examples of subclassing and above all one for multiple inheritance.

Starting from the superclass Robot we will derive two classes: A FightingRobot class and a NursingRobot class.

Finally we will define a 'combination' of both the FightingRobot class and the NursingRobot class, i.e. we will implement a class FightingNurseRobot, which will inherit both from FightingRobot and NursingRobot.

Let us start with our Robot class: We use a private class attribute \_\_illegal\_names containing a set of names not allowed to be used for naming robots.

By providing an \_\_add\_\_ method we make sure that our robots are capable of propagating. The name of the resulting robot will be automatically created. The name of a 'baby' robot will consist of the concatenation of the names of both parents separated by an hyphen. If a parent name has a name containing a hyphen, we will use only the first part before the hyphen.

The robots will 'come to live' with a random value between 0 and 1 for the attribute health\_level. If the health\_level of a Robot is below a threshold, which is defined by the class attribute Robot.\_\_crucial\_health\_level, it will need the nursing powers of a robot from the NursingClass. To determine if a Robots needs healing, we provide a method needs\_a\_nurse which returns True if the value is below Robot.\_\_crucial\_health\_level and False otherwise.

**import** **random**

**class** **Robot**():

\_\_illegal\_names = {"Henry", "Oscar"}

\_\_crucial\_health\_level = 0.6

**def** \_\_init\_\_(self, name):

self.name = name *#---> property setter*

self.health\_level = random.random()

@property

**def** name(self):

**return** self.\_\_name

@name.setter

**def** name(self, name):

**if** name **in** Robot.\_\_illegal\_names:

self.\_\_name = "Marvin"

**else**:

self.\_\_name = name

**def** \_\_str\_\_(self):

**return** self.name + ", Robot"

**def** \_\_add\_\_(self, other):

first = self.name.split("-")[0]

second = other.name.split("-")[0]

**return** Robot(first + "-" + second)

**def** needs\_a\_nurse(self):

**if** self.health\_level < Robot.\_\_crucial\_health\_level:

**return** **True**

**else**:

**return** **False**

**def** say\_hi(self):

print("Hi, I am " + self.name)

print("My health level is: " + str(self.health\_level))

We can test the newly designed Robot class now. Watch out how the hyphened names change from generation to generation:

first\_generation = (Robot("Marvin"),

Robot("Enigma-Alan"),

Robot("Charles-Henry"))

gen1 = first\_generation *# used as an abbreviation*

babies = [gen1[0] + gen1[1], gen1[1] + gen1[2]]

babies.append(babies[0] + babies[1])

**for** baby **in** babies:

baby.say\_hi()

Hi, I am Marvin-Enigma

My health level is: 0.8879034405855623

Hi, I am Enigma-Charles

My health level is: 0.46930784534193803

Hi, I am Marvin-Enigma

My health level is: 0.2009467867758803

### Subclass NursingRobot

We are ready now for subclassing the Robot class. We will start by creating the NursingRobot class. We extend the \_\_init\_\_ method with a new attribute healing\_power. At first we have to understand the concept of 'healing power'. Generally, it makes only sense to heal a Robot, if its health level is below 1. The 'healing' in the heal method is done by setting the health\_level to a random value between the old health\_level and healing\_power of a ǸursingRobot. This value is calculated by the uniform function of the random module.

**class** **NursingRobot**(Robot):

**def** \_\_init\_\_(self, name="Hubert", healing\_power=**None**):

super().\_\_init\_\_(name)

**if** healing\_power **is** **None**:

self.healing\_power = random.uniform(0.8, 1)

**else**:

self.healing\_power = healing\_power

**def** say\_hi(self):

print("Well, well, everything will be fine ... " + self.name + " takes care of you!")

**def** say\_hi\_to\_doc(self):

Robot.say\_hi(self)

**def** heal(self, robo):

**if** robo.health\_level > self.healing\_power:

print(self.name + " not strong enough to heal " + robo.name)

**else**:

robo.health\_level = random.uniform(robo.health\_level, self.healing\_power)

print(robo.name + " has been healed by " + self.name + "!")

Let's heal the robot class instances which we created so far. If you look at the code, you may wonder about the function chain, which is a generator from the itertools module. Logically, the same thing happens, as if we had used first\_generation + babies, but chain is not creating a new list. chain is iterating over both lists, one after the other and this is efficient!

**from** **itertools** **import** chain

nurses = [NursingRobot("Hubert"),

NursingRobot("Emma", healing\_power=1)]

**for** nurse **in** nurses:

print("Healing power of " + nurse.name,

nurse.healing\_power)

print("**\n**Let's start the healing")

**for** robo **in** chain(first\_generation, babies):

robo.say\_hi()

**if** robo.needs\_a\_nurse():

*# choose randomly a nurse:*

nurse = random.choice(nurses)

nurse.heal(robo)

print("New health level: ", robo.health\_level)

**else**:

print(robo.name + " is healthy enough!")

print()

Healing power of Hubert 0.8730202054681965

Healing power of Emma 1

Let's start the healing

Hi, I am Marvin

My health level is: 0.3529060178554311

Marvin has been healed by Emma!

New health level: 0.9005591116354148

Hi, I am Enigma-Alan

My health level is: 0.0004876210374201717

Enigma-Alan has been healed by Emma!

New health level: 0.5750360827887102

Hi, I am Charles-Henry

My health level is: 0.06446896431387994

Charles-Henry has been healed by Hubert!

New health level: 0.09474181073613296

Hi, I am Marvin-Enigma

My health level is: 0.8879034405855623

Marvin-Enigma is healthy enough!

Hi, I am Enigma-Charles

My health level is: 0.46930784534193803

Enigma-Charles has been healed by Hubert!

New health level: 0.6182358513871242

Hi, I am Marvin-Enigma

My health level is: 0.2009467867758803

Marvin-Enigma has been healed by Emma!

New health level: 0.8204101216553784

An interesting question is, what would happen, if Hubert and Emma get added. The question is, what the resulting type will be:

x = nurses[0] + nurses[1]

x.say\_hi()

print(type(x))

Hi, I am Hubert-Emma

My health level is: 0.8844357538670625

<class '\_\_main\_\_.Robot'>

We see that the result of addition of two nursing robots is a plain robot of type Robot. This is not wrong but bad design. We want the resulting robots to be an instance of the NursingRobot class of course. One way to fix this would be to overload the \_\_add\_\_ method inside of the NursingRobot class:

def \_\_add\_\_(self, other):

first = self.name.split("-")[0]

second = other.name.split("-")[0]

return NursingRobot(first + "-" + second)

This is also bad design, because it is mainly a copy the original function with the only exception of creating an instance of NursingRobot instead of a Robot instance. An elegant solution would be having \_\_add\_\_ more generally defined. Instead of creating always a Robot instance, we could have made it dependent on the type of self by using type(self). For simplicity's sake we repeat the complete example:

**import** **random**

**class** **Robot**():

\_\_illegal\_names = {"Henry", "Oscar"}

\_\_crucial\_health\_level = 0.6

**def** \_\_init\_\_(self, name):

self.name = name *#---> property setter*

self.health\_level = random.random()

@property

**def** name(self):

**return** self.\_\_name

@name.setter

**def** name(self, name):

**if** name **in** Robot.\_\_illegal\_names:

self.\_\_name = "Marvin"

**else**:

self.\_\_name = name

**def** \_\_str\_\_(self):

**return** self.name + ", Robot"

**def** \_\_add\_\_(self, other):

first = self.name.split("-")[0]

second = other.name.split("-")[0]

**return** type(self)(first + "-" + second)

**def** needs\_a\_nurse(self):

**if** self.health\_level < Robot.\_\_crucial\_health\_level:

**return** **True**

**else**:

**return** **False**

**def** say\_hi(self):

print("Hi, I am " + self.name)

print("My health level is: " + str(self.health\_level))

**class** **NursingRobot**(Robot):

**def** \_\_init\_\_(self, name="Hubert", healing\_power=**None**):

super().\_\_init\_\_(name)

**if** healing\_power:

self.healing\_power = healing\_power

**else**:

self.healing\_power = random.uniform(0.8, 1)

**def** say\_hi(self):

print("Well, well, everything will be fine ... " + self.name + " takes care of you!")

**def** say\_hi\_to\_doc(self):

Robot.say\_hi(self)

**def** heal(self, robo):

**if** robo.health\_level > self.healing\_power:

print(self.name + " not strong enough to heal " + robo.name)

**else**:

robo.health\_level = random.uniform(robo.health\_level, self.healing\_power)

print(robo.name + " has been healed by " + self.name + "!")

### Subclass FightingRobot

Unfortunately, our virtual robot world is not better than their human counterpart. In other words, there will be some fighting going on as well. We subclass Robot once again to create a class with the name FightingRobot.

