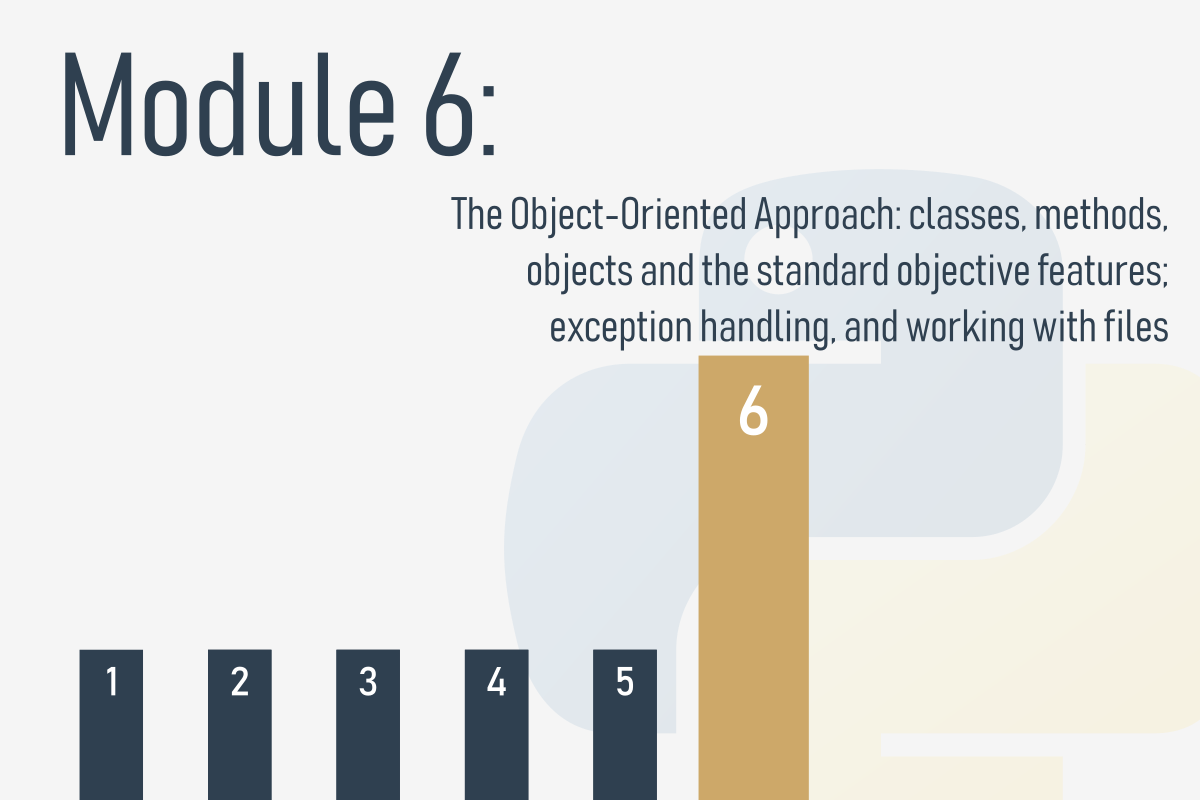
**In this module, you will learn about:**

* the object-oriented approach - foundations;
* classes, methods, objects, and the standard objective features;
* exception handling;
* working with files.



6.1.1.2 The foundations of OOP

# The basic concepts of the object-oriented approach

Let's take a step outside of computer programming and computers in general, and discuss object programming issues.

Nearly all of the programs and techniques you have used till now fall under the procedural style of programming. Admittedly, you have made use of some built-in objects, but when referring to them, we just mentioned the absolute minimum.

The procedural style of programming was the dominant approach to software development for decades of IT, and it is still in use today. Moreover, it isn't going to disappear in the future, as it works very well for specific types of projects (generally, not very complex ones and not large ones, but there are lots of exceptions to that rule).

The object approach is quite young (much younger than the procedural approach) and is particularly useful when applied to big and complex projects carried out by large teams consisting of many developers.

This kind of understanding of a project's structure makes many important tasks easier, e.g., dividing the project into small, independent parts, and independent development of different project elements.

**Python is a universal tool for both object and procedural programming**. It may be successfully utilized in both spheres.

Furthermore, you can create lots of useful applications, even if you know nothing about classes and objects, but you have to keep in mind that some of the problems (e.g., graphical user interface handling) may require a strict object approach.

Fortunately, object programming is relatively simple.

6.1.1.3 The foundations of OOP

# Procedural vs. the object-oriented approach

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

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

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

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

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

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

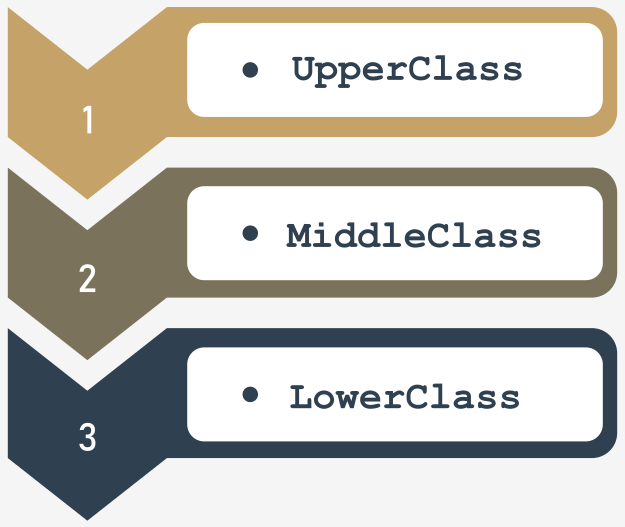
Every object has a set of traits (they are called properties or attributes - we'll use both words synonymously) and is able to perform a set of activities (which are called methods).

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

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

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

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

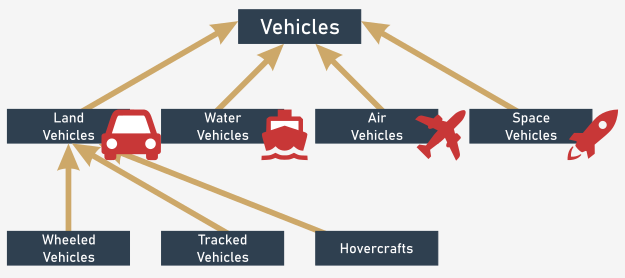
All these concepts are not as abstract as you may at first suspect. On the contrary, they all are taken from real-life experiences, and therefore are extremely useful in computer programming: they don't create artificial life - **they reflect real facts, relationships, and circumstances**.  
  


6.1.1.4 The foundations of OOP

# Class hierarchies

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

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



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

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

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

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

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

* land vehicles;
* water vehicles;
* air vehicles;
* space vehicles.

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

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

* wheeled vehicles;
* tracked vehicles;
* hovercrafts.

The hierarchy we've created is illustrated by the figure.

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

6.1.1.5 The foundations of OOP

# Class hierarchies: continued

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

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

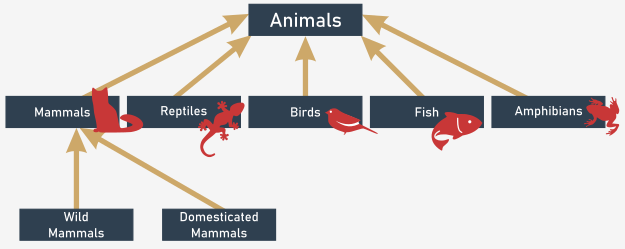
* mammals;
* reptiles;
* birds;
* fish;
* amphibians.

We'll take the first one for further analysis.

We have identified the following subclasses:

* wild mammals;
* domesticated mammals.

Try to extend the hierarchy any way you want, and find the right place for humans.



6.1.1.6 The foundations of OOP

# What is an object?

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

An object is **an incarnation of the requirements, traits, and qualities assigned to a specific class**. This may sound simple, but note the following important circumstances. Classes form a hierarchy. This may mean that an object belonging to a specific class belongs to all the superclasses at the same time. It may also mean that any object belonging to a superclass may not belong to any of its subclasses.

For example: any personal car is an object belonging to the *wheeled vehicles* class. It also means that the same car belongs to all superclasses of its home class; therefore, it is a member of the *vehicles* class, too. Your dog (or your cat) is an object included in the *domesticated mammals* class, which explicitly means that it is included in the *animals* class as well.

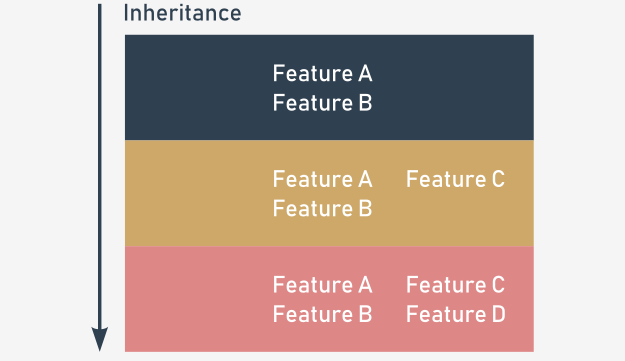
Each **subclass is more specialized** (or more specific) than its superclass. Conversely, each **superclass is more general** (more abstract) than any of its subclasses. Note that we've presumed that a class may only have one superclass - this is not always true, but we'll discuss this issue more a bit later.

## Inheritance

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

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

You shouldn't have any problems matching this rule to specific examples, whether it applies to animals, or to vehicles.



6.1.1.7 The foundations of OOP

# What does an object have?

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

* an object has a **name** that uniquely identifies it within its home namespace (although there may be some anonymous objects, too)
* an object has a **set of individual properties** which make it original, unique or outstanding (although it's possible that some objects may have no properties at all)
* an object has a **set of abilities to perform specific activities**, able to change the object itself, or some of the other objects.

There is a hint (although this doesn't always work) which can help you identify any of the three spheres above. Whenever you describe an object and you use:

* a noun - you probably define the object's name;
* an adjective - you probably define the object's property;
* a verb - you probably define the object's activity.

Two sample phrases should serve as a good example:

* Max is a large cat who sleeps all day.  
    
  Object name = Max  
  Home class = Cat  
  Property = Size (large)  
  Activity = Sleep (all day)
* A pink Cadillac went quickly.  
    
  Object name = Cadillac  
  Home class = Wheeled vehicles  
  Property = Color (pink)  
  Activity = Go (quickly)

6.1.1.8 The foundations of OOP

# Your first class

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

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

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

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

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

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

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

class TheSimplestClass:

pass

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

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

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

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

## Your first object

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

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

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

You do it in the following way:

myFirstObject = TheSimplestClass()

Note:

* the class name tries to pretend that it's a function - can you see this? We'll discuss it soon;
* the newly created object is equipped with everything the class brings; as this class is completely empty, the object is empty, too.

The act of creating an object of the selected class is also called an **instantiation** (as the object becomes an **instance of the class**).

Let's leave classes alone for a short moment, as we're now going to tell you a few words about *stacks*. We know the concept of classes and objects may not be fully clear yet. Don't worry, we'll explain everything very soon.

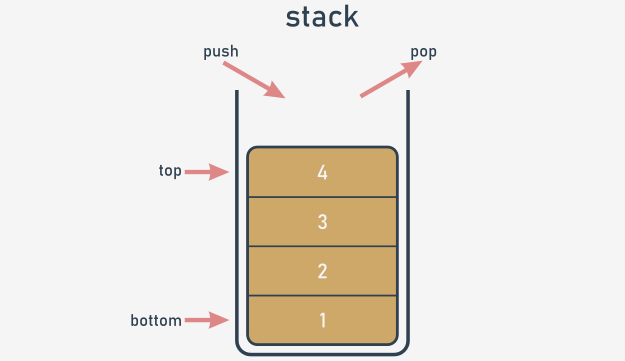
6.1.2.1 A short journey from procedural to object approach

**What is a stack?**

**A stack is a structure developed to store data in a very specific way**. Imagine a stack of coins. You aren't able to put a coin anywhere else but on the top of the stack. Similarly, you can't get a coin off the stack from any place other than the top of the stack. If you want to get the coin that lies on the bottom, you have to remove all the coins from the higher levels.

The alternative name for a stack (but only in IT terminology) is **LIFO**. It's an abbreviation for a very clear description of the stack's behavior: **Last In - First Out**. The coin that came last onto the stack will leave first.

**A stack is an object** with two elementary operations, conventionally named **push** (when a new element is put on the top) and **pop** (when an existing element is taken away from the top).

Stacks are used very often in many classical algorithms, and it's hard to imagine the implementation of many widely used tools without the use of stacks.  
  
Let's implement a stack in Python. This will be a very simple stack, and we'll show you how to do it in two independent approaches: procedural and objective.

Let's start with the first one.

Let's implement a stack in Python. This will be a very simple stack, and we'll show you how to do it in two independent approaches: procedural and objective.

Let's start with the first one.

6.1.2.2 A short journey from procedural to object approach

# The stack - the procedural approach

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

The stack itself is already created:

stack = []

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

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

This is how we've done it - take a look:

def push(val):

stack.append(val)

Now it's time for a **function to take a value off the stack**. This is how you can do it:

* the name of the function is pop;
* the function doesn't get any parameters;
* the function returns the value taken from the stack
* the function reads the value from the top of the stack and removes it.

The function is here:

def pop():

val = stack[-1]

del stack[-1]

return val

Note: the function doesn't check if there is any element in the stack.

Let's assemble all the pieces together to set the stack in motion. The **complete program** pushes three numbers onto the stack, pulls them off, and prints their values on the screen. You can see it in the editor window.

The program outputs the following text to the screen:

1

2

3

Test it.

stack = []

def push(val):

stack.append(val)

def pop():

val = stack[-1]

del stack[-1]

return val

push(3)

push(2)

push(1)

print(pop())

print(pop())

print(pop())

6.1.2.3 A short journey from procedural to object approach

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

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

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

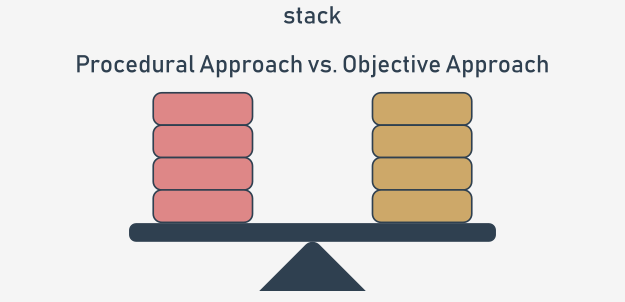
* the essential variable (the stack list) is highly **vulnerable**; anyone can modify it in an uncontrollable way, destroying the stack, in effect; this doesn't mean that it's been done maliciously - on the contrary, it may happen as a result of carelessness, e.g., when somebody confuses variable names; imagine that you have accidentally written something like this:  
    
  stack[0] = 0  
  The functioning of the stack will be completely disorganized;

* it may also happen that one day you need more than one stack; you'll have to create another list for the stack's storage, and probably other push and pop functions too;

* it may also happen that you need not only push and pop functions, but also some other conveniences; you could certainly implement them, but try to imagine what would happen if you had dozens of separately implemented stacks.

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

* the ability to hide (protect) selected values against unauthorized access is called **encapsulation; the encapsulated values can be neither accessed nor modified if you want to use them exclusively**;
* when you have a class implementing all the needed stack behaviors, you can produce as many stacks as you want; you needn't copy or replicate any part of the code;
* the ability to enrich the stack with new functions comes from inheritance; you can create a new class (a subclass) which inherits all the existing traits from the superclass, and adds some new ones.

  
Let's now write a brand new stack implementation from scratch. This time, we'll use the objective approach, guiding you step by step into the world of object programming.

6.1.2.4 A short journey from procedural to object approach

# The stack - the object approach

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

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

class Stack:

Now, we expect two things from it:

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

How is this done?

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

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

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

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

* it has to be named in a strict way;
* it is invoked implicitly, when the new object is created.

Such a function is called a **constructor**, as its general purpose is to **construct a new object**. The constructor should know everything about the object's structure, and must perform all the needed initializations.

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

class Stack:

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

print("Hi!")

stackObject = Stack()

And now:

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

The code is in the editor. Run it now.

Here is its output:

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

class Stack: # defining the Stack class

def \_\_init\_\_(self): # defining the constructor function

print("Hi!")

stackObject = Stack() # instantiating the object

6.1.2.5 A short journey from procedural to object approach

# The stack - the object approach: continued

Any change you make inside the constructor that modifies the state of the self parameter will reflect the newly created object.

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

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

Just like here:

class Stack:

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

self.stackList = []

stackObject = Stack()

print(len(stackObject.stackList))

Note:

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

Yes, we have - the code produces the following output:

0

This is not we want from the stack. We prefer stackList to be **hidden from the outside world**. Is that possible?

Yes, and it's simple, but not very intuitive.

class Stack:

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

self.stackList = []

stackObject = Stack()

print(len(stackObject.stackList))

6.1.2.6 A short journey from procedural to object approach

**The stack - the object approach: continued**

Take a look - we've added two underscores before the stackList name - nothing more:

class Stack:

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

self.\_\_stackList = []

stackObject = Stack()

print(len(stackObject.\_\_stackList))

The change invalidates the program.

Why?

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

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

Run the program to test our assumptions - an AttributeError exception should be raised.

class Stack:

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

self.\_\_stackList = []

stackObject = Stack()

print(len(stackObject.\_\_stackList))

6.1.2.7 A short journey from procedural to object approach

# The object approach: a stack from scratch

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

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

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

The functions themselves are simple. Take a look:

class Stack:

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

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

stackObject = Stack()

stackObject.push(3)

stackObject.push(2)

stackObject.push(1)

print(stackObject.pop())

print(stackObject.pop())

print(stackObject.pop())

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

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

Is it needed? Yes, it is.

