**Python Essentials 1:  
Module 4**

**Functions, Tuples, Dictionaries, Exceptions, and Data Processing**

In this module, you will cover the following topics:

* code structuring and the concept of function;
* function invocation and returning a result from a function;
* name scopes and variable shadowing;
* tuples and their purpose, constructing and using tuples;
* dictionaries and their purpose, constructing and using dictionaries;
* exceptions – the *try* statement and the *except* clause, Python built-in exceptions, code testing and debugging.

# Why do we need functions?

You've come across **functions** many times so far, but the view on their merits that we have given you has been rather one-sided. You've only invoked the functions by using them as tools to make life easier, and to simplify time-consuming and tedious tasks.

When you want some data to be printed on the console, you use print(). When you want to read the value of a variable, you use input(), coupled with either int() or float().

You've also made use of some **methods**, which are in fact functions, but declared in a very specific way.

Now you'll learn how to write your own functions, and how to use them. We'll write several functions together, from the very simple to the rather complex, which will require your focus and attention.

It often happens that a particular piece of code is **repeated many times in your program**. It's repeated either literally, or with only a few minor modifications, consisting of the use of other variables in the same algorithm. It also happens that a programmer cannot resist simplifying the work, and begins to clone such pieces of code using the clipboard and copy-paste operations.

It could end up as greatly frustrating when suddenly it turns out that there was an error in the cloned code. The programmer will have a lot of drudgery to find all the places that need corrections. There's also a high risk of the corrections causing errors.

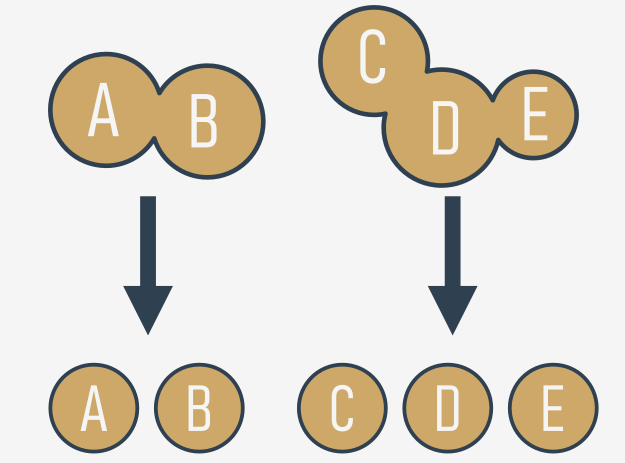
We can now define the first condition which can help you decide when to start writing your own functions: **if a particular fragment of the code begins to appear in more than one place, consider the possibility of isolating it in the form of a function** invoked from the points where the original code was placed before.

It may happen that the algorithm you're going to implement is so complex that your code begins to grow in an uncontrolled manner, and suddenly you notice that you're not able to navigate through it so easily anymore.

You can try to cope with the issue by commenting the code extensively, but soon you find that this dramatically worsens your situation - **too many comments make the code larger and harder to read**. Some say that a **well-written function should be viewed entirely in one glance**.

A good and attentive developer **divides the code** (or more accurately: the problem) into well-isolated pieces, and **encodes each of them in the form of a function**.

This considerably simplifies the work of the program, because each piece of code can be encoded separately, and tested separately. The process described here is often called **decomposition**.

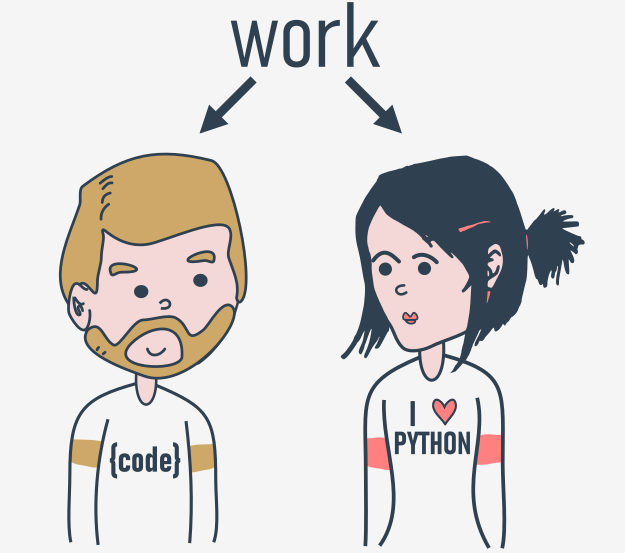


We can now state the second condition: **if a piece of code becomes so large that reading and understating it may cause a problem, consider dividing it into separate, smaller problems, and implement each of them in the form of a separate function**.