**class** **FightingRobot**(Robot):

\_\_maximum\_damage = 0.2

**def** \_\_init\_\_(self, name="Hubert",

fighting\_power=**None**):

super().\_\_init\_\_(name)

**if** fighting\_power:

self.fighting\_power = fighting\_power

**else**:

max\_dam = FightingRobot.\_\_maximum\_damage

self.fighting\_power = random.uniform(max\_dam, 1)

**def** say\_hi(self):

print("I am the terrible ... " + self.name)

**def** attack(self, other):

other.health\_level = \

other.health\_level \* self.fighting\_power

**if** isinstance(other, FightingRobot):

*# the other robot fights back*

self.health\_level = \

self.health\_level \* other.fighting\_power

Let us see now, how the fighting works:

fighters = (FightingRobot("Rambo", 0.4),

FightingRobot("Terminator", 0.2))

**for** robo **in** first\_generation:

print(robo, robo.health\_level)

fighters[0].attack(robo)

print(robo, robo.health\_level)

Marvin, Robot 0.9005591116354148

Marvin, Robot 0.36022364465416595

Enigma-Alan, Robot 0.5750360827887102

Enigma-Alan, Robot 0.23001443311548408

Charles-Henry, Robot 0.09474181073613296

Charles-Henry, Robot 0.037896724294453184

What about Rambo fighting Terminator? This spectacular fight can be viewed in the following code:

*# let us make them healthier first:*

print("Before the battle:")

**for** fighter **in** fighters:

nurses[1].heal(fighter)

print(fighter,

fighter.health\_level,

fighter.fighting\_power)

fighters[0].attack(fighters[1])

print("**\n**After the battle:")

**for** fighter **in** fighters:

print(fighter,

fighter.health\_level,

fighter.fighting\_power)

Before the battle:

Rambo has been healed by Emma!

Rambo, Robot 0.8481669878334196 0.4

Terminator has been healed by Emma!

Terminator, Robot 0.8458447223409832 0.2

After the battle:

Rambo, Robot 0.16963339756668394 0.4

Terminator, Robot 0.33833788893639327 0.2

### An Example of Multiple Inheritance

The underlying idea of the following class FightingNurseRobot consists in having robots who can both heal and fight.

**class** **FightingNurseRobot**(NursingRobot, FightingRobot):

**def** \_\_init\_\_(self, name, mode="nursing"):

super().\_\_init\_\_(name)

self.mode = mode *# alternatively "fighting"*

**def** say\_hi(self):

**if** self.mode == "fighting":

FightingRobot.say\_hi(self)

**elif** self.mode == "nursing":

NursingRobot.say\_hi(self)

**else**:

Robot.say\_hi(self)

We will instantiate two instances of FightingNurseRobot. You can see that after creation they are capable to heal themselves if necessary. They can also attack other robots.

fn1 = FightingNurseRobot("Donald", mode="fighting")

fn2 = FightingNurseRobot("Angela")

**if** fn1.needs\_a\_nurse():

fn1.heal(fn1)

**if** fn2.needs\_a\_nurse():

fn2.heal(fn2)

print(fn1.health\_level, fn2.health\_level)

fn1.say\_hi()

fn2.say\_hi()

fn1.attack(fn2)

print(fn1.health\_level, fn2.health\_level)

Angela has been healed by Angela!

0.8198049162122293 0.21016386954814018

I am the terrible ... Donald

Well, well, everything will be fine ... Angela takes care of you!

0.7176570622156614 0.16365553164488025

Previous Chapter: [Multiple Inheritance](https://www.python-course.eu/python3_multiple_inheritance.php)  
Next Chapter: [Magic Methods and Operator Overloading](https://www.python-course.eu/python3_magic_methods.php)

**Magic Methods and Operator Overloading**

**Introduction**



The so-called magic methods have nothing to do with wizardry. You have already seen them in the previous chapters of our tutorial. They are special methods with fixed names. They are the methods with this clumsy syntax, i.e. the double underscores at the beginning and the end. They are also hard to talk about. How do you pronounce or say a method name like \_\_init\_\_? "Underscore underscore init underscore underscore" sounds horrible and is almost a tongue twister. "Double underscore init double underscore" is a lot better, but the ideal way is "dunder init dunder"*That's why magic methods methods are sometimes called dunder methods*!

So what's magic about the \_\_init\_\_ method? The answer is, you don't have to invoke it directly. The invocation is realized behind the scenes. When you create an instance x of a class A with the statement "x = A()", Python will do the necessary calls to \_\_new\_\_ and \_\_init\_\_.

Towards the end of this chapter of our tutorial we will introduce the \_\_call\_\_ method. It is overlooked by many beginners and even advanced programmers of Python. It is a functionality which many programming languages do not have, so programmers generally do not look for it. The \_\_call\_\_ method enables Python programmers to write classes where the instances behave like functions. Both functions and the instances of such classes are called callables.

We have encountered the concept of operator overloading many times in the course of this tutorial. We had used the plus sign to add numerical values, to concatenate strings or to combine lists:

4 + 5

Output: :

9

3.8 + 9

Output: :

12.8

"Peter" + " " + "Pan"

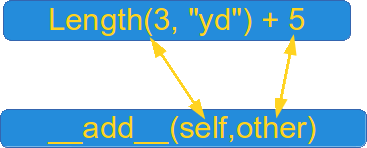
Output: :

'Peter Pan'

[3,6,8] + [7,11,13]

Output: :

[3, 6, 8, 7, 11, 13]



It's even possible to overload the "+" operator as well as all the other operators for the purposes of your own class. To do this, you need to understand the underlying mechanism. There is a special (or a "magic") method for every operator sign. The magic method for the "+" sign is the \_\_add\_\_ method. For "-" it is \_\_sub\_\_ and so on. We have a complete listing of all the magic methods a little further down.

The mechanism works like this: If we have an expression "x + y" and x is an instance of class K, then Python will check the class definition of K. If K has a method \_\_add\_\_ it will be called with x.\_\_add\_\_(y), otherwise we will get an error message:

Traceback (most recent call last):

File "", line 1, in

TypeError: unsupported operand type(s) for +: 'K' and 'K'

**Overview of Magic Methods**

**Binary Operators**

|  |  |
| --- | --- |
| **Operator** | **Method** |
| + | object.\_\_add\_\_(self, other) |
| - | object.\_\_sub\_\_(self, other) |
| \* | object.\_\_mul\_\_(self, other) |
| // | object.\_\_floordiv\_\_(self, other) |
| / | object.\_\_truediv\_\_(self, other) |
| % | object.\_\_mod\_\_(self, other) |
| \*\* | object.\_\_pow\_\_(self, other[, modulo]) |
| << | object.\_\_lshift\_\_(self, other) |
| >> | object.\_\_rshift\_\_(self, other) |
| & | object.\_\_and\_\_(self, other) |
| ^ | object.\_\_xor\_\_(self, other) |
| | | object.\_\_or\_\_(self, other) |

**Extended Assignments**

|  |  |
| --- | --- |
| **Operator** | **Method** |
| += | object.\_\_iadd\_\_(self, other) |
| -= | object.\_\_isub\_\_(self, other) |
| \*= | object.\_\_imul\_\_(self, other) |
| /= | object.\_\_idiv\_\_(self, other) |
| //= | object.\_\_ifloordiv\_\_(self, other) |
| %= | object.\_\_imod\_\_(self, other) |
| \*\*= | object.\_\_ipow\_\_(self, other[, modulo]) |
| <<= | object.\_\_ilshift\_\_(self, other) |
| >>= | object.\_\_irshift\_\_(self, other) |
| &= | object.\_\_iand\_\_(self, other) |
| ^= | object.\_\_ixor\_\_(self, other) |
| |= | object.\_\_ior\_\_(self, other) |

**Unary Operators**

|  |  |
| --- | --- |
| **Operator** | **Method** |
| - | object.\_\_neg\_\_(self) |
| + | object.\_\_pos\_\_(self) |
| abs() | object.\_\_abs\_\_(self) |
| ~ | object.\_\_invert\_\_(self) |
| complex() | object.\_\_complex\_\_(self) |
| int() | object.\_\_int\_\_(self) |
| long() | object.\_\_long\_\_(self) |
| float() | object.\_\_float\_\_(self) |
| oct() | object.\_\_oct\_\_(self) |
| hex() | object.\_\_hex\_\_(self |

**Comparison Operators**

|  |  |
| --- | --- |
| **Operator** | **Method** |
| < | object.\_\_lt\_\_(self, other) |
| <= | object.\_\_le\_\_(self, other) |
| == | object.\_\_eq\_\_(self, other) |
| != | object.\_\_ne\_\_(self, other) |
| >= | object.\_\_ge\_\_(self, other) |
| > | object.\_\_gt\_\_(self, other) |
|  |  |

**Example class: Length**

We will demonstrate the Length class and how you can overload the "+" operator for your own class. To do this, we have to overload the \_\_add\_\_ method. Our class contains the \_\_str\_\_ and \_\_repr\_\_ methods as well. The instances of the class Length contain length or distance information. The attributes of an instance are self.value and self.unit.