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

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

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

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

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

Fortunately, it's much simpler than it looks:

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

The class declaration is complete, and all its components have been listed. The class is ready for use.

class Stack:

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

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

stackObject = Stack()

stackObject.push(3)

stackObject.push(2)

stackObject.push(1)

print(stackObject.pop())

print(stackObject.pop())

print(stackObject.pop())

6.1.2.8 A short journey from procedural to object approach

**The object approach: a stack from scratch**

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

This is exactly what we want for this example.

Analyze the code:

class Stack:

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

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

stackObject1 = Stack()

stackObject2 = Stack()

stackObject1.push(3)

stackObject2.push(stackObject1.pop())

print(stackObject2.pop())

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

Run the code in the editor and see what happens. Carry out your own experiments.

class Stack:

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

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

stackObject1 = Stack()

stackObject2 = Stack()

stackObject1.push(3)

stackObject2.push(stackObject1.pop())

print(stackObject2.pop())

6.1.2.9 A short journey from procedural to object approach

# The object approach: a stack from scratch (continued)

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

class Stack:

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

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

littleStack = Stack()

anotherStack = Stack()

funnyStack = Stack()

littleStack.push(1)

anotherStack.push(littleStack.pop() + 1)

funnyStack.push(anotherStack.pop() - 2)

print(funnyStack.pop())

So, what's the result? Run the program and check if you were right.

class Stack:

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

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

# enter code here

6.1.2.10 A short journey from procedural to object approach

# The object approach: a stack from scratch (continued)

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

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

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

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

This is what it looks like:

class AddingStack(Stack):

pass

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

This is what we want from the new stack:

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

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

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

class AddingStack(Stack):

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

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

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

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

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

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

Note the syntax:

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

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

class Stack:

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

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

class AddingStack(Stack):

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

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

6.1.2.11 A short journey from procedural to object approach

# The object approach: a stack from scratch (continued)

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

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

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

* to add the value to the \_\_sum variable;
* to push the value onto the stack.

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

This is how the push method looks in the subclass:

def push(self, val):

self.\_\_sum += val

Stack.push(self, val)

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

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

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

class Stack:

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

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

class AddingStack(Stack):

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

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

# enter code here

6.1.2.12 A short journey from procedural to object approach

**The object approach: a stack from scratch (continued)**

This is the new pop function:

def pop(self):

val = Stack.pop(self)

self.\_\_sum -= val

return val

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

We have to define a new method. We'll name it getSum. Its only task will be to **return the**\_\_sum**value**.

Here it is:

def getSum(self):

return self.\_\_sum

So, let's look at the program in the editor. The complete code of the class is there. We can check its functioning now, and we do it with the help of a very few additional lines of code.

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

Okay, this has been a very brief introduction to Python's object programming. Soon we're going to tell you about it all in more detail.

class Stack:

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

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

class AddingStack(Stack):

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

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

def getSum(self):

return self.\_\_sum

def push(self, val):

self.\_\_sum += val

Stack.push(self, val)

def pop(self):

val = Stack.pop(self)

self.\_\_sum -= val

return val

stackObject = AddingStack()

for i in range(5):

stackObject.push(i)

print(stackObject.getSum())

for i in range(5):

print(stackObject.pop())

6.1.3.1 OOP: Properties

# Instance variables

In general, a class can be equipped with two different kinds of data to form a class's properties. You already saw one of them when we were looking at stacks.

This kind of class property exists when and only when it is explicitly created and added to an object. As you already know, this can be done during the object's initialization, performed by the constructor.

Moreover, it can be done in any moment of the object's life. Furthermore, any existing property can be removed at any time.

Such an approach has some important consequences:

* different objects of the same class **may possess different sets of properties**;
* there must be a way to **safely check if a specific object owns the property** you want to utilize (unless you want to provoke an exception - it's always worth considering)
* each object **carries its own set of properties** - they don't interfere with one another in any way.

Such variables (properties) are called **instance variables**.

The word *instance* suggests that they are closely connected to the objects (which are class instances), not to the classes themselves. Let's take a closer look at them.

Here is an example:

class ExampleClass:

def \_\_init\_\_(self, val = 1):

self.first = val

def setSecond(self, val):

self.second = val

exampleObject1 = ExampleClass()

exampleObject2 = ExampleClass(2)

exampleObject2.setSecond(3)

exampleObject3 = ExampleClass(4)

exampleObject3.third = 5

print(exampleObject1.\_\_dict\_\_)

print(exampleObject2.\_\_dict\_\_)

print(exampleObject3.\_\_dict\_\_)

It needs one additional explanation before we go into any more detail. Take a look at the last three lines of the code.

Python objects, when created, are gifted with a **small set of predefined properties and methods**. Each object has got them, whether you want them or not. One of them is a variable named \_\_dict\_\_ (it's a dictionary).

The variable contains the names and values of all the properties (variables) the object is currently carrying. Let's make use of it to safely present an object's contents.

Let's dive into the code now:

* the class named ExampleClass has a constructor, which **unconditionally creates an instance variable** named first, and sets it with the value passed through the first argument (from the class user's perspective) or the second argument (from the constructor's perspective); note the default value of the parameter - any trick you can do with a regular function parameter can be applied to methods, too;
* the class also has a **method which creates another instance variable**, named second;
* we've created three objects of the class ExampleClass, but all these instances differ:  
  + exampleObject1 only has the property named first;
  + exampleObject2 has two properties: first and second;
  + exampleObject3 has been enriched with a property named third just on the fly, outside the class's code - this is possible and fully permissible.

The program's output clearly shows that our assumptions are correct - here it is:

{'first': 1}

{'second': 3, 'first': 2}

{'third': 5, 'first': 4}

There is one additional conclusion that should be stated here: **modifying an instance variable of any object has no impact on all the remaining objects**. Instance variables are perfectly isolated from each other.

6.1.3.2 OOP: Properties

# Instance variables: continued

Take a look at the modified example in the editor.

It's nearly the same as the previous one. The only difference is in the property names. We've **added two underscores (**\_\_**)** in front of them.

As you know, such an addition makes the instance variable **private** - it becomes inaccessible from the outer world.

The actual behavior of these names is a bit more complicated, so let's run the program. This is the output:

{'\_ExampleClass\_\_first': 1}

{'\_ExampleClass\_\_first': 2, '\_ExampleClass\_\_second': 3}

{'\_ExampleClass\_\_first': 4, '\_\_third': 5}

Can you see these strange names full of underscores? Where did they come from?

When Python sees that you want to add an instance variable to an object and you're going to do it inside any of the object's methods, it **mangles the operation** in the following way:

* it puts a class name before your name;
* it puts an additional underscore at the beginning.

This is why the \_\_first becomes \_ExampleClass\_\_first.

**The name is now fully accessible from outside the class**. You can run a code like this:

print(exampleObject1.\_ExampleClass\_\_first)

and you'll get a valid result with no errors or exceptions.

As you can see, making a property private is limited.

**The mangling won't work if you add an instance variable outside the class code**. In this case, it'll behave like any other ordinary property.

class ExampleClass:

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

def setSecond(self, val = 2):

self.\_\_second = val

exampleObject1 = ExampleClass()

exampleObject2 = ExampleClass(2)

exampleObject2.setSecond(3)

exampleObject3 = ExampleClass(4)

exampleObject3.\_\_third = 5

print(exampleObject1.\_\_dict\_\_)

print(exampleObject2.\_\_dict\_\_)

print(exampleObject3.\_\_dict\_\_)

6.1.3.3 OOP: Properties

# Class variables

A class variable is **a property which exists in just one copy and is stored outside any object**.

Note: no instance variable exists if there is no object in the class; a class variable exists in one copy even if there are no objects in the class.

Class variables are created differently to their instance siblings. The example will tell you more:

class ExampleClass:

counter = 0

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

ExampleClass.counter += 1

exampleObject1 = ExampleClass()

exampleObject2 = ExampleClass(2)

exampleObject3 = ExampleClass(4)

print(exampleObject1.\_\_dict\_\_, exampleObject1.counter)

print(exampleObject2.\_\_dict\_\_, exampleObject2.counter)

print(exampleObject3.\_\_dict\_\_, exampleObject3.counter)

Look:

* there is an assignment in the first list of the class definition - it sets the variable named counter to 0; initializing the variable inside the class but outside any of its methods makes the variable a class variable;
* accessing such a variable looks the same as accessing any instance attribute - you can see it in the constructor body; as you can see, the constructor increments the variable by one; in effect, the variable counts all the created objects.

Running the code will cause the following output:

{'\_ExampleClass\_\_first': 1} 3

{'\_ExampleClass\_\_first': 2} 3

{'\_ExampleClass\_\_first': 4} 3

Two important conclusions come from the example:

* class variables **aren't shown in an object's**\_\_dict\_\_ (this is natural as class variables aren't parts of an object) but you can always try to look into the variable of the same name, but at the class level - we'll show you this very soon;
* a class variable **always presents the same value** in all class instances (objects)

6.1.3.4 OOP: Properties

# Class variables: continued

Mangling a class variable's name has the same effects as those you're already familiar with.

Look at the example in the editor. Can you guess its output?

Run the program and check if your predictions were correct. Everything works as expected, doesn't it?

class ExampleClass:

\_\_counter = 0

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

ExampleClass.\_\_counter += 1

exampleObject1 = ExampleClass()

exampleObject2 = ExampleClass(2)

exampleObject3 = ExampleClass(4)

print(exampleObject1.\_\_dict\_\_, exampleObject1.\_ExampleClass\_\_counter)

print(exampleObject2.\_\_dict\_\_, exampleObject2.\_ExampleClass\_\_counter)

print(exampleObject3.\_\_dict\_\_, exampleObject3.\_ExampleClass\_\_counter)

6.1.3.5 OOP: Properties

# Class variables: continued

We told you before that class variables exist even when no class instance (object) had been created.

Now we're going to take the opportunity to show you **the difference between these two**\_\_dict\_\_**variables**, the one from the class and the one from the object.

Look at the code in the editor. The proof is there.

Let's take a closer look at it:

* we define one class named ExampleClass;
* the class defines one class variable named varia;
* the class constructor sets the variable with the parameter's value;
* naming the variable is the most important aspect of the example because:
  + changing the assignment to self.varia = val would create an instance variable of the same name as the class's one;
  + changing the assignment to varia = val would operate on a method's local variable; (we strongly encourage you to test both of the above cases - this will make it easier for you to remember the difference)
* the first line of the off-class code prints the value of the ExampleClass.varia attribute; note - we use the value before the very first object of the class is instantiated.

Run the code in the editor and check its output.

As you can see, the class' \_\_dict\_\_ contains much more data than its object's counterpart. Most of them are useless now - the one we want you to check carefully shows the current varia value.

Note that the object's \_\_dict\_\_ is empty - the object has no instance variables.

class ExampleClass:

varia = 1

def \_\_init\_\_(self, val):

ExampleClass.varia = val

print(ExampleClass.\_\_dict\_\_)

exampleObject = ExampleClass(2)

print(ExampleClass.\_\_dict\_\_)

print(exampleObject.\_\_dict\_\_)

6.1.3.6 OOP: Properties

**Checking an attribute's existence**

Python's attitude to object instantiation raises one important issue - in contrast to other programming languages, **you may not expect that all objects of the same class have the same sets of properties**.

Just like in the example in the editor. Look at it carefully.

The object created by the constructor can have only one of two possible attributes: a or b.

Executing the code will produce the following output:

1

Traceback (most recent call last):

File ".main.py", line 11, in

print(exampleObject.b)

AttributeError: 'ExampleClass' object has no attribute 'b'  
As you can see, accessing a non-existing object (class) attribute causes an AttributeError exception.

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

exampleObject = ExampleClass(1)

print(exampleObject.a)

print(exampleObject.b)

6.1.3.7 OOP: Properties

# Checking an attribute's existence: continued

The try-except instruction gives you the chance to avoid issues with non-existent properties.

It's easy - look at the code in the editor.

As you can see, this action isn't very sophisticated. Essentially, we've just swept the issue under the carpet.

Fortunately, there is one more way to cope with the issue.

Python provides a **function which is able to safely check if any object/class contains a specified property**. The function is named hasattr, and expects two arguments to be passed to it:

* the class or the object being checked;
* the name of the property whose existence has to be reported (note: it has to be a string containing the attribute name, not the name alone)

The function returns True or False.

This is how you can utilize it:

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

exampleObject = ExampleClass(1)

print(exampleObject.a)

if hasattr(exampleObject, 'b'):

print(exampleObject.b)

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

exampleObject = ExampleClass(1)

print(exampleObject.a)

try:

print(exampleObject.b)

except AttributeError:

pass

6.1.3.8 OOP: Properties

**Checking an attribute's existence: continued**

Don't forget that the hasattr() function can operate on classes, too. You can use it **to find out if a class variable is available**, just like here in the example in the editor.

The function returns True if the specified class contains a given attribute, and False otherwise.

Can you guess the code's output? Run it to check your guesses.

And one more example - look at the code below and try to predict its output:

class ExampleClass:

a = 1

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

self.b = 2

exampleObject = ExampleClass()

print(hasattr(exampleObject, 'b'))

print(hasattr(exampleObject, 'a'))

print(hasattr(ExampleClass, 'b'))

print(hasattr(ExampleClass, 'a'))

Were you successful? Run the code to check your predictions.

Okay, we've made it to the end of this section. In the next section we're going to talk about methods, as methods drive the objects and make them active.

class ExampleClass:

attr = 1

print(hasattr(ExampleClass, 'attr'))

print(hasattr(ExampleClass, 'prop'))

6.1.4.1 OOP: Methods

# Methods in detail

Let's summarize all the facts regarding the use of methods in Python classes.

As you already know, a **method is a function embedded inside a class**.

There is one fundamental requirement - a **method is obliged to have at least one parameter** (there are no such thing as parameterless methods - a method may be invoked without an argument, but not declared without parameters).

The first (or only) parameter is usually named self. We suggest that you follow the convention - it's commonly used, and you'll cause a few surprises by using other names for it.

The name self suggests the parameter's purpose - **it identifies the object for which the method is invoked**.

If you're going to invoke a method, you mustn't pass the argument for the self parameter - Python will set it for you.

The example in the editor shows the difference.

The code outputs:

method

Note the way we've created the object - we've **treated the class name like a function**, returning a newly instantiated object of the class.

If you want the method to accept parameters other than self, you should:

* place them after self in the method's definition;
* deliver them during invocation without specifying self (as previously)

Just like here:

class Classy:

def method(self, par):

print("method:", par)

obj = Classy()

obj.method(1)

obj.method(2)

obj.method(3)

The code outputs:

method: 1

method: 2

method: 3

class Classy:

def method(self):

print("method")

obj = Classy()

obj.method()

6.1.4.2 OOP: Methods

**Methods in detail: continued**

The self parameter is used **to obtain access to the object's instance and class variables**.

The example shows both ways of utilizing self:

class Classy:

varia = 2

def method(self):

print(self.varia, self.var)

obj = Classy()

obj.var = 3

obj.method()

The code outputs:

2 3

# test examples here

6.1.4.3 OOP: Methods

# Methods in detail: continued

If you name a method like this: \_\_init\_\_, it won't be a regular method - it will be a **constructor**.

If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated.

The constructor:

* is **obliged to have the**self**parameter** (it's set automatically, as usual);
* **may (but doesn't need to) have more parameters** than just self; if this happens, the way in which the class name is used to create the object must reflect the \_\_init\_\_ definition;
* **can be used to set up the object**, i.e., properly initialize its internal state, create instance variables, instantiate any other objects if their existence is needed, etc.

Look at the code in the editor. The example shows a very simple constructor at work.

Run it. The code outputs:

object

Note that the constructor:

* **cannot return a value**, as it is designed to return a newly created object and nothing else;
* **cannot be invoked directly either from the object or from inside the class** (you can invoke a constructor from any of the object's superclasses, but we'll discuss this issue later.)

class Classy:

def \_\_init\_\_(self, value):

self.var = value

obj1 = Classy("object")

print(obj1.var)

6.1.4.4 OOP: Methods

**Methods in detail: continued**

As \_\_init\_\_ is a method, and a method is a function, you can do the same tricks with constructors/methods as you do with ordinary functions.

The example in the editor shows how to define a constructor with a default argument value. Test it.

The code outputs:

object

None

Everything we've said about **property name mangling** applies to method names, too - a method whose name starts with \_\_ is (partially) hidden.

The example shows this effect:

class Classy:

def visible(self):

print("visible")

def \_\_hidden(self):

print("hidden")

obj = Classy()

obj.visible()

try:

obj.\_\_hidden()

except:

print("failed")

obj.\_Classy\_\_hidden()

The code outputs:

visible

failed

hidden

Run the program, and test it.

class Classy:

def \_\_init\_\_(self, value = None):

self.var = value

obj1 = Classy("object")

obj2 = Classy()

print(obj1.var)

print(obj2.var)

6.1.4.5 OOP: Methods

# The inner life of classes and objects

Each Python class and each Python object is pre-equipped with a set of useful attributes which can be used to examine its capabilities.

You already know one of these - it's the \_\_dict\_\_ property.

Let's observe how it deals with methods - look at the code in the editor.

Run it to see what it outputs. Check the output carefully.

Find all the defined methods and attributes. Locate the context in which they exist: inside the object or inside the class.

class Classy:

varia = 1

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

self.var = 2

def method(self):

pass

def \_\_hidden(self):

pass

obj = Classy()

print(obj.\_\_dict\_\_)

print(Classy.\_\_dict\_\_)

6.1.4.6 OOP: Methods

**The inner life of classes and objects: continued**

\_\_dict\_\_ is a dictionary. Another built-in property worth mentioning is \_\_name\_\_, which is a string.