This decomposition continues until you get a set of short functions, easy to understand and test.

# Decomposition

It often happens that the problem is so large and complex that it cannot be assigned to a single developer, and a **team of developers** have to work on it. The problem must be split between several developers in a way that ensures their efficient and seamless cooperation.



It seems inconceivable that more than one programmer should write the same piece of code at the same time, so the job has to be dispersed among all the team members.

This kind of decomposition has a different purpose to the one described previously - it's not only about **sharing the work**, but also about **sharing the responsibility** among many developers.

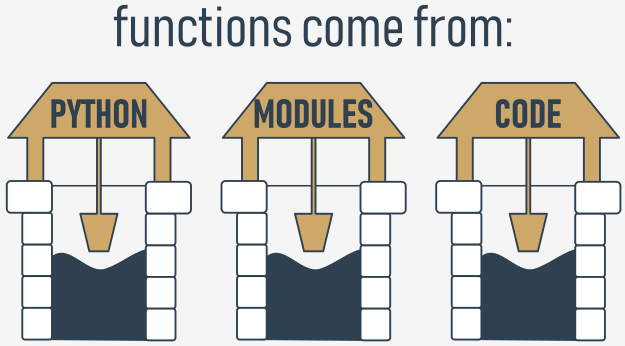
Each of them writes a clearly defined and described set of functions, which when **combined into the module** (we'll tell you about this a bit later) will give the final product.

This leads us directly to the third condition: if you're going to divide the work among multiple programmers, **decompose the problem to allow the product to be implemented as a set of separately written functions packed together in different modules**.

## Where do the functions come from?

In general, functions come from at least three places:

* from Python itself - numerous functions (like print()) are an **integral part of Python**, and are always available without any additional effort on behalf of the programmer; we call these functions **built-in functions**;
* from Python's **preinstalled modules** - a lot of functions, very useful ones, but used significantly less often than built-in ones, are available in a number of modules installed together with Python; the use of these functions requires some additional steps from the programmer in order to make them fully accessible (we'll tell you about this in a while);
* **directly from your code** - you can write your own functions, place them inside your code, and use them freely;
* there is one other possibility, but it's connected with classes, so we'll omit it for now.



# Your first function

Take a look at the snippet in the editor.

It's rather simple, but we only want it to be an example of **transforming a repeating part of a code into a function**.

The messages sent to the console by the print() function are always the same. Of course, there's nothing really bad in such a code, but try to imagine what you would have to do if your boss asked you to change the message to make it more polite, e.g., to start it with the phrase "Please,".

It seems that you'd have to spend some time changing all the occurrences of the message (you'd use a clipboard, of course, but it wouldn't make your life much easier). It's obvious that you'd probably make some mistakes during the amendment process, and you (and your boss) would get a bit frustrated.

Is it possible to separate such a *repeatable* part of the code, name it and make it reusable? It would mean that a **change made once in one place would be propagated to all the places where it's used**.

Of course, such a code should work only when it's explicitly launched.

Yes, it's possible. This is exactly what functions are for.

print("Enter a value: ")

a = int(input())

print("Enter a value: ")

b = int(input())

print("Enter a value: ")

c = int(input())

**Your first function**

How do you make such a function?

You need to **define** it. The word *define* is significant here.

This is what the simplest function definition looks like:

def function\_name():

function\_body

* It always starts with the **keyword**def (for *define*)
* next after def goes the **name of the function** (the rules for naming functions are exactly the same as for naming variables)
* after the function name, there's a place for a pair of **parentheses** (they contain nothing here, but that will change soon)
* the line has to be ended with a **colon**
* the line directly after def begins the **function body** ‒ a couple (at least one) of necessarily **nested instructions**, which will be executed every time the function is invoked; note: the **function ends where the nesting ends**, so you have to be careful.

We're ready to define our **prompting** function. We'll name it message ‒ here it is:

def message():

print("Enter a value: ")

The function is extremely simple, but fully **usable**. We've named it message, but you can label it according to your taste. Let's use it.

Our code contains the function definition now:

def message():

print("Enter a value: ")

print("We start here.")

print("We end here.")

Note: we don't use the function at all ‒ there's no **invocation** of it inside the code.

When you run it, you see the following output:

We start here.

We end here.