This class allows us to calculate expressions with mixed units like this one:

2.56 m + 3 yd + 7.8 in + 7.03 cm

The class can be used like this:

**from** **unit\_conversions** **import** Length

L = Length

print(L(2.56,"m") + L(3,"yd") + L(7.8,"in") + L(7.03,"cm"))

5.57162

The listing of the class:

**class** **Length**:

\_\_metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,

"in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,

"mi" : 1609.344 }

**def** \_\_init\_\_(self, value, unit = "m" ):

self.value = value

self.unit = unit

**def** Converse2Metres(self):

**return** self.value \* Length.\_\_metric[self.unit]

**def** \_\_add\_\_(self, other):

l = self.Converse2Metres() + other.Converse2Metres()

**return** Length(l / Length.\_\_metric[self.unit], self.unit )

**def** \_\_str\_\_(self):

**return** str(self.Converse2Metres())

**def** \_\_repr\_\_(self):

**return** "Length(" + str(self.value) + ", '" + self.unit + "')"

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Length(4)

print(x)

y = eval(repr(x))

z = Length(4.5, "yd") + Length(1)

print(repr(z))

print(z)

4

Length(5.593613298337708, 'yd')

5.1148

We use the method \_\_iadd\_\_ to implement the extended assignment:

**def** \_\_iadd\_\_(self, other):

l = self.Converse2Metres() + other.Converse2Metres()

self.value = l / Length.\_\_metric[self.unit]

**return** self

Now we are capable of writing the following assignments:

x += Length(1)

x += Length(4, "yd")

We added 1 metre in the example above by writing "x += Length(1))". Most certainly, you will agree with us that it would be more convenient to simply write "x += 1" instead. We also want to treat expressions like "Length(5,"yd") + 4.8" similarly. So, if somebody uses a type int or float, our class takes it automatically for "metre" and converts it into a Length object. It's easy to adapt our \_\_add\_\_ and \_\_iadd\_\_ method for this task. All we have to do is to check the type of the parameter "other":

**def** \_\_add\_\_(self, other):

**if** type(other) == int **or** type(other) == float:

l = self.Converse2Metres() + other

**else**:

l = self.Converse2Metres() + other.Converse2Metres()

**return** Length(l / Length.\_\_metric[self.unit], self.unit )

**def** \_\_iadd\_\_(self, other):

**if** type(other) == int **or** type(other) == float:

l = self.Converse2Metres() + other

**else**:

l = self.Converse2Metres() + other.Converse2Metres()

self.value = l / Length.\_\_metric[self.unit]

**return** self

It's a safe bet that if somebody works with adding integers and floats from the right side for a while, he or she will want to have the same from the left side! SWhat will happen, if we execute the following code line:

x = 5 + Length(3, "yd")

We will get an exception:

AttributeError: 'int' object has no attribute 'Converse2Metres'

Of course, the left side has to be of type "Length", because otherwise Python tries to apply the \_\_add\_\_ method from int, which can't cope with Length objects as second arguments!

Python provides a solution for this problem as well. It's the \_\_radd\_\_ method. It works like this: Python tries to evaluate the expression "5 + Length(3, 'yd')". First it calls int.\_\_add\_\_(5,Length(3, 'yd')), which will raise an exception. After this it will try to invoke Length.\_\_radd\_\_(Length(3, "yd"), 5). It's easy to recognize that the implementation of \_\_radd\_\_ is analogue to \_\_add\_\_:

**def** \_\_radd\_\_(self, other):

**if** type(other) == int **or** type(other) == float:

l = self.Converse2Metres() + otherLength.\_\_radd\_\_(Length(3, "yd"), 5)

**else**:

l = self.Converse2Metres() + other.Converse2Metres()

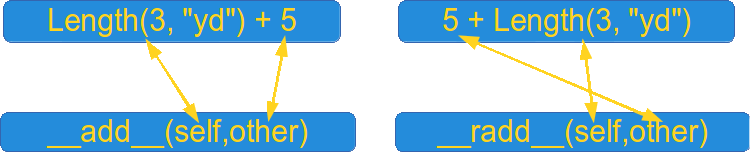
**return** Length(l / Length.\_\_metric[self.unit], self.unit )

It's advisable to make use of the \_\_add\_\_ method in the \_\_radd\_\_ method:

**def** \_\_radd\_\_(self, other):

**return** Length.\_\_add\_\_(self,other)

The following diagram illustrates the relationship between \_\_add\_\_ and \_\_radd\_\_:



**The call method**

Before we will come to the \_\_call\_\_ method, we have to know what a callable is. In general, a "callable" is an object that can be called like a function and behaves like one. All functions are also callables. Python provides a function with the name callable. With the help of this funciton we can determine whether an object is callable or not. The function callable returns a Boolean truth value which indicates whether the object passed as an argument can be called like a function or not. In addition to functions, we have already seen another form of callables: classes

**def** the\_answer(question):

**return** 42

print("the\_answer: ", callable(the\_answer))

the\_answer: True

The \_\_call\_\_ method can be used to turn the instances of the class into callables. Functions are callable objects. A callable object is an object which can be used and behaves like a function but might not be a function. By using the \_\_call\_\_ method it is possible to define classes in a way that the instances will be callable objects. The \_\_call\_\_ method is called, if the instance is called "like a function", i.e. using brackets. The following class definition is the simplest possible way to define a class with a \_\_call\_\_ method.

**class** **FoodSupply**:

**def** \_\_call\_\_(self):

**return** "spam"

foo = FoodSupply()

bar = FoodSupply()

print(foo(), bar())

spam spam

The previous class example is extremely simple, but useless in practical terms. Whenever we create an instance of the class, we get a callable. These callables are always defining the same constant function. A function without any input and a constant output "spam". We'll now define a class which is slightly more useful. We define a class with the name TriangleArea. This class has only one method, which is the \_\_call\_\_method. The \_\_call\_\_ method calculates the area of an arbitrary triangle, if the length of the three sides are given.

**class** **TriangleArea**:

**def** \_\_call\_\_(self, a, b, c):

p = (a + b + c) / 2

result = (p \* (p - a) \* (p - b) \* (p - c)) \*\* 0.5

**return** result

area = TriangleArea()

print(area(3, 4, 5))

6.0

This program returns 6.0. This class is not very exciting, even though we can create an arbitrary number of instances where each instance just executes an unaltered \_\_call\_\_ function of the TrianlgeClass. We cannot pass parameters to the instanciation and the \_\_call\_\_ of each instance returns the value of the area of the triangle. So each instance behaves like the area function.

After the two very didactic and not very practical examples, we want to demonstrate a more practical example with the following. We define a class that can be used to define linear equations:

**class** **StraightLines**():

**def** \_\_init\_\_(self, m, c):

self.slope = m

self.y\_intercept = c

**def** \_\_call\_\_(self, x):

**return** self.slope \* x + self.y\_intercept

line = StraightLines(0.4, 3)

**for** x **in** range(-5, 6):

print(x, line(x))

-5 1.0

-4 1.4

-3 1.7999999999999998

-2 2.2

-1 2.6

0 3.0

1 3.4

2 3.8

3 4.2

4 4.6

5 5.0

We will use this class now to create some straight lines and visualize them with matplotlib:

lines = []

lines.append(StraightLines(1, 0))

lines.append(StraightLines(0.5, 3))

lines.append(StraightLines(-1.4, 1.6))

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

X = np.linspace(-5,5,100)

**for** index, line **in** enumerate(lines):

line = np.vectorize(line)

plt.plot(X, line(X), label='line' + str(index))

plt.title('Some straight lines')

plt.xlabel('x', color='#1C2833')

plt.ylabel('y', color='#1C2833')

plt.legend(loc='upper left')

plt.grid()

plt.show()

Chart, line chart

Description automatically generated

Our next example is also exciting. The class FuzzyTriangleArea defines a \_\_call\_\_ method which implements a fuzzy behaviour in the calculations of the area. The result should be correct with a likelihood of p, e.g. 0.8. If the result is not correct the result will be in a range of ± v %. e.g. 0.1.