The property contains **the name of the class**. It's nothing exciting, just a string.

Note: the \_\_name\_\_ attribute is absent from the object - **it exists only inside classes**.

If you want to **find the class of a particular object**, you can use a function named type(), which is able (among other things) to find a class which has been used to instantiate any object.

Look at the code in the editor, run it, and see for yourself.

The code outputs:

Classy

Classy

Note: a statement like this one:

print(obj.\_\_name\_\_) will cause an error.

class Classy:

pass

print(Classy.\_\_name\_\_)

obj = Classy()

print(type(obj).\_\_name\_\_)

6.1.4.7 OOP: Methods

**The inner life of classes and objects: continued**

\_\_module\_\_ is a string, too - it **stores the name of the module which contains the definition of the class**.

Let's check it - run the code in the editor.

The code outputs:

\_\_main\_\_

\_\_main\_\_

As you know, any module named \_\_main\_\_ is actually not a module, but the **file currently being run**.

class Classy:

pass

print(Classy.\_\_module\_\_)

obj = Classy()

print(obj.\_\_module\_\_)

6.1.4.8 OOP: Methods

**The inner life of classes and objects: continued**

\_\_bases\_\_ is a tuple. The **tuple contains classes** (not class names) which are direct superclasses for the class.

The order is the same as that used inside the class definition.

We'll show you only a very basic example, as we want to highlight **how inheritance works**.

Moreover, we're going to show you how to use this attribute when we discuss the objective aspects of exceptions.

Note: **only classes have this attribute** - objects don't.

We've defined a function named printbases(), designed to present the tuple's contents clearly.

Look at the code in the editor. Analyze it and run it. It will output:

( object )

( object )

( SuperOne SuperTwo )

Note: **a class without explicit superclasses points to object** (a predefined Python class) as its direct ancestor.

class SuperOne:

pass

class SuperTwo:

pass

class Sub(SuperOne, SuperTwo):

pass

def printBases(cls):

print('( ', end='')

for x in cls.\_\_bases\_\_:

print(x.\_\_name\_\_, end=' ')

print(')')

printBases(SuperOne)

printBases(SuperTwo)

printBases(Sub)

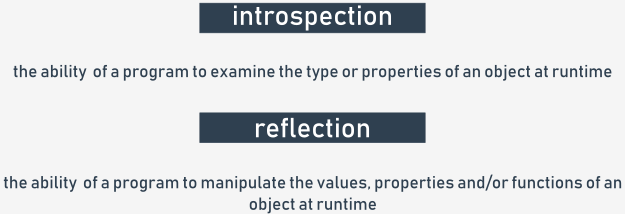
6.1.4.9 OOP: Methods

# Reflection and introspection

All these means allow the Python programmer to perform two important activities specific to many objective languages. They are:

* **introspection**, which is the ability of a program to examine the type or properties of an object at runtime;
* **reflection**, which goes a step further, and is the ability of a program to manipulate the values, properties and/or functions of an object at runtime.

In other words, you don't have to know a complete class/object definition to manipulate the object, as the object and/or its class contain the metadata allowing you to recognize its features during program execution.



# Investigating classes

What can you find out about classes in Python? The answer is simple - everything.

Both reflection and introspection enable a programmer to do anything with every object, no matter where it comes from.

Analyze the the code in the editor.

The function named incIntsI() gets an object of any class, scans its contents in order to find all integer attributes with names starting with i, and increments them by one.

Impossible? Not at all!

This is how it works:

* line 1: define a very simple class...
* lines 3 through 10: ... and fill it with some attributes;
* line 12: this is our function!
* line 13: scan the \_\_dict\_\_ attribute, looking for all attribute names;
* line 14: if a name starts with i...
* line 15: ... use the getattr() function to get its current value; note: getattr() takes two arguments: an object, and its property name (as a string), and returns the current attribute's value;
* line 16: check if the value is of type integer, and use the function isinstance() for this purpose (we'll discuss this later);
* line 17: if the check goes well, increment the property's value by making use of the setattr() function; the function takes three arguments: an object, the property name (as a string), and the property's new value.

The code outputs:

{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}

{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}

That's all!

class MyClass:

pass

obj = MyClass()

obj.a = 1

obj.b = 2

obj.i = 3

obj.ireal = 3.5

obj.integer = 4

obj.z = 5

def incIntsI(obj):

for name in obj.\_\_dict\_\_.keys():

if name.startswith('i'):

val = getattr(obj, name)

if isinstance(val, int):

setattr(obj, name, val + 1)

print(obj.\_\_dict\_\_)

incIntsI(obj)

print(obj.\_\_dict\_\_)

6.1.5.1 OOP Fundamentals: Inheritance

**Inheritance - why and how?**

Before we start talking about inheritance, we want to present a new, handy mechanism utilized by Python's classes and objects - it's **the way in which the object is able to introduce itself**.

Let's start with an example. Look at the code in the editor.

The program prints out just one line of text, which in our case is this:

<\_\_main\_\_.Star object at 0x7f1074cc7c50>

If you run the same code on your computer, you'll see something very similar, although the hexadecimal number (the substring starting with 0x) will be different, as it's just an internal object identifier used by Python, and it's unlikely that it would appear the same when the same code is run in a different environment.

As you can see, the printout here isn't really useful, and something more specific, or just prettier, may be more preferable.

Fortunately, Python offers just such a function.

class Star:

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

self.name = name

self.galaxy = galaxy

sun = Star("Sun", "Milky Way")

print(sun)

6.1.5.2 OOP Fundamentals: Inheritance

# Inheritance - why and how?

When Python needs any class/object to be presented as a string (putting an object as an argument in the print() function invocation fits this condition) it tries to invoke a method named \_\_str\_\_() from the object and to use the string it returns.

The default \_\_str\_\_() method returns the previous string - ugly and not very informative. You can change it just by **defining your own method of the name**.

We've just done it - look at the code in the editor.

This new \_\_str\_\_() method makes a string consisting of the star's and galaxy's names - nothing special, but the print results look better now, doesn't it?

Can you guess the output? Run the code to check if you were right.

class Star:

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

self.name = name

self.galaxy = galaxy

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

return self.name + ' in ' + self.galaxy

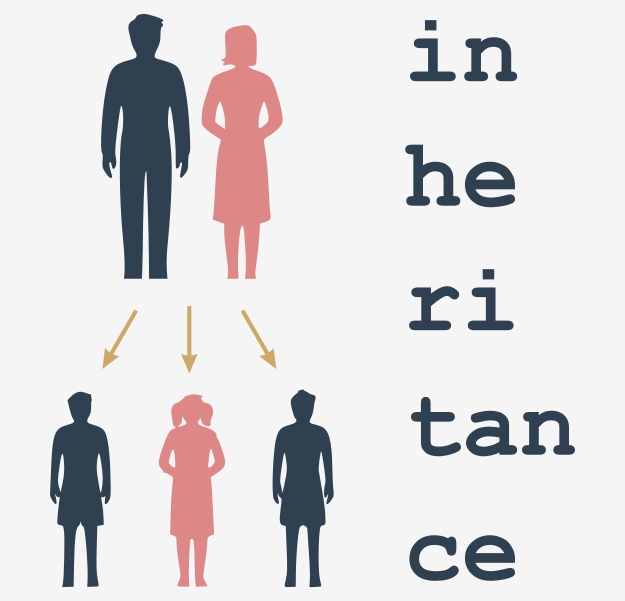
sun = Star("Sun", "Milky Way")

print(sun)

6.1.5.3 OOP Fundamentals: Inheritance

# Inheritance - why and how?

The term inheritance is older than computer programming, and it describes the common practice of passing different goods from one person to another upon that person's death. The term, when related to computer programming, has an entirely different meaning.



Let's define the term for our purposes:

Inheritance is a common practice (in object programming) of **passing attributes and methods from the superclass (defined and existing) to a newly created class, called the subclass**.

In other words, inheritance is **a way of building a new class, not from scratch, but by using an already defined repertoire of traits**. The new class inherits (and this is the key) all the already existing equipment, but is able to add some new ones if needed.

Thanks to that, it's possible to **build more specialized (more concrete) classes** using some sets of predefined general rules and behaviors.

The most important factor of the process is the relation between the superclass and all of its subclasses (note: if *B* is a subclass of *A* and *C* is a subclass of *B*, this also means than *C* is a subclass of *A*, as the relationship is fully transitive).

A very simple example of **two-level inheritance** is presented here:

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

All the presented classes are empty for now, as we're going to show you how the mutual relations between the super- and subclasses work. We'll fill them with contents soon.

We can say that:

* The Vehicle class is the superclass for both the LandVehicle and TrackedVehicle classes;
* The LandVehicle class is a subclass of Vehicle and a superclass of TrackedVehicle at the same time;
* The TrackedVehicle class is a subclass of both the Vehicle and LandVehicle classes.

The above knowledge comes from reading the code (in other words, we know it because we can see it).

Does Python know the same? Is it possible to ask Python about it? Yes, it is.

6.1.5.4 OOP Fundamentals: Inheritance

**Inheritance: issubclass()**

Python offers a function which is able to **identify a relationship between two classes**, and although its diagnosis isn't complex, it can **check if a particular class is a subclass of any other class**.

This is how it looks:

issubclass(ClassOne, ClassTwo)

The function returns True if ClassOne is a subclass of ClassTwo, and False otherwise.

Let's see it in action - it may surprise you. Look at the code in the editor. Read it carefully.

There are two nested loops. Their purpose is to **check all possible ordered pairs of classes, and to print the results of the check to determine whether the pair matches the subclass-superclass relationship**.

Run the code. The program produces the following output:

True False False

True True False

True True True

Let's make the result more readable:

| **↓ is a subclass of →** | **Vehicle** | **LandVehicle** | **TrackedVehicle** |
| --- | --- | --- | --- |
| **Vehicle** | True | False | False |
| **LandVehicle** | True | True | False |
| **TrackedVehicle** | True | True | True |

There is one important observation to make: **each class is considered to be a subclass of itself**.

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:

for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:

print(issubclass(cls1, cls2), end="\t")

print()

6.1.5.5 OOP Fundamentals: Inheritance

**Inheritance: isinstance()**

As you already know, **an object is an incarnation of a class**. This means that the object is like a cake baked using a recipe which is included inside the class.

This can generate some important issues.

Let's assume that you've got a cake (e.g., as an argument passed to your function). You want to know what recipe has been used to make it. Why? Because you want to know what to expect from it, e.g., whether it contains nuts or not, which is crucial information to some people.

Similarly, it can be crucial if the object does have (or doesn't have) certain characteristics. In other words, **whether it is an object of a certain class or not**.

Such a fact could be detected by the function named isinstance():

isinstance(objectName, ClassName)

The functions returns True if the object is an instance of the class, or False otherwise.

**Being an instance of a class means that the object (the cake) has been prepared using a recipe contained in either the class or one of its superclasses**.

Don't forget: if a subclass contains at least the same equipment as any of its superclasses, it means that objects of the subclass can do the same as objects derived from the superclass, ergo, it's an instance of its home class and any of its superclasses.

Let's test it. Analyze the code in the editor.

We've created three objects, one for each of the classes. Next, using two nested loops, we check all possible object-class pairs **to find out if the objects are instances of the classes**.

Run the code.

This is what we get:

True False False

True True False

True True True

Let's make the result more readable once again:

| **↓ is an instance of →** | **Vehicle** | **LandVehicle** | **TrackedVehicle** |
| --- | --- | --- | --- |
| **myVehicle** | True | False | False |
| **myLandVehicle** | True | True | False |
| **myTrackedVehicle** | True | True | True |

Does the table confirm our expectations?

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

myVehicle = Vehicle()

myLandVehicle = LandVehicle()

myTrackedVehicle = TrackedVehicle()

for obj in [myVehicle, myLandVehicle, myTrackedVehicle]:

for cls in [Vehicle, LandVehicle, TrackedVehicle]:

print(isinstance(obj, cls), end="\t")

print()

6.1.5.6 OOP Fundamentals: Inheritance

# Inheritance: the is operator

There is also a Python operator worth mentioning, as it refers directly to objects - here it is:

objectOne is objectTwo

**The**is**operator checks whether two variables (**objectOne**and**objectTwo**here) refer to the same object**.

Don't forget that **variables don't store the objects themselves, but only the handles pointing to the internal Python memory**.

Assigning a value of an object variable to another variable doesn't copy the object, but only its handle. This is why an operator like is may be very useful in particular circumstances.

Take a look at the code in the editor. Let's analyze it:

* there is a very simple class equipped with a simple constructor, creating just one property. The class is used to instantiate two objects. The former is then assigned to another variable, and its val property is incremented by one.
* afterward, the is operator is applied three times to check all possible pairs of objects, and all val property values are also printed.
* the last part of the code carries out another experiment. After three assignments, both strings contain the same texts, but **these texts are stored in different objects**.

The code prints:

False

False

True

1 2 1

True False

The results prove that ob1 and ob3 are actually the same objects, while str1 and str2 aren't, despite their contents being the same.

class SampleClass:

def \_\_init\_\_(self, val):

self.val = val

ob1 = SampleClass(0)

ob2 = SampleClass(2)

ob3 = ob1

ob3.val += 1

print(ob1 is ob2)

print(ob2 is ob3)

print(ob3 is ob1)

print(ob1.val, ob2.val, ob3.val)

str1 = "Mary had a little "

str2 = "Mary had a little lamb"

str1 += "lamb"

print(str1 == str2, str1 is str2)

6.1.5.7 OOP Fundamentals: Inheritance

# How Python finds properties and methods

Now we're going to look at how Python deals with inheriting methods.

Take a look at the example in the editor. Let's analyze it:

* there is a class named Super, which defines its own constructor used to assign the object's property, named name.
* the class defines the \_\_str\_\_() method, too, which makes the class able to present its identity in clear text form.
* the class is next used as a base to create a subclass named Sub. The Sub class defines its own constructor, which invokes the one from the superclass. Note how we've done it: Super.\_\_init\_\_(self, name).
* we've explicitly named the superclass, and pointed to the method to invoke \_\_init\_\_(), providing all needed arguments.
* we've instantiated one object of class Sub and printed it.

The code outputs:

My name is Andy.

Note: As there is no \_\_str\_\_() method within the Sub class, the printed string is to be produced within the Super class. This means that the \_\_str\_\_() method has been inherited by the Sub class.

class Super:

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

self.name = name

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

return "My name is " + self.name + "."

class Sub(Super):

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

Super.\_\_init\_\_(self, name)

obj = Sub("Andy")

print(obj)

6.1.5.8 OOP Fundamentals: Inheritance

**How Python finds properties and methods: continued**

Look at the code in the editor. We've modified it to show you another method of accessing any entity defined inside the superclass.

In the last example, we explicitly named the superclass. In this example, we make use of the super() function, which **accesses the superclass without needing to know its name**:

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

The super() function creates a context in which you don't have to (moreover, you mustn't) pass the self argument to the method being invoked - this is why it's possible to activate the superclass constructor using only one argument.

Note: you can use this mechanism not only to **invoke the superclass constructor, but also to get access to any of the resources available inside the superclass**.

class Super:

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

self.name = name

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

return "My name is " + self.name + "."

class Sub(Super):

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

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

obj = Sub("Andy")

print(obj)

6.1.5.9 OOP Fundamentals: Inheritance

**How Python finds properties and methods: continued**

Let's try to do something similar, but with properties (more precisely: with **class variables**).

Take a look at the example in the editor.

As you can see, the Super class defines one class variable named supVar, and the Sub class defines a variable named subVar.

Both these variables are visible inside the object of class Sub - this is why the code outputs:

2

1

# Testing properties: class variables

class Super:

supVar = 1

class Sub(Super):

subVar = 2

obj = Sub()

print(obj.subVar)

print(obj.supVar)

6.1.5.10 OOP Fundamentals: Inheritance

**How Python finds properties and methods: continued**

The same effect can be observed with **instance variables** - take a look at the second example in the editor.

The Sub class constructor creates an instance variable named subVar, while the Super constructor does the same with a variable named supVar. As previously, both variables are accessible from within the object of class Sub.

The program's output is:

12

11

Note: the existence of the supVar variable is obviously conditioned by the Super class constructor invocation. Omitting it would result in the absence of the variable in the created object (try it yourself).

# Testing properties: instance variables

class Super:

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

self.supVar = 11

class Sub(Super):

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

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

self.subVar = 12

obj = Sub()

print(obj.subVar)

print(obj.supVar)

6.1.5.11 OOP Fundamentals: Inheritance

# How Python finds properties and methods: continued

It's now possible to formulate a general statement describing Python's behavior.

When you try to access any object's entity, Python will try to (in this order):

* find it **inside the object** itself;
* find it **in all classes** involved in the object's inheritance line from bottom to top;

If both of the above fail, an **exception (**AttributeError**) is raised**.

The first condition may need some additional attention. As you know, all objects deriving from a particular class may have different sets of attributes, and some of the attributes may be added to the object a long time after the object's creation.

The example in the editor summarizes this in a **three-level inheritance line**. Analyze it carefully.

All the comments we've made so far are related to **single inheritance**, when a subclass has exactly one superclass. This is the most common situation (and the recommended one, too).