**import** **random**

**class** **FuzzyTriangleArea**:

**def** \_\_init\_\_(self, p=0.8, v=0.1):

self.p, self.v = p, v

**def** \_\_call\_\_(self, a, b, c):

p = (a + b + c) / 2

result = (p \* (p - a) \* (p - b) \* (p - c)) \*\* 0.5

**if** random.random() <= self.p:

**return** result

**else**:

**return** random.uniform(result-self.v,

result+self.v)

area1 = FuzzyTriangleArea()

area2 = FuzzyTriangleArea(0.5, 0.2)

**for** i **in** range(12):

print(f"{area1(3, 4, 5):4.3f}, {area2(3, 4, 5):4.2f}")

6.000, 6.00

5.960, 6.00

6.000, 6.00

6.000, 6.00

6.000, 5.92

6.000, 6.00

6.000, 6.05

6.000, 5.81

6.000, 6.17

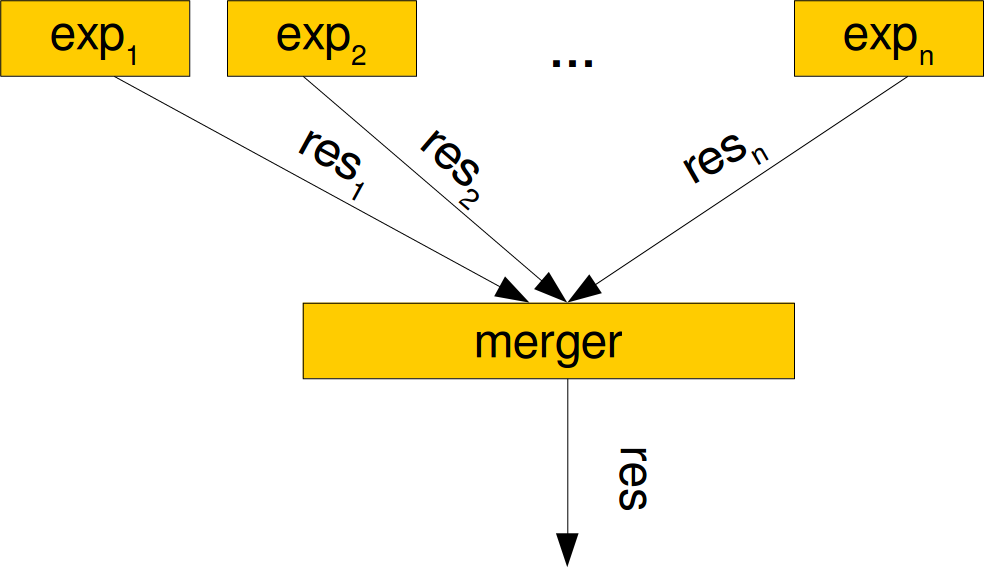
6.000, 6.00

6.000, 6.00

6.000, 6.07

Beware that this output differs with every call! We can see the in most cases we get the right value for the area but sometimes not.

We can create many different instances of the previous class. Each of these behaves like an area function, which returns a value for the area, which may or may not be correct, depending on the instantiation parameters p and v. We can see those instances as experts (expert functions) which return in most cases the correct answer, if we use p values close to 1. If the value v is close to zero, the error will be small, if at all. The next task would be merging such experts, let's call them exp1, exp2, ..., expn to get an improved result. We can perform a vote on the results, i.e. we will return the value which is most often occuring, the correct value. Alternatively, we can calculate the arithmetic mean. We will implement both possibilities in our class FuzzyTriangleArea:



**from** **random** **import** uniform, random

**from** **collections** **import** Counter

**class** **FuzzyTriangleArea**:

**def** \_\_init\_\_(self, p=0.8, v=0.1):

self.p, self.v = p, v

**def** \_\_call\_\_(self, a, b, c):

p = (a + b + c) / 2

result = (p \* (p - a) \* (p - b) \* (p - c)) \*\* 0.5

**if** random() <= self.p:

**return** result

**else**:

**return** uniform(result-self.v,

result+self.v)

**class** **MergeExperts**:

**def** \_\_init\_\_(self, mode, \*experts):

self.mode, self.experts = mode, experts

**def** \_\_call\_\_(self, a, b, c):

results= [exp(a, b, c) **for** exp **in** self.experts]

**if** self.mode == "vote":

c = Counter(results)

**return** c.most\_common(1)[0][0]

**elif** self.mode == "mean":

**return** sum(results) / len(results)

rvalues = [(uniform(0.7, 0.9), uniform(0.05, 0.2)) **for** \_ **in** range(20)]

experts = [FuzzyTriangleArea(p, v) **for** p, v **in** rvalues]

merger1 = MergeExperts("vote", \*experts)

print(merger1(3, 4, 5))

merger2 = MergeExperts("mean", \*experts)

print(merger2(3, 4, 5))

6.0

6.004827882562133

The following example defines a class with which we can create abitrary polynomial functions:

**class** **Polynomial**:

**def** \_\_init\_\_(self, \*coefficients):

self.coefficients = coefficients[::-1]

**def** \_\_call\_\_(self, x):

res = 0

**for** index, coeff **in** enumerate(self.coefficients):

res += coeff \* x\*\* index

**return** res

*# a constant function*

p1 = Polynomial(42)

*# a straight Line*

p2 = Polynomial(0.75, 2)

*# a third degree Polynomial*

p3 = Polynomial(1, -0.5, 0.75, 2)

**for** i **in** range(1, 10):

print(i, p1(i), p2(i), p3(i))

1 42 2.75 3.25

2 42 3.5 9.5

3 42 4.25 26.75

4 42 5.0 61.0

5 42 5.75 118.25

6 42 6.5 204.5

7 42 7.25 325.75

8 42 8.0 488.0

9 42 8.75 697.25

You will find further interesting examples of the \_\_call\_\_ function in our tutorial in the chapters [Decorators](https://www.python-course.eu/python3_decorators.php) and [Memoization with Decorators](https://www.python-course.eu/python3_memoization.php). You may also consult our chapter on [Polynomials](https://www.python-course.eu/polynomial_class_in_python.php).

**Standard Classes as Base Classes**

It's possible to use standard classes - like int, float, dict or lists - as base classes as well.

We extend the list class by adding a push method:

**class** **Plist**(list):

**def** \_\_init\_\_(self, l):

list.\_\_init\_\_(self, l)

**def** push(self, item):

self.append(item)

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Plist([3,4])

x.push(47)

print(x)

[3, 4, 47]

This means that all the previously introduced binary and extended assignment operators exist in the "reversed" version as well:  
\_\_radd\_\_ , \_\_rsub\_\_ , \_\_rmul\_\_ etc.

**Exercises**



1) Write a class with the name Ccy, similar to the previously defined Length class.Ccy should contain values in various currencies, e.g. "EUR", "GBP" or "USD". An instance should contain the amount and the currency unit. The class, you are going to design as an exercise, might be best described with the following example session:

from currencies import Ccy

v1 = Ccy(23.43, "EUR")

v2 = Ccy(19.97, "USD")

print(v1 + v2)

In [ ]:

print(v2 + v1)

In [ ]:

print(v1 + 3) *# an int or a float is considered to be a EUR value*

In [ ]:

print(3 + v1)

**Solutions to our Exercises**

1)First exercise:

*"""*

*The class "Ccy" can be used to define money values in various currencies. A Ccy instance has the string attributes 'unit' (e.g. 'CHF', 'CAD' od 'EUR' and the 'value' as a float.*

*A currency object consists of a value and the corresponding unit.*

*"""*

**class** **Ccy**:

currencies = {'CHF': 1.0821202355817312,

'CAD': 1.488609845538393,

'GBP': 0.8916546282920325,

'JPY': 114.38826536281809,

'EUR': 1.0,

'USD': 1.11123458162018}

**def** \_\_init\_\_(self, value, unit="EUR"):

self.value = value

self.unit = unit

**def** \_\_str\_\_(self):

**return** "**{0:5.2f}**".format(self.value) + " " + self.unit

**def** changeTo(self, new\_unit):

*"""*

*An Ccy object is transformed from the unit "self.unit" to "new\_unit"*

*"""*

self.value = (self.value / Ccy.currencies[self.unit] \* Ccy.currencies[new\_unit])

self.unit = new\_unit

**def** \_\_add\_\_(self, other):

*"""*

*Defines the '+' operator.*

*If other is a CCy object the currency values*

*are added and the result will be the unit of*

*self. If other is an int or a float, other will*

*be treated as a Euro value.*

*"""*

**if** type(other) == int **or** type(other) == float:

x = (other \* Ccy.currencies[self.unit])

**else**:

x = (other.value / Ccy.currencies[other.unit] \* Ccy.currencies[self.unit])

**return** Ccy(x + self.value, self.unit)

**def** \_\_iadd\_\_(self, other):

*"""*

*Similar to \_\_add\_\_*

*"""*

**if** type(other) == int **or** type(other) == float:

x = (other \* Ccy.currencies[self.unit])

**else**:

x = (other.value / Ccy.currencies[other.unit] \* Ccy.currencies[self.unit])

self.value += x

**return** self

**def** \_\_radd\_\_(self, other):

res = self + other

**if** self.unit != "EUR":

res.changeTo("EUR")

**return** res

*# \_\_sub\_\_, \_\_isub\_\_ and \_\_rsub\_\_ can be defined analogue*

Overwriting currencies.py

**from** **currencies** **import** Ccy

x = Ccy(10,"USD")

y = Ccy(11)

z = Ccy(12.34, "JPY")

z = 7.8 + x + y + 255 + z

print(z)

lst = [Ccy(10,"USD"), Ccy(11), Ccy(12.34, "JPY"), Ccy(12.34, "CAD")]

z = sum(lst)

print(z)

282.91 EUR

28.40 EUR

Another interesting aspect of this currency converter class in Python can be shown, if we add multiplication. You will easily understand that it makes no sense to allow expressions like "12.4 € \* 3.4 USD" (or in prefix notation: "€ 12.4 *$ 3.4"), but it makes perfect sense to evaluate "3*4.54 €". You can find the new currency converter class with the newly added methods for \_\_mul\_\_, \_\_imul\_\_ and \_\_rmul\_\_ in the following listing:

*"""*

*The class "Ccy" can be used to define money values in various currencies. A Ccy instance has the string attributes 'unit' (e.g. 'CHF', 'CAD' od 'EUR' and the 'value' as a float.*

*A currency object consists of a value and the corresponding unit.*

*"""*

**class** **Ccy**:

currencies = {'CHF': 1.0821202355817312,

'CAD': 1.488609845538393,

'GBP': 0.8916546282920325,

'JPY': 114.38826536281809,

'EUR': 1.0,

'USD': 1.11123458162018}

**def** \_\_init\_\_(self, value, unit="EUR"):

self.value = value

self.unit = unit

**def** \_\_str\_\_(self):

**return** "**{0:5.2f}**".format(self.value) + " " + self.unit

**def** \_\_repr\_\_(self):

**return** 'Ccy(' + str(self.value) + ', "' + self.unit + '")'

**def** changeTo(self, new\_unit):

*"""*

*An Ccy object is transformed from the unit "self.unit" to "new\_unit"*

*"""*

self.value = (self.value / Ccy.currencies[self.unit] \* Ccy.currencies[new\_unit])

self.unit = new\_unit

**def** \_\_add\_\_(self, other):

*"""*

*Defines the '+' operator.*

*If other is a CCy object the currency values*

*are added and the result will be the unit of*

*self. If other is an int or a float, other will*

*be treated as a Euro value.*

*"""*

**if** type(other) == int **or** type(other) == float:

x = (other \* Ccy.currencies[self.unit])

**else**:

x = (other.value / Ccy.currencies[other.unit] \* Ccy.currencies[self.unit])

**return** Ccy(x + self.value, self.unit)

**def** \_\_iadd\_\_(self, other):

*"""*

*Similar to \_\_add\_\_*

*"""*

**if** type(other) == int **or** type(other) == float:

x = (other \* Ccy.currencies[self.unit])

**else**:

x = (other.value / Ccy.currencies[other.unit] \* Ccy.currencies[self.unit])

self.value += x

**return** self

**def** \_\_radd\_\_(self, other):

res = self + other

**if** self.unit != "EUR":

res.changeTo("EUR")

**return** res

*# \_\_sub\_\_, \_\_isub\_\_ and \_\_rsub\_\_ can be defined analogue*

**def** \_\_mul\_\_(self, other):

*"""*

*Multiplication is only defined as a scalar multiplication,*

*i.e. a money value can be multiplied by an int or a float.*

*It is not possible to multiply to money values*

*"""*

**if** type(other)==int **or** type(other)==float:

**return** Ccy(self.value \* other, self.unit)

**else**:

**raise** **TypeError**("unsupported operand type(s) for \*: 'Ccy' and " + type(other).\_\_name\_\_)

**def** \_\_rmul\_\_(self, other):

**return** self.\_\_mul\_\_(other)

**def** \_\_imul\_\_(self, other):

**if** type(other)==int **or** type(other)==float:

self.value \*= other

**return** self

**else**:

**raise** **TypeError**("unsupported operand type(s) for \*: 'Ccy' and " + type(other).\_\_name\_\_)

Overwriting currency\_converter.py

Assuming that you have saved the class under the name currency\_converter, you can use it in the following way in the command shell:

**from** **currency\_converter** **import** Ccy

x = Ccy(10.00, "EUR")

y = Ccy(10.00, "GBP")

x + y

Output: :

Ccy(21.215104685942173, "EUR")

print(x + y)

21.22 EUR

print(2\*x + y\*0.9)

30.09 EUR

**Footnotes**

* as suggested by Mark Jackson

Previous Chapter: [Multiple Inheritance Example](https://www.python-course.eu/python3_multiple_inheritance_example.php)  
Next Chapter: [OOP, Inheritance Example](https://www.python-course.eu/python3_inheritance_example.php)

## Inheritance Example

### Introduction



There aren't many good examples on inheritance available on the web. They are either extremely simple and artificial or they are way too complicated. We want to close the gap by providing an example which is on the one hand more realistic - but still not realistic - and on the other hand simple enough to see and understand the basic aspects of inheritance. In our previous chapter, we introduced inheritance formally.

To this purpose we define two base classes: One is an implementation of a clock and the other one of a calendar. Based on these two classes, we define a class CalendarClock, which inherits both from the class Calendar and from the class Clock.

### The Clock Class

**class** **Clock**(object):

**def** \_\_init\_\_(self,hours=0, minutes=0, seconds=0):

self.\_\_hours = hours

self.\_\_minutes = minutes

self.\_\_seconds = seconds

**def** set(self,hours, minutes, seconds=0):

self.\_\_hours = hours

self.\_\_minutes = minutes

self.\_\_seconds = seconds

**def** tick(self):

*""" Time will be advanced by one second """*

**if** self.\_\_seconds == 59:

self.\_\_seconds = 0

**if** (self.\_\_minutes == 59):

self.\_\_minutes = 0

self.\_\_hours = 0 **if** self.\_\_hours==23 **else** self.\_\_hours + 1

**else**:

self.\_\_minutes += 1;

**else**:

self.\_\_seconds += 1;

**def** display(self):

print("**%d**:**%d**:**%d**" % (self.\_\_hours, self.\_\_minutes, self.\_\_seconds))

**def** \_\_str\_\_(self):

**return** "**%2d**:**%2d**:**%2d**" % (self.\_\_hours, self.\_\_minutes, self.\_\_seconds)

x = Clock()

print(x)

**for** i **in** range(10000):

x.tick()

print(x)

0: 0: 0

2:46:40

### The Calendar Class

**class** **Calendar**(object):

months = (31,28,31,30,31,30,31,31,30,31,30,31)

**def** \_\_init\_\_(self, day=1, month=1, year=1900):

self.\_\_day = day

self.\_\_month = month

self.\_\_year = year

**def** leapyear(self,y):

**if** y % 4:

*# not a leap year*

**return** 0;

**else**:

**if** y % 100:

**return** 1;

**else**:

**if** y % 400:

**return** 0

**else**:

**return** 1;

**def** set(self, day, month, year):

self.\_\_day = day

self.\_\_month = month

self.\_\_year = year

**def** get():

**return** (self, self.\_\_day, self.\_\_month, self.\_\_year)

**def** advance(self):

months = Calendar.months

max\_days = months[self.\_\_month-1]

**if** self.\_\_month == 2:

max\_days += self.leapyear(self.\_\_year)

**if** self.\_\_day == max\_days:

self.\_\_day = 1

**if** (self.\_\_month == 12):

self.\_\_month = 1

self.\_\_year += 1

**else**:

self.\_\_month += 1

**else**:

self.\_\_day += 1

**def** \_\_str\_\_(self):

**return** str(self.\_\_day)+"/"+ str(self.\_\_month)+ "/"+ str(self.\_\_year)

**if** \_\_name\_\_ == "\_\_main\_\_":

x = Calendar()

print(x)

x.advance()

print(x)

1/1/1900

2/1/1900

### The Calendar-Clock Class

**from** **clock1** **import** Clock

**from** **calendar** **import** Calendar

**class** **CalendarClock**(Clock, Calendar):

**def** \_\_init\_\_(self, day,month,year,hours=0, minutes=0,seconds=0):

Calendar.\_\_init\_\_(self, day, month, year)

Clock.\_\_init\_\_(self, hours, minutes, seconds)

**def** \_\_str\_\_(self):

**return** Calendar.\_\_str\_\_(self) + ", " + Clock.\_\_str\_\_(self)

**if** \_\_name\_\_ == "\_\_main\_\_":

x = CalendarClock(24,12,57)

print(x)

**for** i **in** range(1000):

x.tick()

**for** i **in** range(1000):

x.advance()

print(x)

0: 0: 0

2:46:40

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-5-ced1ddf95fd9>** in <module>

13

14 **if** \_\_name\_\_ **==** **"\_\_main\_\_":**

**---> 15** x **=** CalendarClock**(24,12,57)**

16 print**(**x**)**

17 **for** i **in** range**(1000):**

**<ipython-input-5-ced1ddf95fd9>** in \_\_init\_\_**(self, day, month, year, hours, minutes, seconds)**

5

6 **def** \_\_init\_\_**(**self**,** day**,**month**,**year**,**hours**=0,** minutes**=0,**seconds**=0):**

**----> 7** Calendar**.**\_\_init\_\_**(**self**,** day**,** month**,** year**)**

8 Clock**.**\_\_init\_\_**(**self**,** hours**,** minutes**,** seconds**)**

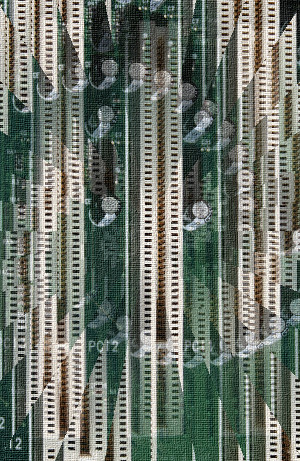
9

**TypeError**: \_\_init\_\_() takes from 1 to 2 positional arguments but 4 were given

Previous Chapter: [Magic Methods and Operator Overloading](https://www.python-course.eu/python3_magic_methods.php)  
Next Chapter: [Slots](https://www.python-course.eu/python3_slots.php)