Python, however, offers much more here. In the next lessons we're going to show you some examples of **multiple inheritance**.

class Level1:

varia1 = 100

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

self.var1 = 101

def fun1(self):

return 102

class Level2(Level1):

varia2 = 200

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

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

self.var2 = 201

def fun2(self):

return 202

class Level3(Level2):

varia3 = 300

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

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

self.var3 = 301

def fun3(self):

return 302

obj = Level3()

print(obj.varia1, obj.var1, obj.fun1())

print(obj.varia2, obj.var2, obj.fun2())

print(obj.varia3, obj.var3, obj.fun3())

6.1.5.12 OOP Fundamentals: Inheritance

**How Python finds properties and methods: continued**

**Multiple inheritance occurs when a class has more than one superclass**.

Syntactically, such inheritance is presented as a comma-separated list of superclasses put inside parentheses after the new class name - just like here:

class SuperA:

varA = 10

def funA(self):

return 11

class SuperB:

varB = 20

def funB(self):

return 21

class Sub(SuperA, SuperB):

pass

obj = Sub()

print(obj.varA, obj.funA())

print(obj.varB, obj.funB())

The Sub class has two superclasses: SuperA and SuperB. This means that the Sub class **inherits all the goods offered by both**SuperA**and**SuperB.

The code prints:

10 11

20 21

Now it's time to introduce a brand new term - **overriding**.

What do you think will happen if more than one of the superclasses defines an entity of a particular name?

class SuperA:

varA = 10

def funA(self):

return 11

class SuperB:

varB = 20

def funB(self):

return 21

class Sub(SuperA, SuperB):

pass

obj = Sub()

print(obj.varA, obj.funA())

print(obj.varB, obj.funB())

6.1.5.13 OOP Fundamentals: Inheritance

# How Python finds properties and methods: continued

Let's analyze the example in the editor.

Both, Level1 and Level2 classes define a method named fun() and a property named var. Does this mean that the Level3 class object will be able to access two copies of each entity? Not at all.

**The entity defined later (in the inheritance sense) overrides the same entity defined earlier**. This is why the code produces the following output:

200 201

As you can see, the var class variable and fun() method from the Level2 class override the entities of the same names derived from the Level1 class.

This feature can be intentionally used to modify default (or previously defined) class behaviors when any of its classes needs to act in a different way to its ancestor.

We can also say that **Python looks for an entity from bottom to top**, and is fully satisfied with the first entity of the desired name.

How does it work when a class has two ancestors offering the same entity, and they lie on the same level? In other words, what should you expect when a class emerges using multiple inheritance? Let's look at this.

class Level1:

var = 100

def fun(self):

return 101

class Level2(Level1):

var = 200

def fun(self):

return 201

class Level3(Level2):

pass

obj = Level3()

print(obj.var, obj.fun())

6.1.5.14 OOP Fundamentals: Inheritance

# How Python finds properties and methods: continued

Let's take a look at the example in the editor.

The Sub class inherits goods from two superclasses, Left and Right (these names are intended to be meaningful).

There is no doubt that the class variable varRight comes from the Right class, and varLeft comes from Left respectively.

This is clear. But where does var come from? Is it possible to guess it? The same problem is encountered with the fun() method - will it be invoked from Left or from Right? Let's run the program - its output is:

L LL RR Left

This proves that both unclear cases have a solution inside the Left class. Is this a sufficient premise to formulate a general rule? Yes, it is.

We can say that **Python looks for object components** in the following order:

* **inside the object** itself;
* **in its superclasses**, from bottom to top;
* if there is more than one class on a particular inheritance path, Python scans them from left to right.

Do you need anything more? Just make a small amendment in the code - replace: class Sub(Left, Right): with: class Sub(Right, Left):, then run the program again, and see what happens.

What do you see now? We see:

R LL RR Right

Do you see the same, or something different?

class Left:

var = "L"

varLeft = "LL"

def fun(self):

return "Left"

class Right:

var = "R"

varRight = "RR"

def fun(self):

return "Right"

class Sub(Left, Right):

pass

obj = Sub()

print(obj.var, obj.varLeft, obj.varRight, obj.fun())

6.1.5.15 OOP Fundamentals: Inheritance

# How to build a hierarchy of classes

Building a hierarchy of classes isn't just art for art's sake.

If you divide a problem among classes and decide which of them should be located at the top and which should be placed at the bottom of the hierarchy, you have to carefully analyze the issue, but before we show you how to do it (and how not to do it), we want to highlight an interesting effect. It's nothing extraordinary (it's just a consequence of the general rules presented earlier), but remembering it may be key to understanding how some codes work, and how the effect may be used to build a flexible set of classes.

Take a look at the code in the editor. Let's analyze it:

* there are two classes, named One and Two, while Two is derived from One. Nothing special. However, one thing looks remarkable - the doit() method.
* the doit()method is **defined twice**: originally inside One and subsequently inside Two. The essence of the example lies in the fact that it is **invoked just once** - inside One.

The question is - which of the two methods will be invoked by the last two lines of the code?

The first invocation seems to be simple, and it is simple, actually - invoking doanything() from the object named one will obviously activate the first of the methods.

The second invocation needs some attention. It's simple, too if you keep in mind how Python finds class components. The second invocation will launch doit() in the form existing inside the Two class, regardless of the fact that the invocation takes place within the One class.

In effect, the code produces the following output:

doit from One

doit from Two

Note: the situation in which **the subclass is able to modify its superclass behavior (just like in the example) is called polymorphism**. The word comes from Greek (polys: "many, much" and morphe, "form, shape"), which means that one and the same class can take various forms depending on the redefinitions done by any of its subclasses.

The method, redefined in any of the superclasses, thus changing the behavior of the superclass, is called **virtual**.

In other words, no class is given once and for all. Each class's behavior may be modified at any time by any of its subclasses.

We're going to show you **how to use polymorphism to extend class flexibility**.

class One:

def doit(self):

print("doit from One")

def doanything(self):

self.doit()

class Two(One):

def doit(self):

print("doit from Two")

one = One()

two = Two()

one.doanything()

two.doanything()

6.1.5.16 OOP Fundamentals: Inheritance

# How to build a hierarchy of classes: continued

Look at the example in the editor.

Does it resemble anything? Yes, of course it does. It refers to the example shown at the beginning of the module when we talked about the general concepts of objective programming.

It may look weird, but we didn't use inheritance in any way - just to show you that it doesn't limit us, and we managed to get ours.

We defined two separate classes able to produce two different kinds of land vehicles. The main difference between them is in how they turn. A wheeled vehicle just turns the front wheels (generally). A tracked vehicle has to stop one of the tracks.

Can you follow the code?

* a tracked vehicle performs a turn by stopping and moving on one of its tracks (this is done by the controltrack() method, which will be implemented later)
* a wheeled vehicle turns when its front wheels turn (this is done by the turnfrontwheels() method)
* the turn() method uses the method suitable for each particular vehicle.

Can you see **what's wrong with the code**?

The turn() methods look too similar to leave them in this form.

Let's rebuild the code - we're going to introduce a superclass to gather all the similar aspects of the driving vehicles, moving all the specifics to the subclasses.

import time

class TrackedVehicle:

def controltrack(left, stop):

pass

def turn(left):

controltrack(left, True)

time.sleep(0.25)

controltrack(left, False)

class WheeledVehicle:

def turnfrontwheels(left, on):

pass

def turn(left):

turnfrontwheels(left, True)

time.sleep(0.25)

turnfrontwheel(left, False)

6.1.5.17 OOP Fundamentals: Inheritance

# How to build a hierarchy of classes: continued

Look at the code in the editor again. This is what we've done:

* we defined a superclass named Vehicle, which uses the turn() method to implement a general scheme of turning, while the turning itself is done by a method named changedirection(); note: the former method is empty, as we are going to put all the details into the subclass (such a method is often called an **abstract method**, as it only demonstrates some possibility which will be instantiated later)
* we defined a subclass named TrackedVehicle (note: it's derived from the Vehicle class) which instantiated the changedirection() method by using the specific (concrete) method named controltrack()
* respectively, the subclass named WheeledVehicle does the same trick, but uses the turnfrontwheel() method to force the vehicle to turn.

The most important advantage (omitting readability issues) is that this form of code enables you to implement a brand new turning algorithm just by modifying the turn() method, which can be done in just one place, as all the vehicles will obey it.

This is how **polymorphism helps the developer to keep the code clean and consistent**.

import time

class Vehicle:

def changedirection(left, on):

pass

def turn(left):

changedirection(left, True)

time.sleep(0.25)

changedirection(left, False)

class TrackedVehicle(Vehicle):

def controltrack(left, stop):

pass

def changedirection(left, on):

controltrack(left, on)

class WheeledVehicle(Vehicle):

def turnfrontwheels(left, on):

pass

def changedirection(left, on):

turnfrontwheels(left, on)

6.1.5.18 OOP Fundamentals: Inheritance

# How to build a hierarchy of classes: continued

Inheritance is not the only way of constructing adaptable classes. You can achieve the same goals (not always, but very often) by using a technique named composition.

**Composition is the process of composing an object using other different objects**. The objects used in the composition deliver a set of desired traits (properties and/or methods) so we can say that they act like blocks used to build a more complicated structure.

It can be said that:

* **inheritance extends a class's capabilities** by adding new components and modifying existing ones; in other words, the complete recipe is contained inside the class itself and all its ancestors; the object takes all the class's belongings and makes use of them;
* **composition projects a class as a container** able to store and use other objects (derived from other classes) where each of the objects implements a part of a desired class's behavior.

Let us illustrate the difference by using the previously defined vehicles. The previous approach led us to a hierarchy of classes in which the top-most class was aware of the general rules used in turning the vehicle, but didn't know how to control the appropriate components (wheels or tracks).

The subclasses implemented this ability by introducing specialized mechanisms. Let's do (almost) the same thing, but using composition. The class - like in the previous example - is aware of how to turn the vehicle, but the actual turn is done by a specialized object stored in a property named controller. The controller is able to control the vehicle by manipulating the relevant vehicle's parts.

Take a look into the editor - this is how it could look.

There are two classes named Tracks and Wheels - they know how to control the vehicle's direction. There is also a class named Vehicle which can use any of the available controllers (the two already defined, or any other defined in the future) - the controller itself is passed to the class during initialization.

In this way, the vehicle's ability to turn is composed using an external object, not implemented inside the Vehicle class.

In other words, we have a universal vehicle and can install either tracks or wheels onto it.

The code produces the following output:

wheels: True True

wheels: True False

tracks: False True

tracks: False False

import time

class Tracks:

def changedirection(self, left, on):

print("tracks: ", left, on)

class Wheels:

def changedirection(self, left, on):

print("wheels: ", left, on)

class Vehicle:

def \_\_init\_\_(self, controller):

self.controller = controller

def turn(self, left):

self.controller.changedirection(left, True)

time.sleep(0.25)

self.controller.changedirection(left, False)

wheeled = Vehicle(Wheels())

tracked = Vehicle(Tracks())

wheeled.turn(True)

tracked.turn(False)

6.1.5.19 OOP Fundamentals: Inheritance

# Single inheritance vs. multiple inheritance

As you already know, there are no obstacles to using multiple inheritance in Python. You can derive any new class from more than one previously defined classes.

There is only one "but". The fact that you can do it does not mean you have to.

Don't forget that:

* a single inheritance class is always simpler, safer, and easier to understand and maintain;
* multiple inheritance is always risky, as you have many more opportunities to make a mistake in identifying these parts of the superclasses which will effectively influence the new class;
* multiple inheritance may make overriding extremely tricky; moreover, using the super() function becomes ambiguous;
* multiple inheritance violates the **single responsibility principle** (more details here: <https://en.wikipedia.org/wiki/Single_responsibility_principle>) as it makes a new class of two (or more) classes that know nothing about each other;
* we strongly suggest multiple inheritance as the last of all possible solutions - if you really need the many different functionalities offered by different classes, composition may be a better alternative.

6.1.6.1 Exceptions once again

**More about exceptions**

Discussing object programming offers a very good opportunity to return to exceptions. The objective nature of Python's exceptions makes them a very flexible tool, able to fit to specific needs, even those you don't yet know about.

Before we dive into the **objective face of exceptions**, we want to show you some syntactical and semantic aspects of the way in which Python treats the try-except block, as it offers a little more than what we have presented so far.

The first feature we want discuss here is an additional, possible branch that can be placed inside (or rather, directly behind) the try-except block - it's the part of the code starting with else - just like in the example in the editor.

A code labelled in this way is executed when (and only when) no exception has been raised inside the try: part. We can say that exactly one branch can be executed after try: - either the one beginning with except (don't forget that there can be more than one branch of this kind) or the one starting with else.

Note: the else: branch has to be located after the last except branch.

The example code produces the following output:

Everything went fine

0.5

Division failed

None

def reciprocal(n):

try:

n = 1 / n

except ZeroDivisionError:

print("Division failed")

return None

else:

print("Everything went fine")

return n

print(reciprocal(2))

print(reciprocal(0))

6.1.6.2 Exceptions once again

**More about exceptions**

The try-except block can be extended in one more way - by adding a part headed by the finally keyword (it must be the last branch of the code designed to handle exceptions).

Note: these two variants (else and finally) aren't dependent in any way, and they can coexist or occur independently.

The finally block is always executed (it finalizes the try-except block execution, hence its name), no matter what happened earlier, even when raising an exception, no matter whether this has been handled or not.

Look at the code in the editor. It outputs:

Everything went fine

It's time to say good bye

0.5

Division failed

It's time to say good bye

None

def reciprocal(n):

try:

n = 1 / n

except ZeroDivisionError:

print("Division failed")

n = None

else:

print("Everything went fine")

finally:

print("It's time to say goodbye")

return n

print(reciprocal(2))

print(reciprocal(0))

6.1.6.3 Exceptions once again

# Exceptions are classes

All the previous examples were content with detecting a specific kind of exception and responding to it in an appropriate way. Now we're going to delve deeper, and look inside the exception itself.

You probably won't be surprised to learn that **exceptions are classes**. Furthermore, when an exception is raised, an object of the class is instantiated, and goes through all levels of program execution, looking for the except branch that is prepared to deal with it.

Such an object carries some useful information which can help you to precisely identify all aspects of the pending situation. To achieve that goal, Python offers a special variant of the exception clause - you can find it in the editor.

As you can see, the except statement is extended, and contains an additional phrase starting with the as keyword, followed by an identifier. The identifier is designed to catch the exception object so you can analyze its nature and draw proper conclusions.

Note: the identifier's scope covers its except branch, and doesn't go any further.

The example presents a very simple way of utilizing the received object - just print it out (as you can see, the output is produced by the object's \_\_str\_\_() method) and it contains a brief message describing the reason.

The same message will be printed if there is no fitting except block in the code, and Python is forced to handle it alone.

try:

i = int("Hello!")

except Exception as e:

print(e)

print(e.\_\_str\_\_())

6.1.6.4 Exceptions once again

# Exceptions are classes

All the built-in Python exceptions form a hierarchy of classes. There is no obstacle to extending it if you find it reasonable.

Look at the code in the editor.

This program dumps all predefined exception classes in the form of a tree-like printout.

As **a tree is a perfect example of a recursive data structure**, a recursion seems to be the best tool to traverse through it. The printExcTree() function takes two arguments:

* a point inside the tree from which we start traversing the tree;
* a nesting level (we'll use it to build a simplified drawing of the tree's branches)

Let's start from the tree's root - the root of Python's exception classes is the BaseException class (it's a superclass of all other exceptions).

For each of the encountered classes, perform the same set of operations:

* print its name, taken from the \_\_name\_\_ property;
* iterate through the list of subclasses delivered by the \_\_subclasses\_\_() method, and recursively invoke the printExcTree() function, incrementing the nesting level respectively.

Note how we've drawn the branches and forks. The printout isn't sorted in any way - you can try to sort it yourself, if you want a challenge. Moreover, there are some subtle inaccuracies in the way in which some branches are presented. That can be fixed, too, if you wish.

This is how it looks:

BaseException

+---Exception

| +---TypeError

| +---StopAsyncIteration

| +---StopIteration

| +---ImportError

| | +---ModuleNotFoundError

| | +---ZipImportError

| +---OSError

| | +---ConnectionError

| | | +---BrokenPipeError

| | | +---ConnectionAbortedError

| | | +---ConnectionRefusedError

| | | +---ConnectionResetError

| | +---BlockingIOError

| | +---ChildProcessError

| | +---FileExistsError

| | +---FileNotFoundError

| | +---IsADirectoryError

| | +---NotADirectoryError

| | +---InterruptedError

| | +---PermissionError

| | +---ProcessLookupError

| | +---TimeoutError

| | +---UnsupportedOperation

| | +---herror

| | +---gaierror

| | +---timeout

| | +---Error

| | | +---SameFileError

| | +---SpecialFileError

| | +---ExecError

| | +---ReadError

| +---EOFError

| +---RuntimeError

| | +---RecursionError

| | +---NotImplementedError

| | +---\_DeadlockError

| | +---BrokenBarrierError

| +---NameError

| | +---UnboundLocalError

| +---AttributeError

| +---SyntaxError

| | +---IndentationError

| | | +---TabError

| +---LookupError

| | +---IndexError

| | +---KeyError

| | +---CodecRegistryError

| +---ValueError

| | +---UnicodeError

| | | +---UnicodeEncodeError

| | | +---UnicodeDecodeError

| | | +---UnicodeTranslateError

| | +---UnsupportedOperation

| +---AssertionError

| +---ArithmeticError

| | +---FloatingPointError

| | +---OverflowError

| | +---ZeroDivisionError

| +---SystemError

| | +---CodecRegistryError

| +---ReferenceError

| +---BufferError

| +---MemoryError

| +---Warning

| | +---UserWarning

| | +---DeprecationWarning

| | +---PendingDeprecationWarning

| | +---SyntaxWarning

| | +---RuntimeWarning

| | +---FutureWarning

| | +---ImportWarning

| | +---UnicodeWarning

| | +---BytesWarning

| | +---ResourceWarning

| +---error

| +---Verbose

| +---Error

| +---TokenError

| +---StopTokenizing

| +---Empty

| +---Full

| +---\_OptionError

| +---TclError

| +---SubprocessError

| | +---CalledProcessError

| | +---TimeoutExpired

| +---Error

| | +---NoSectionError

| | +---DuplicateSectionError

| | +---DuplicateOptionError

| | +---NoOptionError

| | +---InterpolationError

| | | +---InterpolationMissingOptionError

| | | +---InterpolationSyntaxError

| | | +---InterpolationDepthError

| | +---ParsingError

| | | +---MissingSectionHeaderError

| +---InvalidConfigType

| +---InvalidConfigSet

| +---InvalidFgBg

| +---InvalidTheme

| +---EndOfBlock

| +---BdbQuit

| +---error

| +---\_Stop

| +---PickleError

| | +---PicklingError

| | +---UnpicklingError

| +---\_GiveupOnSendfile

| +---error

| +---LZMAError

| +---RegistryError

| +---ErrorDuringImport

+---GeneratorExit

+---SystemExit

+---KeyboardInterrupt

def printExcTree(thisclass, nest = 0):

if nest > 1:

print(" |" \* (nest - 1), end="")

if nest > 0:

print(" +---", end="")

print(thisclass.\_\_name\_\_)

for subclass in thisclass.\_\_subclasses\_\_():

printExcTree(subclass, nest + 1)

printExcTree(BaseException)

6.1.6.5 Exceptions once again

**Detailed anatomy of exceptions**

Let's take a closer look at the exception's object, as there are some really interesting elements here (we'll return to the issue soon when we consider Python's input/output base techniques, as their exception subsystem extends these objects a bit).

The BaseException class introduces a property named args. It's a **tuple designed to gather all arguments passed to the class constructor**. It is empty if the construct has been invoked without any arguments, or contains just one element when the constructor gets one argument (we don't count the self argument here), and so on.

We've prepared a simple function to print the args property in an elegant way. You can see the function in the editor.

We've used the function to print the contents of the args property in three different cases, where the exception of the Exception class is raised in three different ways. To make it more spectacular, we've also printed the object itself, along with the result of the \_\_str\_\_() invocation.

The first case looks routine - there is just the name Exception after the raise keyword. This means that the object of this class has been created in a most routine way.

The second and third cases may look a bit weird at first glance, but there's nothing odd here - these are just the constructor invocations. In the second raise statement, the constructor is invoked with one argument, and in the third, with two.

As you can see, the program output reflects this, showing the appropriate contents of the args property:

: :

my exception : my exception : my exception

('my', 'exception') : ('my', 'exception') : ('my', 'exception')

def printargs(args):

lng = len(args)

if lng == 0:

print("")

elif lng == 1:

print(args[0])

else:

print(str(args))

try:

raise Exception

except Exception as e:

print(e, e.\_\_str\_\_(), sep=' : ' ,end=' : ')

printargs(e.args)

try:

raise Exception("my exception")

except Exception as e:

print(e, e.\_\_str\_\_(), sep=' : ', end=' : ')

printargs(e.args)

try:

raise Exception("my", "exception")

except Exception as e:

print(e, e.\_\_str\_\_(), sep=' : ', end=' : ')

printargs(e.args)

6.1.6.6 Exceptions once again

# How to create your own exception

The exceptions hierarchy is neither closed nor finished, and you can always extend it if you want or need to create your own world populated with your own exceptions.

It may be useful when you create a complex module which detects errors and raises exceptions, and you want the exceptions to be easily distinguishable from any others brought by Python.

This is done by **defining your own, new exceptions as subclasses derived from predefined ones**.

Note: if you want to create an exception which will be utilized as a specialized case of any built-in exception, derive it from just this one. If you want to build your own hierarchy, and don't want it to be closely connected to Python's exception tree, derive it from any of the top exception classes, like Exception.

Imagine that you've created a brand new arithmetic, ruled by your own laws and theorems. It's clear that division has been redefined, too, and has to behave in a different way than routine dividing. It's also clear that this new division should raise its own exception, different from the built-in ZeroDivisionError, but it's reasonable to assume that in some circumstances, you (or your arithmetic's user) may want to treat all zero divisions in the same way.

Demands like these may be fulfilled in the way presented in the editor. Look at the code, and let's analyze it:

* We've defined our own exception, named MyZeroDivisionError, derived from the built-in ZeroDivisionError. As you can see, we've decided not to add any new components to the class.  
    
  In effect, an exception of this class can be - depending on the desired point of view - treated like a plain ZeroDivisionError, or considered separately.

* The doTheDivision() function raises either a MyZeroDivisionError or ZeroDivisionError exception, depending on the argument's value.  
    
  The function is invoked four times in total, while the first two invocations are handled using only one except branch (the more general one) and the last two ones with two different branches, able to distinguish the exceptions (don't forget: the order of the branches makes a fundamental difference!)

class MyZeroDivisionError(ZeroDivisionError):

pass

def doTheDivision(mine):

if mine:

raise MyZeroDivisionError("some worse news")

else:

raise ZeroDivisionError("some bad news")

for mode in [False, True]:

try:

doTheDivision(mode)

except ZeroDivisionError:

print('Division by zero')

for mode in [False, True]:

try:

doTheDivision(mode)

except MyZeroDivisionError:

print('My division by zero')

except ZeroDivisionError:

print('Original division by zero')

6.1.6.7 Exceptions once again

# How to create your own exception: continued

When you're going to build a completely new universe filled with completely new creatures that have nothing in common with all the familiar things, you may want to **build your own exception structure**.

For example, if you work on a large simulation system which is intended to model the activities of a pizza restaurant, it can be desirable to form a separate hierarchy of exceptions.

You can start building it by **defining a general exception as a new base class** for any other specialized exception. We've done in in the following way:

class PizzaError(Exception):

def \_\_init\_\_(self, pizza, message):

Exception.\_\_init\_\_(message)

self.pizza = pizza

Note: we're going to collect more specific information here than a regular Exception does, so our constructor will take two arguments:

* one specifying a pizza as a subject of the process,
* and one containing a more or less precise description of the problem.

As you can see, we pass the second parameter to the superclass constructor, and save the first inside our own property.

A more specific problem (like an excess of cheese) can require a more specific exception. It's possible to derive the new class from the already defined PizzaError class, like we've done here:

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza, cheese, message):

PizzaError.\_init\_\_(self, pizza, message)

self.cheese = cheese

The TooMuchCheeseError exception needs more information than the regular PizzaError exception, so we add it to the constructor - the name cheese is then stored for further processing.

lass PizzaError(Exception):

def \_\_init\_\_(self, pizza, message):

Exception.\_\_init\_\_(message)

self.pizza = pizza

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza, cheese, message):

PizzaError.\_init\_\_(self, pizza, message)

self.cheese = cheese

6.1.6.8 Exceptions once again

# How to create your own exception: continued

Look at the code in the editor. We've coupled together the two previously defined exceptions and harnessed them to work in a small example snippet.

One of these is raised inside the makePizza() function when any of these two erroneous situations is discovered: a wrong pizza request, or a request for too much cheese.

Note:

* removing the branch starting with except TooMuchCheeseError will cause all appearing exceptions to be classified as PizzaError;
* removing the branch starting with except PizzaErrorwill cause the TooMuchCheeseError exceptions to remain unhandled, and will cause the program to terminate.

The previous solution, although elegant and efficient, has one important weakness. Due to the somewhat easygoing way of declaring the constructors, the new exceptions cannot be used as-is, without a full list of required arguments.

We'll remove this weakness by **setting the default values for all constructor parameters**. Take a look:

class PizzaError(Exception):

def \_\_init\_\_(self, pizza='uknown', message=''):

Exception.\_\_init\_\_(self, message)

self.pizza = pizza

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza='uknown', cheese='>100', message=''):

PizzaError.\_\_init\_\_(self, pizza, message)

self.cheese = cheese

def makePizza(pizza, cheese):

if pizza not in ['margherita', 'capricciosa', 'calzone']:

raise PizzaError

if cheese > 100:

raise TooMuchCheeseError

print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:

try:

makePizza(pz, ch)

except TooMuchCheeseError as tmce:

print(tmce, ':', tmce.cheese)

except PizzaError as pe:

print(pe, ':', pe.pizza)

Now, if the circumstances permit, it is possible to use the class names alone.

class PizzaError(Exception):

def \_\_init\_\_(self, pizza, message):

Exception.\_\_init\_\_(self, message)

self.pizza = pizza

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza, cheese, message):

PizzaError.\_\_init\_\_(self, pizza, message)

self.cheese = cheese

def makePizza(pizza, cheese):

if pizza not in ['margherita', 'capricciosa', 'calzone']:

raise PizzaError(pizza, "no such pizza on the menu")

if cheese > 100:

raise TooMuchCheeseError(pizza, cheese, "too much cheese")

print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:

try:

makePizza(pz, ch)

except TooMuchCheeseError as tmce:

print(tmce, ':', tmce.cheese)

except PizzaError as pe:

print(pe, ':', pe.pizza)

6.1.7.1 Generators and closures

# Generators - where to find them

**Generator** - what do you associate this word with? Perhaps it refers to some electronic device. Or perhaps it refers to a heavy and serious machine designed to produce power, electrical or other.

A Python generator is **a piece of specialized code able to produce a series of values, and to control the iteration process**. This is why generators are very often called **iterators**, and although some may find a very subtle distinction between these two, we'll treat them as one.

You may not realize it, but you've encountered generators many, many times before. Take a look at the very simple snippet:

for i in range(5):

print(i)

The range() function is, in fact, a generator, which is (in fact, again) an iterator

What is the difference?

A function returns one, well-defined value - it may be the result of a more or less complex evaluation of, e.g., a polynomial, and is invoked once - only once.

A generator **returns a series of values**, and in general, is (implicitly) invoked more than once.

In the example, the range() generator is invoked six times, providing five subsequent values from zero to four, and finally signaling that the series is complete.

The above process is completely transparent. Let's shed some light on it. Let's show you the **iterator protocol**.

for i in range(5):

print(i)

6.1.7.2 Generators and closures

**Generators - where to find them: continued**

The **iterator protocol is a way in which an object should behave to conform to the rules imposed by the context of the**for**and**in**statements**. An object conforming to the iterator protocol is called an **iterator**.

An iterator must provide two methods:

* \_\_iter\_\_() which should **return the object itself** and which is invoked once (it's needed for Python to successfully start the iteration)
* \_\_next\_\_() which is intended to **return the next value** (first, second, and so on) of the desired series - it will be invoked by the for/in statements in order to pass through the next iteration; if there are no more values to provide, the method should **raise the**StopIteration**exception**.

Does it sound strange? Not at all. Look at the example in the editor.

We've built a class able to iterate through the first n values (where n is a constructor parameter) of the Fibonacci numbers.

Let us remind you - the Fibonacci numbers (*Fibi*) are defined as follows:

*Fib1 = 1  
Fib2 = 1  
Fibi = Fibi-1 + Fibi-2*

In other words:

* the first two Fibonacci numbers are equal to 1;
* any other Fibonacci number is the sum of the two previous ones (e.g., Fib3 = 2, Fib4 = 3, Fib5 = 5, and so on)

Let's dive into the code:

* lines 2 through 6: the class constructor prints a message (we'll use this to trace the class's behavior), prepares some variables (\_\_n to store the series limit, \_\_i to track the current Fibonacci number to provide, and \_\_p1 along with \_\_p2 to save the two previous numbers);

* lines 8 through 10: the \_\_iter\_\_ method is obliged to return the iterator object itself; its purpose may be a bit ambiguous here, but there's no mystery; try to imagine an object which is not an iterator (e.g., it's a collection of some entities), but one of its components is an iterator able to scan the collection; the \_\_iter\_\_ method should **extract the iterator and entrust it with the execution of the iteration protocol**; as you can see, the method starts its action by printing a message;

* lines 12 through 21: the \_\_next\_\_ method is responsible for creating the sequence; it's somewhat wordy, but this should make it more readable; first, it prints a message, then it updates the number of desired values, and if it reaches the end of the sequence, the method breaks the iteration by raising the StopIteration exception; the rest of the code is simple, and it precisely reflects the definition we showed you earlier;

* lines 23 and 24 make use of the iterator.

The code produces the following output:

\_\_init\_\_

\_\_iter\_\_

\_\_next\_\_

1

\_\_next\_\_

1

\_\_next\_\_

2

\_\_next\_\_

3

\_\_next\_\_

5

\_\_next\_\_

8

\_\_next\_\_

13

\_\_next\_\_

21

\_\_next\_\_

34

\_\_next\_\_

55

\_\_next\_\_

Look:

* the iterator object is instantiated first;
* next, Python invokes the \_\_iter\_\_ method to get access to the actual iterator;
* the \_\_next\_\_ method is invoked eleven times - the first ten times produce useful values, while the eleventh terminates the iteration.

class Fib:

def \_\_init\_\_(self, nn):

print("\_\_init\_\_")

self.\_\_n = nn

self.\_\_i = 0

self.\_\_p1 = self.\_\_p2 = 1

def \_\_iter\_\_(self):

print("\_\_iter\_\_")

return self

def \_\_next\_\_(self):

print("\_\_next\_\_")

self.\_\_i += 1

if self.\_\_i > self.\_\_n:

raise StopIteration

if self.\_\_i in [1, 2]:

return 1

ret = self.\_\_p1 + self.\_\_p2

self.\_\_p1, self.\_\_p2 = self.\_\_p2, ret

return ret

for i in Fib(10):

print(i)

6.1.7.3 Generators and closures

# Generators - where to find them: continued

The previous example shows you a solution where the **iterator object is a part of a more complex class**.

The code isn't really sophisticated, but it presents the concept in a clear way.

Take a look at the code in the editor.

We've built the Fib iterator into another class (we can say that we've composed it into the Class class). It's instantiated along with Class's object.

The object of the class may be used as an iterator when (and only when) it positively answers to the \_\_iter\_\_ invocation - this class can do it, and if it's invoked in this way, it provides an object able to obey the iteration protocol.

This is why the output of the code is the same as previously, although the object of the Fib class isn't used explicitly inside the for loop's context.

class Fib:

def \_\_init\_\_(self, nn):

self.\_\_n = nn

self.\_\_i = 0

self.\_\_p1 = self.\_\_p2 = 1

def \_\_iter\_\_(self):

print("Fib iter")

return self

def \_\_next\_\_(self):

self.\_\_i += 1

if self.\_\_i > self.\_\_n:

raise StopIteration

if self.\_\_i in [1, 2]:

return 1

ret = self.\_\_p1 + self.\_\_p2

self.\_\_p1, self.\_\_p2 = self.\_\_p2, ret

return ret

class Class:

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

self.\_\_iter = Fib(n)

def \_\_iter\_\_(self):

print("Class iter")

return self.\_\_iter;

object = Class(8)

for i in object:

print(i)

6.1.7.4 Generators and closures

# The yield statement

The iterator protocol isn't particularly difficult to understand and use, but it is also indisputable that the **protocol is rather inconvenient**.

The main discomfort it brings is **the need to save the state of the iteration between subsequent**\_\_iter\_\_**invocations**.

For example, the Fib iterator is forced to precisely store the place in which the last invocation has been stopped (i.e., the evaluated number and the values of the two previous elements). This makes the code larger and less comprehensible.

This is why Python offers a much more effective, convenient, and elegant way of writing iterators.

The concept is fundamentally based on a very specific and powerful mechanism provided by the yield keyword.

You may think of the yield keyword as a smarter sibling of the return statement, with one essential difference.

Take a look at this function:

def fun(n):

for i in range(n):

return i

It looks strange, doesn't it? It's clear that the for loop has no chance to finish its first execution, as the return will break it irrevocably.

Moreover, invoking the function won't change anything - the for loop will start from scratch and will be broken immediately.

We can say that such a function is not able to save and restore its state between subsequent invocations.

This also means that a function like this **cannot be used as a generator**.

We've replaced exactly one word in the code - can you see it?

def fun(n):

for i in range(n):

yield i

We've added yield instead of return. This little amendment **turns the function into a generator**, and executing the yield statement has some very interesting effects.

First of all, it provides the value of the expression specified after the yield keyword, just like return, but doesn't lose the state of the function.

All the variables' values are frozen, and wait for the next invocation, when the execution is resumed (not taken from scratch, like after return).

There is one important limitation: such a **function should not be invoked explicitly** as - in fact - it isn't a function anymore; **it's a generator object**.

The invocation will **return the object's identifier**, not the series we expect from the generator.

Due to the same reasons, the previous function (the one with the return statement) may only be invoked explicitly, and must not be used as a generator.

## How to build a generator

Let us show you the new generator in action.

This is how we can use it:

def fun(n):

for i in range(n):

yield i

for v in fun(5):

print(v)

Can you guess the output?

Check

0

1

2

3

4

6.1.7.5 Generators and closures

# How to build your own generator

What if you need a **generator to produce the first *n* powers of *2***?

Nothing easier. Just look at the code in the editor.

Can you guess the output? Run the code to check your guesses.

Generators may also be used within **list comprehensions**, just like here:

def powersOf2(n):

pow = 1

for i in range(n):

yield pow

pow \*= 2

t = [x for x in powersOf2(5)]

print(t)

Run the example and check the output.

The list() function can transform a series of subsequent generator invocations into **a real list**:

def powersOf2(n):

pow = 1

for i in range(n):

yield pow

pow \*= 2

t = list(powersOf2(3))

print(t)

Again, try to predict the output and run the code to check your predictions.

Moreover, the context created by the in operator allows you to use a generator, too.