## Slots

### Avoiding Dynamically Created Attributes



The attributes of objects are stored in a dictionary \_\_dict\_\_. Like any other dictionary, a dictionary used for attribute storage doesn't have a fixed number of elements. In other words, you can add elements to dictionaries after they are defined, as we have seen in our chapter on dictionaries. This is the reason, why you can dynamically add attributes to objects of classes that we have created so far:

**class** **A**(object):

**pass**

a = A()

a.x = 66

a.y = "dynamically created attribute"

The dictionary containing the attributes of "a" can be accessed like this:

a.\_\_dict\_\_

Output: :

{'x': 66, 'y': 'dynamically created attribute'}

You might have wondered that you can dynamically add attributes to the classes, we have defined so far, but that you can't do this with built-in classes like 'int', or 'list':

x = 42

x.a = "not possible to do it"

**---------------------------------------------------------------------------**

**AttributeError** Traceback (most recent call last)

**<ipython-input-3-8c5f7956a976>** in <module>

1 x **=** **42**

**----> 2** x**.**a **=** **"not possible to do it"**

**AttributeError**: 'int' object has no attribute 'a'

lst = [34, 999, 1001]

lst.a = "forget it"

**---------------------------------------------------------------------------**

**AttributeError** Traceback (most recent call last)

**<ipython-input-4-df06616479b6>** in <module>

1 lst **=** **[34,** **999,** **1001]**

**----> 2** lst**.**a **=** **"forget it"**

**AttributeError**: 'list' object has no attribute 'a'

Using a dictionary for attribute storage is very convenient, but it can mean a waste of space for objects, which have only a small amount of instance variables. The space consumption can become critical when creating large numbers of instances. Slots are a nice way to work around this space consumption problem. Instead of having a dynamic dict that allows adding attributes to objects dynamically, slots provide a static structure which prohibits additions after the creation of an instance.

When we design a class, we can use slots to prevent the dynamic creation of attributes. To define slots, you have to define a list with the name \_\_slots\_\_. The list has to contain all the attributes, you want to use. We demonstrate this in the following class, in which the slots list contains only the name for an attribute "val".

**class** **S**(object):

\_\_slots\_\_ = ['val']

**def** \_\_init\_\_(self, v):

self.val = v

x = S(42)

print(x.val)

x.new = "not possible"

42

**---------------------------------------------------------------------------**

**AttributeError** Traceback (most recent call last)

**<ipython-input-5-58aeffbcf9f0>** in <module>

10 print**(**x**.**val**)**

11

**---> 12** x**.**new **=** **"not possible"**

**AttributeError**: 'S' object has no attribute 'new'

If we start this program, we can see, that it is not possible to create dynamically a new attribute. We fail to create an attribute "new".

We mentioned in the beginning that slots are preventing a waste of space with objects. Since Python 3.3 this advantage is not as impressive any more. With Python 3.3 Key-Sharing Dictionaries are used for the storage of objects. The attributes of the instances are capable of sharing part of their internal storage between each other, i.e. the part which stores the keys and their corresponding hashes. This helps reducing the memory consumption of programs, which create many instances of non-builtin types.

Previous Chapter: [OOP, Inheritance Example](https://www.python-course.eu/python3_inheritance_example.php)  
Next Chapter: [Classes and Class Creation](https://www.python-course.eu/python3_classes_and_type.php)

**Classes and Class Creation**

**Behind the scenes: Relationship between Class and type**



In this chapter of our tutorial, we will provide you with a deeper insight into the magic happening behind the scenes, when we are defining a class or creating an instance of a class. You may ask yourself: "Do I really have to learn theses additional details on object oriented programming in Python?" Most probably not, or you belong to the few people who design classes at a very advanced level.

First, we will concentrate on the relationship between type and class. While defining classes so far, you may have asked yourself, what is happening "behind the lines". We have already seen, that applying "type" to an object returns the class of which the object is an instance of:

x = [4, 5, 9]

y = "Hello"

print(type(x), type(y))

<class 'list'> <class 'str'>

If you apply type on the name of a class itself, you get the class "type" returned.

print(type(list), type(str))

<class 'type'> <class 'type'>

This is similar to applying type on type(x) and type(y):

x = [4, 5, 9]

y = "Hello"

print(type(x), type(y))

print(type(type(x)), type(type(y)))

<class 'list'> <class 'str'>

<class 'type'> <class 'type'>

A user-defined class (or the class "object") is an instance of the class "type". So, we can see, that classes are created from type. In Python3 there is no difference between "classes" and "types". They are in most cases used as synonyms.

The fact that classes are instances of a class "type" allows us to program metaclasses. We can create classes, which inherit from the class "type". So, a metaclass is a subclass of the class "type".

Instead of only one argument, type can be called with three parameters:

type(classname, superclasses, attributes\_dict)

If type is called with three arguments, it will return a new type object. This provides us with a dynamic form of the class statement.

* "classname" is a string defining the class name and becomes the name attribute;
* "superclasses" is a list or tuple with the superclasses of our class. This list or tuple will become the bases attribute;
* the attributes\_dict is a dictionary, functioning as the namespace of our class. It contains the definitions for the class body and it becomes the dict attribute.

Let's have a look at a simple class definition:

**class** **A**:

**pass**

x = A()

print(type(x))

<class '\_\_main\_\_.A'>

We can use "type" for the previous class defintion as well:

A = type("A", (), {})

x = A()

print(type(x))

<class '\_\_main\_\_.A'>

Generally speaking, this means, that we can define a class A with

type(classname, superclasses, attributedict)

When we call "type", the call method of type is called. The call method runs two other methods: new and init:

type.\_\_new\_\_(typeclass, classname, superclasses, attributedict)

type.\_\_init\_\_(cls, classname, superclasses, attributedict)

The new method creates and returns the new class object, and after this, the init method initializes the newly created object.

**class** **Robot**:

counter = 0

**def** \_\_init\_\_(self, name):

self.name = name

**def** sayHello(self):

**return** "Hi, I am " + self.name

**def** Rob\_init(self, name):

self.name = name

Robot2 = type("Robot2",

(),

{"counter":0,

"\_\_init\_\_": Rob\_init,

"sayHello": **lambda** self: "Hi, I am " + self.name})

x = Robot2("Marvin")

print(x.name)

print(x.sayHello())

y = Robot("Marvin")

print(y.name)

print(y.sayHello())

print(x.\_\_dict\_\_)

print(y.\_\_dict\_\_)

Marvin

Hi, I am Marvin

Marvin

Hi, I am Marvin

{'name': 'Marvin'}

{'name': 'Marvin'}

The class definitions for Robot and Robot2 are syntactically completely different, but they implement logically the same class.

What Python actually does in the first example, i.e. the "usual way" of defining classes, is the following: Python processes the complete class statement from class Robot to collect the methods and attributes of Robot to add them to the attributes\_dict of the type call. So, Python will call type in a similar or the same way as we did in Robot2.

Previous Chapter: [Slots](https://www.python-course.eu/python3_slots.php)  
Next Chapter: [Road to Metaclasses](https://www.python-course.eu/python3_road_to_metaclasses.php)

## On the Road to Metaclasses

### Motivation for Metaclasses



In this chapter of our tutorial we want to provide some incentives or motivation for the use of metaclasses. To demonstrate some design problems, which can be solved by metaclasses, we will introduce and design a bunch of philosopher classes. Each philosopher class (Philosopher1, Philosopher2, and so on) need the same "set" of methods (in our example just one, i.e. "the\_answer") as the basics for his or her pondering and brooding. A stupid way to implement the classes would be having the same code in every philospher class:

**class** **Philosopher1**(object):

**def** the\_answer(self, \*args):

**return** 42

**class** **Philosopher2**(object):

**def** the\_answer(self, \*args):

**return** 42

**class** **Philosopher3**(object):

**def** the\_answer(self, \*args):

**return** 42

plato = Philosopher1()

print (plato.the\_answer())

kant = Philosopher2()

*# let's see what Kant has to say :-)*

print (kant.the\_answer())

42

42

We can see that we have multiple copies of the method "the\_answer". This is error prone and tedious to maintain, of course.

From what we know so far, the easiest way to accomplish our goal without creating redundant code would be designing a base, which contains "the\_answer" as a method. Now each Philosopher class inherits from this base class:

**class** **Answers**(object):

**def** the\_answer(self, \*args):

**return** 42

**class** **Philosopher1**(Answers):

**pass**

**class** **Philosopher2**(Answers):

**pass**

**class** **Philosopher3**(Answers):

**pass**

plato = Philosopher1()

print plato.the\_answer()

kant = Philosopher2()

*# let's see what Kant has to say :-)*

print kant.the\_answer()

42

42

The way we have designed our classes, each Philosopher class will always have a method "the\_answer". Let's assume, we don't know a priori if we want or need this method. Let's assume that the decision, if the classes have to be augmented, can only be made at runtime. This decision might depend on configuration files, user input or some calculations.

*# the following variable would be set as the result of a runtime calculation:*

x = raw\_input("Do you need 'the answer'? (y/n): ")

**if** x=="y":

required = **True**

**else**:

required = **False**

**def** the\_answer(self, \*args):

**return** 42

**class** **Philosopher1**(object):

**pass**

**if** required:

Philosopher1.the\_answer = the\_answer

**class** **Philosopher2**(object):

**pass**

**if** required:

Philosopher2.the\_answer = the\_answer

**class** **Philosopher3**(object):

**pass**

**if** required:

Philosopher3.the\_answer = the\_answer

plato = Philosopher1()

kant = Philosopher2()

*# let's see what Plato and Kant have to say :-)*

**if** required:

print kant.the\_answer()

print plato.the\_answer()

**else**:

print "The silence of the philosphers"

Do you need 'the answer'? (y/n): y

42

42

Even though this is another solution to our problem, there are still some serious drawbacks. It's error-prone, because we have to add the same code to every class and it seems likely that we might forget it. Besides this it's getting hardly manageable and maybe even confusing, if there are many methods we want to add.

We can improve our approach by defining a manager function and avoiding redundant code this way. The manager function will be used to augment the classes conditionally.

*# the following variable would be set as the result of a runtime calculation:*

x = raw\_input("Do you need 'the answer'? (y/n): ")

**if** x=="y":

required = **True**

**else**:

required = **False**

**def** the\_answer(self, \*args):

**return** 42

*# manager function*

**def** augment\_answer(cls):

**if** required:

cls.the\_answer = the\_answer

**class** **Philosopher1**(object):

**pass**

augment\_answer(Philosopher1)

**class** **Philosopher2**(object):

**pass**

augment\_answer(Philosopher2)

**class** **Philosopher3**(object):

**pass**

augment\_answer(Philosopher3)

plato = Philosopher1()

kant = Philosopher2()

*# let's see what Plato and Kant have to say :-)*

**if** required:

print kant.the\_answer()

print plato.the\_answer()

**else**:

print "The silence of the philosphers"

Do you need 'the answer'? (y/n): y

42

42

This is again useful to solve our problem, but we, i.e. the class designers, must be careful not to forget to call the manager function "augment\_answer". The code should be executed automatically. We need a way to make sure that "some" code might be executed automatically after the end of a class definition.

*# the following variable would be set as the result of a runtime calculation:*

x = raw\_input("Do you need 'the answer'? (y/n): ")

**if** x=="y":

required = **True**

**else**:

required = **False**

**def** the\_answer(self, \*args):

**return** 42

**def** augment\_answer(cls):

**if** required:

cls.the\_answer = the\_answer

*# we have to return the class now:*

**return** cls

@augment\_answer

**class** **Philosopher1**(object):

**pass**

@augment\_answer

**class** **Philosopher2**(object):

**pass**

@augment\_answer

**class** **Philosopher3**(object):

**pass**

plato = Philosopher1()

kant = Philosopher2()

*# let's see what Plato and Kant have to say :-)*

**if** required:

print kant.the\_answer()

print plato.the\_answer()

**else**:

print "The silence of the philosphers"

Do you need 'the answer'? (y/n): y

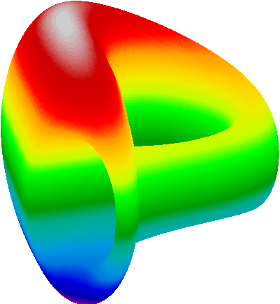
42

42

Metaclasses can also be used for this purpose as we will learn in the next chapter.

Previous Chapter: [Classes and Class Creation](https://www.python-course.eu/python3_classes_and_type.php)  
Next Chapter: [Metaclasses](https://www.python-course.eu/python3_metaclasses.php)

**Metaclasses**



A metaclass is a class whose instances are classes. Like an "ordinary" class defines the behavior of the instances of the class, a metaclass defines the behavior of classes and their instances.

Metaclasses are not supported by every object oriented programming language. Those programming language, which support metaclasses, considerably vary in way they implement them. Python is supporting them.

Some programmers see metaclasses in Python as "solutions waiting or looking for a problem".

There are numerous use cases for metaclasses. Just to name a few:

* logging and profiling
* interface checking
* registering classes at creation time
* automatically adding new methods
* automatic property creation
* proxies
* automatic resource locking/synchronization.

**Defining Metaclasses**

Principially, metaclasses are defined like any other Python class, but they are classes that inherit from "type". Another difference is, that a metaclass is called automatically, when the class statement using a metaclass ends. In other words: If no "metaclass" keyword is passed after the base classes (there may be no base classes either) of the class header, type() (i.e. \_\_call\_\_ of type) will be called. If a metaclass keyword is used, on the other hand, the class assigned to it will be called instead of type.

Now we create a very simple metaclass. It's good for nothing, except that it will print the content of its arguments in the \_\_new\_\_ method and returns the results of the type.\_\_new\_\_ call:

**class** **LittleMeta**(type):

**def** \_\_new\_\_(cls, clsname, superclasses, attributedict):

print("clsname: ", clsname)

print("superclasses: ", superclasses)

print("attributedict: ", attributedict)

**return** type.\_\_new\_\_(cls, clsname, superclasses, attributedict)

We will use the metaclass "LittleMeta" in the following example:

**class** **S**:

**pass**

**class** **A**(S, metaclass=LittleMeta):

**pass**

a = A()

clsname: A

superclasses: (<class '\_\_main\_\_.S'>,)

attributedict: {'\_\_module\_\_': '\_\_main\_\_', '\_\_qualname\_\_': 'A'}

We can see LittleMeta.\_\_new\_\_ has been called and not type.\_\_new\_\_.

Resuming our thread from the last chapter: We define a metaclass "EssentialAnswers" which is capable of automatically including our augment\_answer method:

x = input("Do you need the answer? (y/n): ")

**if** x.lower() == "y":

required = **True**

**else**:

required = **False**

**def** the\_answer(self, \*args):

**return** 42

**class** **EssentialAnswers**(type):

**def** \_\_init\_\_(cls, clsname, superclasses, attributedict):

**if** required:

cls.the\_answer = the\_answer

**class** **Philosopher1**(metaclass=EssentialAnswers):

**pass**

**class** **Philosopher2**(metaclass=EssentialAnswers):

**pass**

**class** **Philosopher3**(metaclass=EssentialAnswers):

**pass**

plato = Philosopher1()

print(plato.the\_answer())

kant = Philosopher2()

*# let's see what Kant has to say :-)*

print(kant.the\_answer())

Do you need the answer? (y/n): y

42

42

We have learned in our chapter "Type and Class Relationship" that after the class definition has been processed, Python calls

type(classname, superclasses, attributes\_dict)

This is not the case, if a metaclass has been declared in the header. That is what we have done in our previous example. Our classes Philosopher1, Philosopher2 and Philosopher3 have been hooked to the metaclass EssentialAnswers. That's why EssentialAnswer will be called instead of type:

EssentialAnswer(classname, superclasses, attributes\_dict)

To be precise, the arguments of the calls will be set the the following values:

EssentialAnswer('Philopsopher1',

(),

{'\_\_module\_\_': '\_\_main\_\_', '\_\_qualname\_\_': 'Philosopher1'})

The other philosopher classes are treated in an analogue way.

**Creating Singletons using Metaclasses**

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. It is used in cases where exactly one object is needed. The concept can be generalized to restrict the instantiation to a certain or fixed number of objects. The term stems from mathematics, where a singleton, - also called a unit set -, is used for sets with exactly one element.

**class** **Singleton**(type):

\_instances = {}

**def** \_\_call\_\_(cls, \*args, \*\*kwargs):

**if** cls **not** **in** cls.\_instances:

cls.\_instances[cls] = super(Singleton, cls).\_\_call\_\_(\*args, \*\*kwargs)

**return** cls.\_instances[cls]

**class** **SingletonClass**(metaclass=Singleton):

**pass**

**class** **RegularClass**():

**pass**

x = SingletonClass()

y = SingletonClass()

print(x == y)

x = RegularClass()

y = RegularClass()

print(x == y)

True

False

**Creating Singletons using Metaclasses**

Alternatively, we can create Singleton classes by inheriting from a Singleton class, which can be defined like this:

**class** **Singleton**(object):

\_instance = **None**

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**if** **not** cls.\_instance:

cls.\_instance = object.\_\_new\_\_(cls, \*args, \*\*kwargs)

**return** cls.\_instance

**class** **SingletonClass**(Singleton):

**pass**

**class** **RegularClass**():

**pass**

x = SingletonClass()

y = SingletonClass()

print(x == y)

x = RegularClass()

y = RegularClass()

print(x == y)

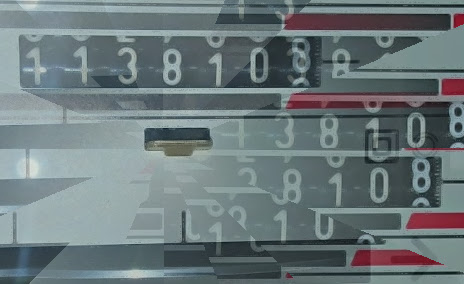
True

False

Previous Chapter: [Road to Metaclasses](https://www.python-course.eu/python3_road_to_metaclasses.php)  
Next Chapter: [Metaclass Use Case: Count Function Calls](https://www.python-course.eu/python3_count_function_calls.php)

## Count Method Calls Using a Metaclass

### Introduction



After you have hopefully gone through our chapter [Introduction into Metaclasses](https://www.python-course.eu/python3_metaclasses.php) you may have asked yourself about the possible use cases for metaclasses. There are some interesting use cases and it's not - like some say - a solution waiting for a problem. We have mentioned already some examples.