The example shows how to do it:

def powersOf2(n):

pow = 1

for i in range(n):

yield pow

pow \*= 2

for i in range(20):

if i in powersOf2(4):

print(i)

What's the code's output? Run the program and check.

Now let's see a **Fibonacci number generator**, and ensure that it looks much better than the objective version based on the direct iterator protocol implementation.

Here it is:

def Fib(n):

p = pp = 1

for i in range(n):

if i in [0, 1]:

yield 1

else:

n = p + pp

pp, p = p, n

yield n

fibs = list(Fib(10))

print(fibs)

Guess the output (a list) produced by the generator, and run the code to check if you were right.

def powersOf2(n):

pow = 1

for i in range(n):

yield pow

pow \*= 2

for v in powersOf2(8):

print(v)

6.1.7.6 Generators and closures

# More about list comprehensions

You should be able to remember the rules governing the creation and use of a very special Python phenomenon named **list comprehension - a simple and very impressive way of creating lists and their contents**.

In case you need it, we've provided a quick reminder in the editor.

There are two parts inside the code, both creating a list containing a few of the first natural powers of ten.

The former uses a routine way of utilizing the for loop, while the latter makes use of the list comprehension and builds the list in situ, without needing a loop, or any other extended code.

It looks like the list is created inside itself - it's not true, of course, as Python has to perform nearly the same operations as in the first snippet, but it is indisputable that the second formalism is simply more elegant, and lets the reader avoid any unnecessary details.

The example outputs two identical lines containing the following text:

[1, 10, 100, 1000, 10000, 100000]

Run the code to check if we're right.

listOne = []

for ex in range(6):

listOne.append(10 \*\* ex)

listTwo = [10 \*\* ex for ex in range(6)]

print(listOne)

print(listTwo)}

6.1.7.7 Generators and closures

# More about list comprehensions: continued

There is a very interesting syntax we want to show you now. Its usability is not limited to list comprehensions, but we have to admit that comprehensions are the ideal environment for it.

It's a **conditional expression - a way of selecting one of two different values based on the result of a Boolean expression**.

Look:

expression\_one if condition else expression\_two

It may look a bit surprising at first glance, but you have to keep in mind that it is **not a conditional instruction**. Moreover, it's not an instruction at all. It's an operator.

The value it provides is equal to expression\_one when the condition is True, and expression\_two otherwise.

A good example will tell you more. Look at the code in the editor.

The code fills a list with 1's and 0s - if the index of a particular element is odd, the element is set to 0, and to 1 otherwise.

Simple? Maybe not at first glance. Elegant? Indisputably.

Can you use the same trick within a list comprehension? Yes, you can.

lst = []

for x in range(10):

lst.append(1 if x % 2 == 0 else 0)

print(lst)

6.1.7.8 Generators and closures

# More about list comprehensions: continued

Look at the example in the editor.

Compactness and elegance - these two words come to mind when looking at the code.

So, what do they have in common, generators and list comprehensions? Is there any connection between them? Yes. A rather loose connection, but an unequivocal one.

Just one change can **turn any comprehension into a generator**.

Now look at the code below and see if you can find the detail that turns a list comprehension into a generator:

lst = [1 if x % 2 == 0 else 0 for x in range(10)]

genr = (1 if x % 2 == 0 else 0 for x in range(10))

for v in lst:

print(v, end=" ")

print()

for v in genr:

print(v, end=" ")

print()

It's the **parentheses**. The brackets make a comprehension, the parentheses make a generator.

The code, however, when run, produces two identical lines:

1 0 1 0 1 0 1 0 1 0

1 0 1 0 1 0 1 0 1 0

How can you know that the second assignment creates a generator, not a list?

There is some proof we can show you. Apply the len() function to both these entities.

len(lst) will evaluate to 10. Clear and predictable. len(genr) will raise an exception, and you will see the following message:

TypeError: object of type 'generator' has no len()

Of course, saving either the list or the generator is not necessary - you can create them exactly in the place where you need them - just like here:

for v in [1 if x % 2 == 0 else 0 for x in range(10)]:

print(v, end=" ")

print()

for v in (1 if x % 2 == 0 else 0 for x in range(10)):

print(v, end=" ")

print()

Note: the same appearance of the output doesn't mean that both loops work in the same way. In the first loop, the list is created (and iterated through) as a whole - it actually exists when the loop is being executed.

In the second loop, there is no list at all - there are only subsequent values produced by the generator, one by one.

Carry out your own experiments.

lst = [1 if x % 2 == 0 else 0 for x in range(10)]

print(lst)

6.1.7.9 Generators and closures

**The lambda function**

The lambda function is a concept borrowed from mathematics, more specifically, from a part called *the Lambda calculus*, but these two phenomena are not the same.

Mathematicians use *the Lambda calculus* in many formal systems connected with logic, recursion, or theorem provability. Programmers use the lambda function to simplify the code, to make it clearer and easier to understand.

A lambda function is a function without a name (you can also call it **an anonymous function**). Of course, such a statement immediately raises the question: how do you use anything that cannot be identified?

Fortunately, it's not a problem, as you can name such a function if you really need, but, in fact, in many cases the lambda function can exist and work while remaining fully incognito.

The declaration of the lambda function doesn't resemble a normal function declaration in any way - see for yourself:

lambda parameters : expression

Such a clause **returns the value of the expression when taking into account the current value of the current**lambda**argument**.

As usual, an example will be helpful. Our example uses three lambda functions, but gives them names. Look at it carefully:

two = lambda : 2

sqr = lambda x : x \* x

pwr = lambda x, y : x \*\* y

for a in range(-2, 3):

print(sqr(a), end=" ")

print(pwr(a, two()))

Let's analzye it:

* the first lambda is an anonymous **parameterless function** that always returns 2. As we've **assigned it to a variable named**two, we can say that the function is not anonymous anymore, and we can use the name to invoke it.

* the second one is a **one-parameter anonymous function** that returns the value of its squared argument. We've named it as such, too.

* the third lambda **takes two parameters** and returns the value of the first one raised to the power of the second one. The name of the variable which carries the lambda speaks for itself. We don't use pow to avoid confusion with the built-in function of the same name and the same purpose.

The program produces the following output:

4 4

1 1

0 0

1 1

4 4

This example is clear enough to show how lambdas are declared and how they behave, but it says nothing about why they're necessary, and what they're used for, since they can all be replaced with routine Python functions.

Where is the benefit?

6.1.7.10 Generators and closures

**How to use lambdas and what for?**

The most interesting part of using lambdas appears when you can use them in their pure form - **as anonymous parts of code intended to evaluate a result**.

Imagine that we need a function (we'll name it printfunction) which prints the values of a given (other) function for a set of selected arguments.

We want printfunction to be universal - it should accept a set of arguments put in a list and a function to be evaluated, both as arguments - we don't want to hardcode anything.

Look at the example in the editor. This is how we've implemented the idea.

Let's analyze it. The printfunction() function takes two parameters:

* the first, a list of arguments for which we want to print the results;
* the second, a function which should be invoked as many times as the number of values that are collected inside the first parameter.

Note: we've also defined a function named poly() - this is the function whose values we're going to print. The calculation the function performs isn't very sophisticated - it's the polynomial (hence its name) of a form:

f(x) = 2x2 - 4x + 2

The name of the function is then passed to the printfunction() along with a set of five different arguments - the set is built with a list comprehension clause.

The code prints the following lines:

f(-2)=18

f(-1)=8

f(0)=2

f(1)=0

f(2)=2

Can we avoid defining the poly() function, as we're not going to use it more than once? Yes, we can - this is the benefit a lambda can bring.

Look at the example below. Can you see the difference?

def printfunction(args, fun):

for x in args:

print('f(', x,')=', fun(x), sep='')

printfunction([x for x in range(-2, 3)], lambda x: 2 \* x\*\*2 - 4 \* x + 2)

The printfunction() has remained exactly the same, but there is no poly() function. We don't need it anymore, as the polynomial is now directly inside the printfunction() invocation in the form of a lambda defined in the following way: lambda x: 2 \* x\*\*2 - 4 \* x + 2.

The code has become shorter, clearer, and more legible.

Let us show you another place where lambdas can be useful. We'll start with a description of map(), a built-in Python function. Its name isn't too descriptive, its idea is simple, and the function itself is really usable.

def printfunction(args, fun):

for x in args:

print('f(', x,')=', fun(x), sep='')

def poly(x):

return 2 \* x\*\*2 - 4 \* x + 2

printfunction([x for x in range(-2, 3)], poly)

6.1.7.11 Generators and closures

# Lambdas and the map() function

In the simplest of all possible cases, the map() function takes two arguments:

* a function;
* a list.

map(function, list)

The above description is extremely simplified, as:

* the second map() argument may be any entity that can be iterated (e.g., a tuple, or just a generator)
* map() can accept more than two arguments.

The map()**function applies the function passed by its first argument to all its second argument's elements, and returns an iterator delivering all subsequent function results**. You can use the resulting iterator in a loop, or convert it into a list using the list() function.

Can you see a role for any lambda here?

Look at the code in the editor - we've used two lambdas in it.

This is the intrigue:

* build the list1 with values from 0 to 4;
* next, use map along with the first lambda to create a new list in which all elements have been evaluated as 2 raised to the power taken from the corresponding element from list1;
* list2 is printed then;
* in the next step, use the map() function again to make use of the generator it returns and to directly print all the values it delivers; as you can see, we've engaged the second lambda here - it just squares each element from list2.

Try to imagine the same code without lambdas. Would it be any better? It's unlikely.

list1 = [x for x in range(5)]

list2 = list(map(lambda x: 2 \*\* x, list1))

print(list2)

for x in map(lambda x: x \* x, list2):

print(x, end=' ')

print()

6.1.7.12 Generators and closures

**Lambdas and the filter() function**

Another Python function which can be significantly beautified by the application of a lambda is filter().

It expects the same kind of arguments as map(), but does something different - it **filters its second argument while being guided by directions flowing from the function specified as the first argument** (the function is invoked for each list element, just like in map()).

The elements which return True from the function **pass the filter** - the others are rejected.

The example in the editor shows the filter() function in action.

Note: we've made use of the random module to initialize the random number generator (not to be confused with the generators we've just talked about) with the seed() function, and to produce five random integer values from -10 to 10 using the randint() function.

The list is then filtered, and only the numbers which are even and greater than zero are accepted.

Of course, it's not likely that you'll receive the same results, but this is what our results looked like:

[6, 3, 3, 2, -7]

[6, 2]

from random import seed, randint

seed()

data = [ randint(-10,10) for x in range(5) ]

filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))

print(data)

print(filtered)

6.1.7.13 Generators and closures

# A brief look at closures

Let's start with a definition: **closure is a technique which allows the storing of values in spite of the fact that the context in which they have been created does not exist anymore**. Intricate? A bit.

Let's analyze a simple example:

def outer(par):

loc = par

var = 1

outer(var)

print(var)

print(loc)

The example is obviously erroneous.

The last two lines will cause a NameError exception - neither par nor loc is accessible outside the function. Both the variables exist when and only when the outer() function is being executed.

Look at the example in the editor. We've modified the code significantly.

There is a brand new element in it - a function (named inner) inside another function (named outer).

How does it work? Just like any other function except for the fact that inner() may be invoked only from within outer(). We can say that inner() is outer()'s private tool - no other part of the code can access it.

Look carefully:

* the inner() function returns the value of the variable accessible inside its scope, as inner() can use any of the entities at the disposal of outer()
* the outer() function returns the inner() function itself; more precisely, it returns a copy of the inner() function, the one which was frozen at the moment of outer()'s invocation; the frozen function contains its full environment, including the state of all local variables, which also means that the value of loc is successfully retained, although outer() ceased to exist a long time ago.

In effect, the code is fully valid, and outputs:

1

The function returned during the outer() invocation is a **closure**.

def outer(par):

loc = par

def inner():

return loc

return inner

var = 1

fun = outer(var)

print(fun())

6.1.7.14 Generators and closures

# A brief look at closures: continued

**A closure has to be invoked in exactly the same way in which it has been declared**.

In the previous example (see the code below):

def outer(par):

loc = par

def inner():

return loc

return inner

var = 1

fun = outer(var)

print(fun())

the inner() function was parameterless, so we had to invoke it without arguments.

Now look at the code in the editor. It is fully possible to **declare a closure equipped with an arbitrary number of parameters**, e.g., one, just like the power() function.

This means that the closure not only makes use of the frozen environment, but it can also **modify its behavior by using values taken from the outside**.

This example shows one more interesting circumstance - you can **create as many closures as you want using one and the same piece of code**. This is done with a function named makeclosure(). Note:

* the first closure obtained from makeclosure() defines a tool squaring its argument;
* the second one is designed to cube the argument.

This is why the code produces the following output:

0 0 0

1 1 1

2 4 8

3 9 27

4 16 64

Carry out your own tests.

def makeclosure(par):

loc = par

def power(p):

return p \*\* loc

return power

fsqr = makeclosure(2)

fcub = makeclosure(3)

for i in range(5):

print(i, fsqr(i), fcub(i))

6.1.8.1 Processing files

**Accessing files from Python code**

One of the most common issues in the developer's job is to **process data stored in files** while the files are usually physically stored using storage devices - hard, optical, network, or solid-state disks.

It's easy to imagine a program that sorts 20 numbers, and it's equally easy to imagine the user of this program entering these twenty numbers directly from the keyboard.

It's much harder to imagine the same task when there are 20,000 numbers to be sorted, and there isn't a single user who is able to enter these numbers without making a mistake.

It's much easier to imagine that these numbers are stored in the disk file which is read by the program. The program sorts the numbers and doesn't send them to the screen, but instead creates a new file and saves the sorted sequence of numbers there.

If we want to implement a simple database, the only way to store the information between program runs is to save it into a file (or files if your database is more complex).

In principle, any non-simple programming problem relies on the use of files, whether it processes images (stored in files), multiplies matrices (stored in files), or calculates wages and taxes (reading data stored in files).



You may ask why we have waited until now to show you these issues.

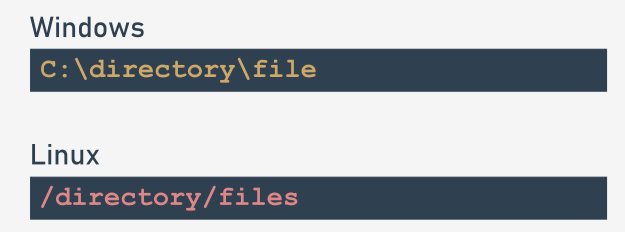
The answer is very simple - Python's way of accessing and processing files is implemented using a consistent set of objects. There is no better moment to talk about it.

6.1.8.2 Processing files

## File names

Different operating systems can treat the files in different ways. For example, Windows uses a different naming convention than the one adopted in Unix/Linux systems.

If we use the notion of a canonical file name (a name which uniquely defines the location of the file regardless of its level in the directory tree) we can realize that these names look different in Windows and in Unix/Linux:



As you can see, systems derived from Unix/Linux don't use the disk drive letter (e.g., C:) and all the directories grow from one root directory called /, while Windows systems recognize the root directory as \.

In addition, Unix/Linux system file names are case-sensitive. Windows systems store the case of letters used in the file name, but don't distinguish between their cases at all.

This means that these two strings:  
  
ThisIsTheNameOfTheFile  
and  
  
thisisthenameofthefile  
describe two different files in Unix/Linux systems, but are the same name for just one file in Windows systems.

The main and most striking difference is that you have to use **two different separators for the directory names**: \ in Windows, and / in Unix/Linux.

This difference is not very important to the normal user, but is **very important when writing programs in Python**.

To understand why, try to recall the very specific role played by the \ inside Python strings.

6.1.8.3 Processing files

## File names: continued

Suppose you're interested in a particular file located in the directory dir, and named file.

Suppose also that you want to assign a string containing the name of the file.

In Unix/Linux systems, it may look as follows:

name = "/dir/file"

But if you try to code it for the Windows system:

name = "\dir\file"

you'll get an unpleasant surprise: either Python will generate an error, or the execution of the program will behave strangely, as if the file name has been distorted in some way.

In fact, it's not strange at all, but quite obvious and natural. Python uses the \ as an escape character (like \n).

This means that Windows file names must be written as follows:

name = "\\dir\\file"

Fortunately, there is also one more solution. Python is smart enough to be able to convert slashes into backslashes each time it discovers that it's required by the OS.

This means that any the following assignments:

name = "/dir/file"

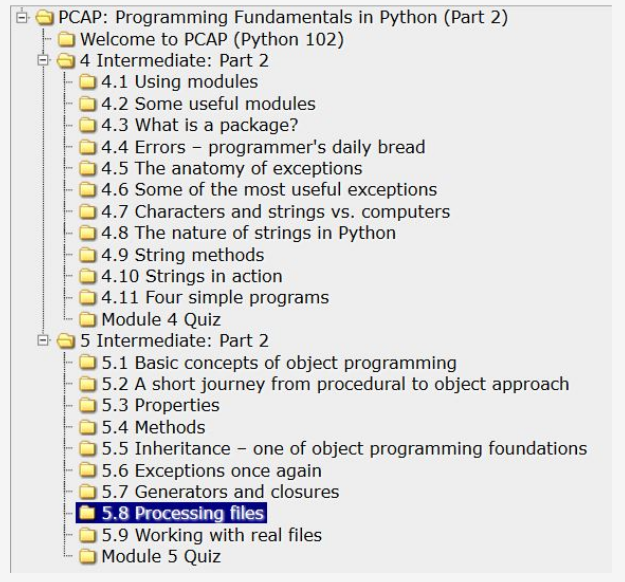
name = "c:/dir/file"

will work with Windows, too.

Any program written in Python (and not only in Python, because that convention applies to virtually all programming languages) does not communicate with the files directly, but through some abstract entities that are named differently in different languages or environments - the most-used terms are **handles** or **streams** (we'll use them as synonyms here).

The programmer, having a more- or less-rich set of functions/methods, is able to perform certain operations on the stream, which affect the real files using mechanisms contained in the operating system kernel.

In this way, you can implement the process of accessing any file, even when the name of the file is unknown at the time of writing the program.



The operations performed with the abstract stream reflect the activities related to the physical file.

To connect (bind) the stream with the file, it's necessary to perform an explicit operation.

The operation of connecting the stream with a file is called **opening the file**, while disconnecting this link is named **closing the file**.

Hence, the conclusion is that the very first operation performed on the stream is always open and the last one is close. The program, in effect, is free to manipulate the stream between these two events and to handle the associated file.

This freedom is limited, of course, by the physical characteristics of the file and the way in which the file has been opened.

Let us say again that the opening of the stream can fail, and it may happen due to several reasons: the most common is the lack of a file with a specified name.

It can also happen that the physical file exists, but the program is not allowed to open it. There's also the risk that the program has opened too many streams, and the specific operating system may not allow the simultaneous opening of more than n files (e.g., 200).

A well-written program should detect these failed openings, and react accordingly.

6.1.8.4 Processing files

## File streams

The opening of the stream is not only associated with the file, but should also declare the manner in which the stream will be processed. This declaration is called an **open mode**.

If the opening is successful, the **program will be allowed to perform only the operations which are consistent with the declared open mode**.

There are two basic operations performed on the stream:

* **read** from the stream: the portions of the data are retrieved from the file and placed in a memory area managed by the program (e.g., a variable);
* **write** to the stream: the portions of the data from the memory (e.g., a variable) are transferred to the file.

There are three basic modes used to open the stream:

* **read mode**: a stream opened in this mode allows **read operations only**; trying to write to the stream will cause an exception (the exception is named UnsupportedOperation, which inherits OSError and ValueError, and comes from the io module);
* **write mode**: a stream opened in this mode allows **write operations only**; attempting to read the stream will cause the exception mentioned above;
* **update mode**: a stream opened in this mode allows **both writes and reads**.

Before we discuss how to manipulate the streams, we owe you some explanation. **The stream behaves almost like a tape recorder**.

When you read something from a stream, a virtual head moves over the stream according to the number of bytes transferred from the stream.

When you write something to the stream, the same head moves along the stream recording the data from the memory.

Whenever we talk about reading from and writing to the stream, try to imagine this analogy. The programming books refer to this mechanism as the **current file position**, and we'll also use this term.



It's necessary now to show you the object responsible for representing streams in programs.

6.1.8.5 Processing files

## File handles

Python assumes that **every file is hidden behind an object of an adequate class**.

Of course, it's hard not to ask how to interpret the word *adequate*.

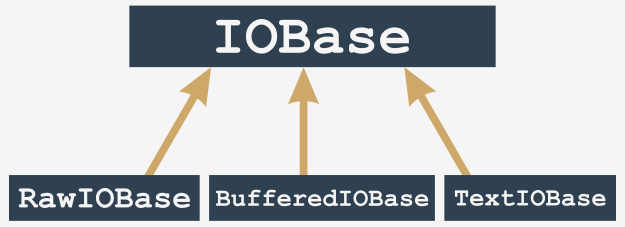
Files can be processed in many different ways - some of them depend on the file's contents, some on the programmer's intentions.

In any case, different files may require different sets of operations, and behave in different ways.

An object of an adequate class is **created when you open the file and annihilate it at the time of closing**.

Between these two events, you can use the object to specify what operations should be performed on a particular stream. The operations you're allowed to use are imposed by **the way in which you've opened the file**.

In general, the object comes from one of the classes shown here:



Note: you never use constructors to bring these objects to life. The only way you **obtain them is to invoke the function named**open().

The function analyses the arguments you've provided, and automatically creates the required object.

If you want to **get rid of the object, you invoke the method named**close().

The invocation will sever the connection to the object, and the file and will remove the object.

For our purposes, we'll concern ourselves only with streams represented by BufferIOBase and TextIOBase objects. You'll understand why soon.

6.1.8.6 Processing files

## File handles: continued

Due to the type of the stream's contents, **all the streams are divided into text and binary streams**.

The text streams ones are structured in lines; that is, they contain typographical characters (letters, digits, punctuation, etc.) arranged in rows (lines), as seen with the naked eye when you look at the contents of the file in the editor.

This file is written (or read) mostly character by character, or line by line.

The binary streams don't contain text but a sequence of bytes of any value. This sequence can be, for example, an executable program, an image, an audio or a video clip, a database file, etc.

Because these files don't contain lines, the reads and writes relate to portions of data of any size. Hence the data is read/written byte by byte, or block by block, where the size of the block usually ranges from one to an arbitrarily chosen value.

Then comes a subtle problem. In Unix/Linux systems, the line ends are marked by a single character named LF (ASCII code 10) designated in Python programs as \n.

Other operating systems, especially these derived from the prehistoric CP/M system (which applies to Windows family systems, too) use a different convention: the end of line is marked by a pair of characters, CR and LF (ASCII codes 13 and 10) which can be encoded as \r\n.



This ambiguity can cause various unpleasant consequences.

If you create a program responsible for processing a text file, and it is written for Windows, you can recognize the ends of the lines by finding the \r\n characters, but the same program running in a Unix/Linux environment will be completely useless, and vice versa: the program written for Unix/Linux systems might be useless in Windows.

Such undesirable features of the program, which prevent or hinder the use of the program in different environments, are called **non-portability**.

Similarly, the trait of the program allowing execution in different environments is called **portability**. A program endowed with such a trait is called a **portable program**.

6.1.8.7 Processing files

## File handles: continued

Since portability issues were (and still are) very serious, a decision was made to definitely resolve the issue in a way that doesn't engage the developer's attention.

It was done at the level of classes, which are responsible for reading and writing characters to and from the stream. It works in the following way:

* when the stream is open and it's advised that the data in the associated file will be processed as text (or there is no such advisory at all), it is **switched into text mode**;

* during reading/writing of lines from/to the associated file, nothing special occurs in the Unix environment, but when the same operations are performed in the Windows environment, a process called a **translation of newline characters** occurs: when you read a line from the file, every pair of \r\n characters is replaced with a single \n character, and vice versa; during write operations, every \n character is replaced with a pair of \r\n characters;

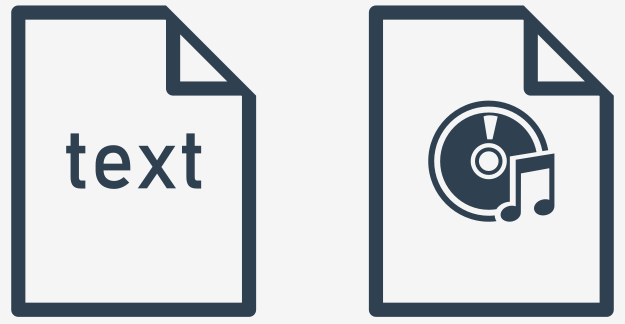
* the mechanism is completely **transparent** to the program, which can be written as if it was intended for processing Unix/Linux text files only; the source code run in a Windows environment will work properly, too;

* when the stream is open and it's advised to do so, its contents are taken as-is, **without any conversion** - no bytes are added or omitted.

## Opening the streams

The **opening of the stream** is performed by a function which can be invoked in the following way:

stream = open(file, mode = 'r', encoding = None)



Let's analyze it:

* the name of the function (open) speaks for itself; if the opening is successful, the function returns a stream object; otherwise, an exception is raised (e.g., FileNotFoundError **if the file you're going to read doesn't exist**);
* the first parameter of the function (file) specifies the name of the file to be associated with the stream;
* the second parameter (mode) specifies the open mode used for the stream; it's a string filled with a sequence of characters, and each of them has its own special meaning (more details soon);
* the third parameter (encoding) specifies the encoding type (e.g., UTF-8 when working with text files)
* the opening must be the very first operation performed on the stream.

Note: the mode and encoding arguments may be omitted - their default values are assumed then. The default opening mode is reading in text mode, while the default encoding depends on the platform used.

Let us now present you with the most important and useful open modes. Ready?

6.1.8.8 Processing files

## Opening the streams: modes

r open mode: read

* the stream will be opened in **read mode**;
* the file associated with the stream **must exist** and has to be readable, otherwise the open() function raises an exception.

w open mode: write

* the stream will be opened in **write mode**;
* the file associated with the stream **doesn't need to exist**; if it doesn't exist it will be created; if it exists, it will be truncated to the length of zero (erased); if the creation isn't possible (e.g., due to system permissions) the open() function raises an exception.

a open mode: append

* the stream will be opened in **append mode**;
* the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched.)

r+ open mode: read and update

* the stream will be opened in **read and update mode**;
* the file associated with the stream **must exist and has to be writeable**, otherwise the open() function raises an exception;
* both read and write operations are allowed for the stream.

w+ open mode: write and update

* the stream will be opened in **write and update** mode;
* the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; the previous content of the file remains untouched;
* both read and write operations are allowed for the stream.

## Selecting text and binary modes

If there is a letter b at the end of the mode string it means that the stream is to be opened in the **binary mode**.

If the mode string ends with a letter t the stream is opened in the **text mode**.

Text mode is the default behaviour assumed when no binary/text mode specifier is used.

Finally, the successful opening of the file will set the current file position (the virtual reading/writing head) before the first byte of the file **if the mode is not**a and after the last byte of file **if the mode is set to**a.

| **Text mode** | **Binary mode** | **Description** |
| --- | --- | --- |
| rt | rb | read |
|  |  |  |
| wt | wb | write |
| at | ab | append |
| r+t | r+b | read and update |
| w+t | w+b | write and update |

EXTRA

You can also open a file for its exclusive creation. You can do this using the x open mode. If the file already exists, the open() function will raise an exception.

6.1.8.9 Processing files

## Opening the stream for the first time

Imagine that we want to develop a program that reads content of the text file named: C:\Users\User\Desktop\file.txt.

How to open that file for reading? Here's the relevant snippet of the code:

try:

stream = open("C:\Users\User\Desktop\file.txt", "rt")

# processing goes here

stream.close()

except Exception as exc:

print("Cannot open the file:", exc)

What's going on here?

* we open the try-except block as we want to handle runtime errors softly;
* we use the open() function to try to open the specified file (note the way we've specified the file name)
* the open mode is defined as text to read (as **text is the default setting**, we can skip the t in mode string)
* in case of success we get an object from the open() function and we assign it to the stream variable;
* if open() fails, we handle the exception printing full error information (it's definitely good to know what exactly happened)

## Pre-opened streams

We said earlier that any stream operation must be preceded by the open() function invocation. There are three well-defined exceptions to the rule.

When our program starts, the three streams are already opened and don't require any extra preparations. What's more, your program can use these streams explicitly if you take care to import the sys module:

import sys

because that's where the declaration of the three streams is placed.

The names of these streams are: sys.stdin, sys.stdout, and sys.stderr.

Let's analyze them:

* sys.stdin
  + stdin (as *standard input*)
  + the stdin stream is normally associated with the keyboard, pre-open for reading and regarded as the primary data source for the running programs;
  + the well-known input() function reads data from stdin by default.

* sys.stdout
  + stdout (as *standard output*)
  + the stdout stream is normally associated with the screen, pre-open for writing, regarded as the primary target for outputting data by the running program;
  + the well-known print() function outputs the data to the stdout stream.

* sys.stderr
  + stderr (as *standard error output*)
  + the stderr stream is normally associated with the screen, pre-open for writing, regarded as the primary place where the running program should send information on the errors encountered during its work;
  + we haven't presented any method to send the data to this stream (we will do it soon, we promise)
  + the separation of stdout (useful results produced by the program) from the stderr (error messages, undeniably useful but does not provide results) gives the possibility of redirecting these two types of information to the different targets. More extensive discussion of this issue is beyond the scope of our course. The operation system handbook will provide more information on these issues.

6.1.8.10 Processing files

## Closing streams

The last operation performed on a stream (this doesn't include the stdin, stdout, and stderr streams which don't require it) should be **closing**.

That action is performed by a method invoked from within open stream object: stream.close().

* the name of the function is definitely self-commenting (close())
* the function expects exactly no arguments; the stream doesn't need to be opened
* the function returns nothing but raises IOError exception in case of error;
* most developers believe that the close() function always succeeds and thus there is no need to check if it's done its task properly.  
    
  This belief is only partly justified. If the stream was opened for writing and then a series of write operations were performed, it may happen that the data sent to the stream has not been transferred to the physical device yet (due to mechanism called **caching** or **buffering**). Since the closing of the stream forces the buffers to flush them, it may be that the flushes fail and therefore the close() fails too.

We have already mentioned failures caused by functions operating with streams but not mentioned a word how exactly we can identify the cause of the failure.

The possibility of making a diagnosis exists and is provided by one of streams' exception component which we are going to tell you about just now.

## Diagnosing stream problems

The IOError object is equipped with a property named errno (the name comes from the phrase *error number*) and you can access it as follows:

try:

# some stream operations

except IOError as exc:

print(exc.errno)

The value of the errno attribute can be compared with one of the predefined symbolic constants defined in the errno module.

Let's take a look at some selected **constants useful for detecting stream errors**:

errno.EACCES → Permission denied

The error occurs when you try, for example, to open a file with the *read only* attribute for writing.

errno.EBADF → Bad file number

The error occurs when you try, for example, to operate with an unopened stream.

errno.EEXIST → File exists

The error occurs when you try, for example, to rename a file with its previous name.

errno.EFBIG → File too large

The error occurs when you try to create a file that is larger than the maximum allowed by the operating system.

errno.EISDIR → Is a directory

The error occurs when you try to treat a directory name as the name of an ordinary file.

errno.EMFILE → Too many open files

The error occurs when you try to simultaneously open more streams than acceptable for your operating system.

errno.ENOENT → No such file or directory

The error occurs when you try to access a non-existent file/directory.

errno.ENOSPC → No space left on device

The error occurs when there is no free space on the media.

The complete list is much longer (it includes also some error codes not related to the stream processing.)

6.1.8.11 Processing files

**Diagnosing stream problems: continued**

If you are a very careful programmer, you may feel the need to use the sequence of statements similar to those presented below:

import errno

try:

s = open("c:/users/user/Desktop/file.txt", "rt")

# actual processing goes here

s.close()

except Exception as exc:

if exc.errno == errno.ENOENT:

print("The file doesn't exist.")

elif exc.errno == errno.EMFILE:

print("You've opened too many files.")

else:

printf("The error number is:", exc.errno)

Fortunately, there is a function that can dramatically **simplify the error handling code**. Its name is strerror(), and it comes from the os module and **expects just one argument - an error number**.

Its role is simple: you give an error number and get a string describing the meaning of the error.

Note: if you pass a non-existent error code (a number which is not bound to any actual error), the function will raise ValueError exception.

Now we can simplify our code in the following way:

from os import strerror

try:

s = open("c:/users/user/Desktop/file.txt", "rt")

# actual processing goes here

s.close()

except Exception as exc:

print("The file could not be opened:", strerror(exc.errno));

Okay. Now it's time to deal with text files and get familiar with some basic techniques you can use to process them.

6.1.9.1 Working with real files

**Processing text files**

In this lesson we're going to prepare a simple text file with some short, simple content.

We're going to show you some basic techniques you can utilize to **read the file contents** in order to process them.

The processing will be very simple - you're going to copy the file's contents to the console, and count all the characters the program has read in.

But remember - our understanding of a text file is very strict. In our sense, it's a plain text file - it may contain only text, without any additional decorations (formatting, different fonts, etc.).

That's why you should avoid creating the file using any advanced text processor like MS Word, LibreOffice Writer, or something like this. Use the very basics your OS offers: Notepad, vim, gedit, etc.

If your text files contain some national characters not covered by the standard ASCII charset, you may need an additional step. Your open() function invocation may require an argument denoting specific text encoding.

For example, if you're using a Unix/Linux OS configured to use UTF-8 as a system-wide setting, the open() function may look as follows:

stream = open('file.txt', 'rt', encoding='utf-8')

where the encoding argument has to be set to a value which is a string representing proper text encoding (UTF-8, here).

Consult your OS documentation to find an encoding name adequate to your environment.

INFORMATION

For the purposes of our experiments with file processing carried out in this section, we're going to use a pre-uploaded set of files (e.g., tzop.txt, or text.txt files) which you'll be able to work with. If you'd like to work with your own files locally on your machine, we strongly encourage you to do so, and to use IDLE to carry out your own tests.

stream = open("tzop.txt", "rt", encoding = "utf-8") # opening tzop.txt in read mode, returning it as a file object

print(stream.read()) # printing the content of the file

6.1.9.2 Working with real files

# Processing text files: continued

Reading a text file's contents can be performed using several different methods - none of them is any better or worse than any other. It's up to you which of them you prefer and like.

Some of them will sometimes be handier, and sometimes more troublesome. Be flexible. Don't be afraid to change your preferences.

The most basic of these methods is the one offered by the read() function, which you were able to see in action in the previous lesson.

If applied to a text file, the function is able to:

* read a desired number of characters (including just one) from the file, and return them as a string;
* read all the file contents, and return them as a string;
* if there is nothing more to read (the virtual reading head reaches the end of the file), the function returns an empty string.

We'll start with the simplest variant and use a file named text.txt. The file has the following contents:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Now look at the code in the editor, and let's analyze it.