In this chapter of our tutorial on Python, we want to elaborate an example metaclass, which will decorate the methods of the subclass. The decorated function returned by the decorator makes it possible to count the number of times each method of the subclass has been called.

This is usually one of the tasks, we expect from a profiler. So we can use this metaclass for simple profiling purposes. Of course, it will be easy to extend our metaclass for further profiling tasks.

### Preliminary Remarks

Before we actually dive into the problem, we want to remind how we can access the attributes of a class. We will demonstrate this with the list class. We can get the list of all the non private attributes of a class - in our example the random class - with the following construct.

**import** **random**

cls = "random" *# name of the class as a string*

all\_attributes = [x **for** x **in** dir(eval(cls)) **if** **not** x.startswith("\_\_") ]

print(all\_attributes)

['BPF', 'LOG4', 'NV\_MAGICCONST', 'RECIP\_BPF', 'Random', 'SG\_MAGICCONST', 'SystemRandom', 'TWOPI', '\_BuiltinMethodType', '\_MethodType', '\_Sequence', '\_Set', '\_acos', '\_ceil', '\_cos', '\_e', '\_exp', '\_inst', '\_log', '\_pi', '\_random', '\_sha512', '\_sin', '\_sqrt', '\_test', '\_test\_generator', '\_urandom', '\_warn', 'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']

Now, we are filtering the callable attributes, i.e. the public methods of the class.

methods = [x **for** x **in** dir(eval(cls)) **if** **not** x.startswith("\_\_")

**and** callable(eval(cls + "." + x))]

print(methods)

['Random', 'SystemRandom', '\_BuiltinMethodType', '\_MethodType', '\_Sequence', '\_Set', '\_acos', '\_ceil', '\_cos', '\_exp', '\_log', '\_sha512', '\_sin', '\_sqrt', '\_test', '\_test\_generator', '\_urandom', '\_warn', 'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']

Getting the non callable attributes of the class can be easily achieved by negating callable, i.e. adding "not":

non\_callable\_attributes = [x **for** x **in** dir(eval(cls)) **if** **not** x.startswith("\_\_")

**and** **not** callable(eval(cls + "." + x))]

print(non\_callable\_attributes)

['BPF', 'LOG4', 'NV\_MAGICCONST', 'RECIP\_BPF', 'SG\_MAGICCONST', 'TWOPI', '\_e', '\_inst', '\_pi', '\_random']

In normal Python programming it is neither recommended nor necessary to apply methods in the following way, but it is possible:

lst = [3,4]

list.\_\_dict\_\_["append"](lst, 42)

lst

Output: :

[3, 4, 42]

Please note the remark from the Python documentation:

"Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class."

### A Decorator for Counting Function Calls

Finally, we will begin designing the metaclass, which we have mentioned as our target in the beginning of this chapter. It will decorate all the methods of its subclass with a decorator, which counts the number of calls. We have defined such a decorator in our chapter [Memoization and Decorators](https://www.python-course.eu/python3_memoization.php):

**def** call\_counter(func):

**def** helper(\*args, \*\*kwargs):

helper.calls += 1

**return** func(\*args, \*\*kwargs)

helper.calls = 0

helper.\_\_name\_\_= func.\_\_name\_\_

**return** helper

We can use it in the usual way:

@call\_counter

**def** f():

**pass**

print(f.calls)

**for** \_ **in** range(10):

f()

print(f.calls)

0

10

It would be better if you add the alternative notation for decorating function. We will need this in our final metaclass:

**def** f():

**pass**

f = call\_counter(f)

print(f.calls)

**for** \_ **in** range(10):

f()

print(f.calls)

0

10

### The "Count Calls" Metaclass

Now we have all the necessary "ingredients" together to write our metaclass. We will include our call\_counter decorator as a staticmethod:

**class** **FuncCallCounter**(type):

*""" A Metaclass which decorates all the methods of the*

*subclass using call\_counter as the decorator*

*"""*

@staticmethod

**def** call\_counter(func):

*""" Decorator for counting the number of function*

*or method calls to the function or method func*

*"""*

**def** helper(\*args, \*\*kwargs):

helper.calls += 1

**return** func(\*args, \*\*kwargs)

helper.calls = 0

helper.\_\_name\_\_= func.\_\_name\_\_

**return** helper

**def** \_\_new\_\_(cls, clsname, superclasses, attributedict):

*""" Every method gets decorated with the decorator call\_counter,*

*which will do the actual call counting*

*"""*

**for** attr **in** attributedict:

**if** callable(attributedict[attr]) **and** **not** attr.startswith("\_\_"):

attributedict[attr] = cls.call\_counter(attributedict[attr])

**return** type.\_\_new\_\_(cls, clsname, superclasses, attributedict)

**class** **A**(metaclass=FuncCallCounter):

**def** foo(self):

**pass**

**def** bar(self):

**pass**

**if** \_\_name\_\_ == "\_\_main\_\_":

x = A()

print(x.foo.calls, x.bar.calls)

x.foo()

print(x.foo.calls, x.bar.calls)

x.foo()

x.bar()

print(x.foo.calls, x.bar.calls)

0 0

1 0

2 1

Previous Chapter: [Metaclasses](https://www.python-course.eu/python3_metaclasses.php)  
Next Chapter: [Abstract Classes](https://www.python-course.eu/python3_abstract_classes.php)

**Abstract Classes**



Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes cannot be instantiated, and require subclasses to provide implementations for the abstract methods.

You can see this in the following examples:

**class** **AbstractClass**:

**def** do\_something(self):

**pass**

**class** **B**(AbstractClass):

**pass**

a = AbstractClass()

b = B()

If we start this program, we see that this is not an abstract class, because:

* we can instantiate an instance from
* we are not required to implement do\_something in the class defintition of B

Our example implemented a case of simple inheritance which has nothing to do with an abstract class. In fact, Python on its own doesn't provide abstract classes. Yet, Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs). This module is called - for obvious reasons - abc.

The following Python code uses the abc module and defines an abstract base class:

**from** **abc** **import** ABC, abstractmethod

**class** **AbstractClassExample**(ABC):

**def** \_\_init\_\_(self, value):

self.value = value

super().\_\_init\_\_()

@abstractmethod

**def** do\_something(self):

**pass**

We will define now a subclass using the previously defined abstract class. You will notice that we haven't implemented the do\_something method, even though we are required to implement it, because this method is decorated as an abstract method with the decorator "abstractmethod". We get an exception that DoAdd42 can't be instantiated:

**class** **DoAdd42**(AbstractClassExample):

**pass**

x = DoAdd42(4)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-4-2bcc42ab0b46> in <module>

**2** pass

**3**

----> 4 x = DoAdd42(4)

TypeError: Can't instantiate abstract class DoAdd42 with abstract methods do\_something

We will do it the correct way in the following example, in which we define two classes inheriting from our abstract class:

**class** **DoAdd42**(AbstractClassExample):

**def** do\_something(self):

**return** self.value + 42

**class** **DoMul42**(AbstractClassExample):

**def** do\_something(self):

**return** self.value \* 42

x = DoAdd42(10)

y = DoMul42(10)

print(x.do\_something())

print(y.do\_something())

52

420

A class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

You may think that abstract methods can't be implemented in the abstract base class. This impression is wrong: An abstract method can have an implementation in the abstract class! Even if they are implemented, designers of subclasses will be forced to override the implementation. Like in other cases of "normal" inheritance, the abstract method can be invoked with super() call mechanism. This enables providing some basic functionality in the abstract method, which can be enriched by the subclass implementation.

**from** **abc** **import** ABC, abstractmethod

**class** **AbstractClassExample**(ABC):

@abstractmethod

**def** do\_something(self):

print("Some implementation!")

**class** **AnotherSubclass**(AbstractClassExample):

**def** do\_something(self):

super().do\_something()

print("The enrichment from AnotherSubclass")

x = AnotherSubclass()

x.do\_something()

Some implementation!

The enrichment from AnotherSubclass

Previous Chapter: [Metaclass Use Case: Count Function Calls](https://www.python-course.eu/python3_count_function_calls.php)