The routine is rather simple:

* use the try-except mechanism and open the file of the predetermined name (text.txt in our case)
* try to read the very first character from the file (ch = s.read(1))
* if you succeed (this is proven by a positive result of the while condition check), output the character (note the end= argument - it's important! You don't want to skip to a new line after every character!);
* update the counter (cnt), too;
* try to read the next character, and the process repeats.

from os import strerror

try:

cnt = 0

s = open('text.txt', "rt")

ch = s.read(1)

while ch != '':

print(ch, end='')

cnt += 1

ch = s.read(1)

s.close()

print("\n\nCharacters in file:", cnt)

except IOError as e:

print("I/O error occurred: ", strerr(e.errno))

6.1.9.3 Working with real files

# Processing text files: continued

If you're absolutely sure that the file's length is safe and you can read the whole file to the memory at once, you can do it - the read() function, invoked without any arguments or with an argument that evaluates to None, will do the job for you.

Remember - **reading a terabyte-long file using this method may corrupt your OS**.

Don't expect miracles - computer memory isn't stretchable.

Look at the code in the editor. What do you think of it?

Let's analyze it:

* open the file as previously;
* read its contents by one read() function invocation;
* next, process the text, iterating through it with a regular for loop, and updating the counter value at each turn of the loop;

The result will be exactly the same as previously.

from os import strerror

try:

cnt = 0

s = open('text.txt', "rt")

content = s.read()

for ch in content:

print(ch, end='')

cnt += 1

ch = s.read(1)

s.close()

print("\n\nCharacters in file:", cnt)

except IOError as e:

print("I/O error occurred: ", strerr(e.errno))

6.1.9.4 Working with real files

**Processing text files: readline()**

If you want to treat the file's contents **as a set of lines**, not a bunch of characters, the readline() method will help you with that.

The method tries to **read a complete line of text from the file**, and returns it as a string in the case of success. Otherwise, it returns an empty string.

This opens up new opportunities - now you can also count lines easily, not only characters.

Let's make use of it. Look at the code in the editor.

As you can see, the general idea is exactly the same as in both previous examples.

from os import strerror

try:

ccnt = lcnt = 0

s = open('text.txt', 'rt')

line = s.readline()

while line != '':

lcnt += 1

for ch in line:

print(ch, end='')

ccnt += 1

line = s.readline()

s.close()

print("\n\nCharacters in file:", ccnt)

print("Lines in file: ", lcnt)

except IOError as e:

print("I/O error occurred:", strerr(e.errno))

6.1.9.5 Working with real files

# Processing text files: readlines()

Another method, which treats text file as a set of lines, not characters, is readlines().

The readlines() method, when invoked without arguments, tries to **read all the file contents, and returns a list of strings, one element per file line**.

If you're not sure if the file size is small enough and don't want to test the OS, you can convince the readlines() method to read not more than a specified number of bytes at once (the returning value remains the same - it's a list of a string).

Feel free to experiment with [this example code](https://edube.org/sandbox/3d223dbe-143c-11e9-b35b-02426ea0318a) to understand how the readlines() method works.

**The maximum accepted input buffer size is passed to the method as its argument**.

You may expect that readlines() can process a file's contents more effectively than readline(), as it may need to be invoked fewer times.

Note: when there is nothing to read from the file, the method returns an empty list. Use it to detect the end of the file.

To the extent of the buffer's size, you can expect that increasing it may improve input performance, but there is no golden rule for it - try to find the optimal values yourself.

Look at the code in the editor. We've modified it to show you how to use readlines().

We've decided to use a 15-byte-long buffer. Don't think it's a recommendation.

We've used such a value to avoid the situation in which the first readlines() invocation consumes the whole file.

We want the method to be forced to work harder, and to demonstrate its capabilities.

There are **two nested loops in the code**: the outer one uses readlines()'s result to iterate through it, while the inner one prints the lines character by character.

from os import strerror

try:

ccnt = lcnt = 0

s = open('text.txt', 'rt')

lines = s.readlines(20)

while len(lines) != 0:

for line in lines:

lcnt += 1

for ch in line:

print(ch, end='')

ccnt += 1

lines = s.readlines(10)

s.close()

print("\n\nCharacters in file:", ccnt)

print("Lines in file: ", lcnt)

except IOError as e:

print("I/O error occurred:", strerr(e.errno))

6.1.9.6 Working with real files

**Processing text files: continued**

The last example we want to present shows a very interesting trait of the object returned by the open() function in text mode.

We think it may surprise you - **the object is an instance of the iterable class**.

Strange? Not at all. Usable? Yes, absolutely.

The **iteration protocol defined for the file object** is very simple - its \_\_next\_\_ method just **returns the next line read in from the file**.

Moreover, you can expect that the object automatically invokes close() when any of the file reads reaches the end of the file.

Look at the editor and see how simple and clear the code has now become.

from os import strerror

try:

ccnt = lcnt = 0

for line in open('text.txt', 'rt'):

lcnt += 1

for ch in line:

print(ch, end='')

ccnt += 1

print("\n\nCharacters in file:", ccnt)

print("Lines in file: ", lcnt)

except IOError as e:

print("I/O error occurred: ", strerr(e.errno))

6.1.9.7 Working with real files

**Dealing with text files: write()**

Writing text files seems to be simpler, as in fact there is one method that can be used to perform such a task.

p>The method is named write() and it expects just one argument - a string that will be transferred to an open file (don't forget - the open mode should reflect the way in which the data is transferred - **writing a file opened in read mode won't succeed**).

No newline character is added to the write()'s argument, so you have to add it yourself if you want the file to be filled with a number of lines.

The example in the editor shows a very simple code that creates a file named newtext.txt (note: the open mode w ensures that **the file will be created from scratch**, even if it exists and contains data) and then puts ten lines into it.

The string to be recorded consists of the word line, followed by the line number. We've decided to write the string's contents character by character (this is done by the inner for loop) but you're not obliged to do it in this way.

We just wanted to show you that write() is able to operate on single characters.

The code creates a file filled with the following text:

line #1

line #2

line #3

line #4

line #5

line #6

line #7

line #8

line #9

line #10

We encourage you to test the behavior of the write() method locally on your machine.

from os import strerror

try:

fo = open('newtext.txt', 'wt') # a new file (newtext.txt) is created

for i in range(10):

s = "line #" + str(i+1) + "\n"

for ch in s:

fo.write(ch)

fo.close()

except IOError as e:

print("I/O error occurred: ", strerr(e.errno))

6.1.9.8 Working with real files

**Dealing with text files: continued**

Look at the example in the editor. We've modified the previous code to write whole lines to the text file.

The contents of the newly created file are the same.

Note: you can use the same method to write to the stderr stream, but don't try to open it, as it's always open implicitly.

For example, if you want to send a message string to stderr to distinguish it from normal program output, it may look like this:

import sys

sys.stderr.write("Error message")

from os import strerror

try:

fo = open('newtext.txt', 'wt')

for i in range(10):

fo.write("line #" + str(i+1) + "\n")

fo.close()

except IOError as e:

print("I/O error occurred: ", strerr(e.errno))

6.1.9.9 Working with real files

## What is a bytearray?

Before we start talking about binary files, we have to tell you about one of the **specialized classes Python uses to store amorphous data**.

**Amorphous data is data which have no specific shape or form** - they are just a series of bytes.

This doesn't mean that these bytes cannot have their own meaning, or cannot represent any useful object, e.g., bitmap graphics.

The most important aspect of this is that in the place where we have contact with the data, we are not able to, or simply don't want to, know anything about it.

Amorphous data cannot be stored using any of the previously presented means - they are neither strings nor lists.

There should be a special container able to handle such data.

Python has more than one such container - one of them is **a specialized class name bytearray** - as the name suggests, it's **an array containing (amorphous) bytes**.

If you want to have such a container, e.g., in order to read in a bitmap image and process it in any way, you need to create it explicitly, using one of available constructors.

Take a look:

data = bytearray(10)

Such an invocation creates a bytearray object able to store ten bytes.

Note: such a constructor **fills the whole array with zeros**.

6.1.9.10 Working with real files

**Bytearrays: continued**

Bytearrays resemble lists in many respects. For example, they are **mutable**, they're a subject of the len() function, and you can access any of their elements using conventional indexing.

There is one important limitation - **you mustn't set any byte array elements with a value which is not an integer** (violating this rule will cause a TypeError exception) and you're **not allowed to assign a value that doesn't come from the range 0 to 255 inclusive** (unless you want to provoke a ValueError exception).

You can **treat any byte array elements as integer values** - just like in the example in the editor.

Note: we've used two methods to iterate the byte arrays, and made use of the hex() function to see the elements printed as hexadecimal values.

Now we're going to show you **how to write a byte array to a binary file** - binary, as we don't want to save its readable representation - we want to write a one-to-one copy of the physical memory content, byte by byte.

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 - i

for b in data:

print(hex(b))

6.1.9.11 Working with real files

# Bytearrays: continued

So, how do we write a byte array to a binary file?

Look at the code in the editor. Let's analyze it:

* first, we initialize bytearray with subsequent values starting from 10; if you want the file's contents to be clearly readable, replace 10 with something like ord('a') - this will produce bytes containing values corresponding to the alphabetical part of the ASCII code (don't think it will make the file a text file - it's still binary, as it was created with a wb flag);
* then, we create the file using the open() function - the only difference compared to the previous variants is the open mode containing the b flag;
* the write() method takes its argument (bytearray) and sends it (as a whole) to the file;
* the stream is then closed in a routine way.

The write() method returns a number of successfully written bytes.

If the values differ from the length of the method's arguments, it may announce some write errors.

In this case, we haven't made use of the result - this may not be appropriate in every case.

Try to run the code and analyze the contents of the newly created output file.

You're going to use it in the next step.

## How to read bytes from a stream

Reading from a binary file requires use of a specialized method name readinto(), as the method doesn't create a new byte array object, but fills a previously created one with the values taken from the binary file.

Note:

* the method returns the number of successfully read bytes;
* the method tries to fill the whole space available inside its argument; if there are more data in the file than space in the argument, the read operation will stop before the end of the file; otherwise, the method's result may indicate that the byte array has only been filled fragmentarily (the result will show you that, too, and the part of the array not being used by the newly read contents remains untouched)

Look at the complete code below:

from os import strerror

data = bytearray(10)

try:

bf = open('file.bin', 'rb')

bf.readinto(data)

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerr(e.errno))

Let's analyze it:

* first, we open the file (the one you created using the previous code) with the mode described as rb;
* then, we read its contents into the byte array named data, of size ten bytes;
* finally, we print the byte array contents - are they the same as you expected?

Run the code and check if it's working.

from os import strerror

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 + i

try:

bf = open('file.bin', 'wb')

bf.write(data)

bf.close()

except IOError as e:

print("I/O error occurred:", strerr(e.errno))

# enter code that reads bytes from the stream here

6.1.9.12 Working with real files

**How to read bytes from a stream**

An alternative way of reading the contents of a binary file is offered by the method named read().

Invoked without arguments, it tries to **read all the contents of the file into the memory**, making them a part of a newly created object of the bytes class.

This class has some similarities to bytearray, with the exception of one significant difference - it's **immutable**.

Fortunately, there are no obstacles to creating a byte array by taking its initial value directly from the bytes object, just like here:

from os import strerror

try:

bf = open('file.bin', 'rb')

data = bytearray(bf.read())

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerr(e.errno))

Be careful - **don't use this kind of read if you're not sure that the file's contents will fit the available memory**.

from os import strerror

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 + i

try:

bf = open('file.bin', 'wb')

bf.write(data)

bf.close()

except IOError as e:

print("I/O error occurred:", strerr(e.errno))

# enter code that reads bytes from the stream here

6.1.9.13 Working with real files

**How to read bytes from a stream: continued**

If the read() method is invoked with an argument, it **specifies the maximum number of bytes to be read**.

The method tries to read the desired number of bytes from the file, and the length of the returned object can be used to determine the number of bytes actually read.

You can use the method just like here:

try:

bf = open('file.bin', 'rb')

data = bytearray(bf.read(5))

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerr(e.errno))

Note: the first five bytes of the file have been read by the code - the next five are still waiting to be processed.

from os import strerror

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 + i

try:

bf = open('file.bin', 'wb')

bf.write(data)

bf.close()

except IOError as e:

print("I/O error occurred:", strerr(e.errno))

# enter code that reads bytes from the stream here

6.1.9.14 Working with real files

# Copying files - a simple and functional tool

Now you're going to amalgamate all this new knowledge, add some fresh elements to it, and use it to write a real code which is able to actually copy a file's contents.

Of course, the purpose is not to make a better replacement for commands like *copy* (MS Windows) or *cp* (Unix/Linux) but to see one possible way of creating a working tool, even if nobody wants to use it.

Look at the code in the editor. Let's analyze it:

* lines 3 through 8: ask the user for the name of the file to copy, and try to open it to read; terminate the program execution if the open fails; note: use the exit() function to stop program execution and to pass the completion code to the OS; any completion code other than 0 says that the program has encountered some problems; use the errno value to specify the nature of the issue;
* lines 9 through 15: repeat nearly the same action, but this time for the output file;
* line 17: prepare a piece of memory for transferring data from the source file to the target one; such a transfer area is often called a buffer, hence the name of the variable; the size of the buffer is arbitrary - in this case, we decided to use 64 kilobytes; technically, a larger buffer is faster at copying items, as a larger buffer means fewer I/O operations; actually, there is always a limit, the crossing of which renders no further improvements; test it yourself if you want.
* line 18: count the bytes copied - this is the counter and its initial value;
* line 20: try to fill the buffer for the very first time;
* line 21: as long as you get a non-zero number of bytes, repeat the same actions;
* line 22: write the buffer's contents to the output file (note: we've used a slice to limit the number of bytes being written, as write() always prefer to write the whole buffer)
* line 23: update the counter;
* line 24: read the next file chunk;
* lines 29 through 31: some final cleaning - the job is done.

from os import strerror

srcname = input("Source file name?: ")

try:

src = open(srcname, 'rb')

except IOError as e:

print("Cannot open source file: ", strerror(e.errno))

exit(e.errno)

dstname = input("Destination file name?: ")

try:

dst = open(dstname, 'wb')

except Exception as e:

print("Cannot create destination file: ", strerr(e.errno))

src.close()

exit(e.errno)

buffer = bytearray(65536)

total = 0

try:

readin = src.readinto(buffer)

while readin > 0:

written = dst.write(buffer[:readin])

total += written

readin = src.readinto(buffer)

except IOError as e:

print("Cannot create destination file: ", strerr(e.errno))

exit(e.errno)

print(total,'byte(s) succesfully written')

src.close()

dst.close()

**LAB**

6.1.9.15 LAB: Character frequency histogram

## Estimated time

30 minutes

## Level of difficulty

Medium

## Objectives

* improving the student's skills in operating with files (reading)
* using data collections for counting numerous data.

## Scenario

A text file contains some text (nothing unusual) but we need to know how often (or how rare) each letter appears in the text. Such an analysis may be useful in cryptography, so we want to be able to do that in reference to the Latin alphabet.

Your task is to write a program which:

* asks the user for the input file's name;
* reads the file (if possible) and counts all the Latin letters (lower- and upper-case letters are treated as equal)
* prints a simple histogram in alphabetical order (only non-zero counts should be presented)

Create a test file for the code, and check if your histogram contains valid results.

Assuming that the test file contains just one line filled with:

aBc

the expected output should look as follows:a -> 1

b -> 1

c -> 1

**Tip**:

We think that a dictionary is a perfect data collection medium for storing the counts. The letters may be keys while the counters can be values.

**LAB**

6.1.9.16 LAB: Sorted character frequency histogram

## Estimated time

15-20 minutes

## Level of difficulty

Medium

## Prerequisites

05\_9.15.1

## Objectives

* improve the student's skills in operating with files (reading/writing)
* using lambdas to change the sort order.

## Scenario

The previous code needs to be improved. It's okay, but it has to be better.

Your task is to make some amendments, which generate the following results:

* the output histogram will be sorted based on the characters' frequency (the bigger counter should be presented first)
* the histogram should be sent to a file with the same name as the input one, but with the suffix '.hist' (it should be concatenated to the original name)

Assuming that the input file contains just one line filled with:

cBabAa

the expected output should look as follows:a -> 3

b -> 2

c -> 1

**Tip**:

Use a lambda to change the sort order.

**LAB**

6.1.9.17 LAB: Evaluating students' results

## Estimated time

30 minutes

## Level of difficulty

Medium

## Objectives

* improve the student's skills in operating with files (reading)
* perfecting the student's abilities in defining and using self-defined exceptions and dictionaries.

## Scenario

Prof. Jekyll conducts classes with students and regularly makes notes in a text file. Each line of the file contains 3 elements: the student's first name, the student's last name, and the number of point the student received during certain classes.

The elements are separated with white spaces. Each student may appear more than once inside Prof. Jekyll's file.

The file may look as follows:

John Smith 5

Anna Boleyn 4.5

John Smith 2

Anna Boleyn 11

Andrew Cox 1.5

Your task is to write a program which:

* asks the user for Prof. Jekyll's file name;
* reads the file contents and counts the sum of the received points for each student;
* prints a simple (but sorted) report, just like this one:

Andrew Cox 1.5

Anna Boleyn 15.5

John Smith 7.0

Note:

* your program must be fully protected against all possible failures: the file's non-existence, the file's emptiness, or any input data failures; encountering any data error should cause immediate program termination, and the erroneous should be presented to the user;
* implement and use your own exceptions hierarchy - we've presented it in the editor; the second exception should be raised when a bad line is detect, and the third when the source file exists but is empty.

**Tip**:

Use a dictionary to store the students' data.

class StudentsDataException(Exception):

pass

class BadLine(StudentsDataException):

# put your code here

class FileEmpty(StudentsDataException):

# put your code here