**output**

This means that Python reads the function's definitions and remembers them, but won't launch any of them without your permission.

We've modified the code now ‒ we've inserted the **function's invocation** between the start and end messages:

def message():

print("Enter a value: ")

print("We start here.")

message()

print("We end here.")

The output looks different now:

We start here.

Enter a value:

We end here.

**output**

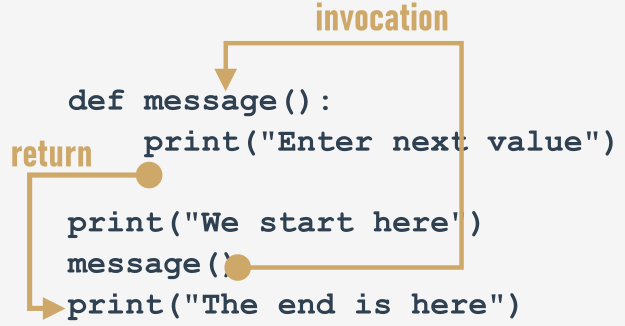
Test the code, modify it, experiment with it.

def my\_function():

# function body

**How functions work**

Look at the picture below:



It tries to show you the whole process:

* when you **invoke** a function, Python remembers the place where it happened and *jumps* into the invoked function;
* the body of the function is then **executed**;
* reaching the end of the function forces Python to **return** to the place directly after the point of invocation.

There are two, very important, catches. Here's the first of them:

**You mustn't invoke a function which is not known at the moment of invocation.**

Remember - Python reads your code from top to bottom. It's not going to look ahead in order to find a function you forgot to put in the right place ("right" means "before invocation".)

We've inserted an error into this code - can you see the difference?

print("We start here.")

message()

print("We end here.")

def message():

print("Enter a value: ")

We've moved the function to the end of the code. Is Python able to find it when the execution reaches the invocation?

No, it isn't. The error message will read:

NameError: name 'message' is not defined

**output**

Don't try to force Python to look for functions you didn't deliver at the right time.

The second catch sounds a little simpler:

**You mustn't have a function and a variable of the same name**.

The following snippet is erroneous:

def message():

print("Enter a value: ")

message = 1

Assigning a value to the name message causes Python to forget its previous role. The function named message becomes unavailable.

Fortunately, you're free to **mix your code with functions** - you're not obliged to put all your functions at the top of your source file.

Look at the snippet:

print("We start here.")

def message():

print("Enter a value: ")

message()

print("We end here.")

It may look strange, but it's completely correct, and works as intended.

Let's return to our primary example, and employ the function for the right job, like here:

def message():

print("Enter a value: ")

message()

a = int(input())

message()

b = int(input())

message()

c = int(input())

Modifying the prompting message is now easy and clear - you can do it by **changing the code in just one place** - inside the function's body.

Open the sandbox, and try to do it yourself.

**Key takeaways**

1. A **function** is a block of code that performs a specific task when the function is called (invoked). You can use functions to make your code reusable, better organized, and more readable. Functions can have parameters and return values.

2. There are at least four basic types of functions in Python:

* **built-in functions** which are an integral part of Python (such as the print() function). You can see a complete list of Python built-in functions at <https://docs.python.org/3/library/functions.html>.
* the ones that come from **pre-installed modules** (you'll learn about them in the *Python Essentials 2* course)
* **user-defined functions** which are written by users for users - you can write your own functions and use them freely in your code,
* the lambda functions (you'll learn about them in the *Python Essentials 2* course.)

3. You can define your own function using the def keyword and the following syntax:

def your\_function(optional parameters):

# the body of the function

You can define a function which doesn't take any arguments, e.g.:

def message(): # defining a function

print("Hello") # body of the function

message() # calling the function

You can define a function which takes arguments, too, just like the one-parameter function below:

def hello(name): # defining a function

print("Hello,", name) # body of the function

name = input("Enter your name: ")

hello(name) # calling the function

We'll tell you more about parametrized functions in the next section. Don't worry.

**Exercise 1**

The input() function is an example of a:  
  
a) user-defined function  
b) built-in function

Check

**b** - it's a built-in function

**Exercise 2**

What happens when you try to invoke a function before you define it? Example:

hi()

def hi():

print("hi!")

Check

An exception is thrown (the NameError exception to be more precise)

**Exercise 3**

What will happen when you run the code below?

def hi():

print("hi")

hi(5)

Check

An exception will be thrown (the TypeError exception to be more precise) - the hi() function doesn't take any arguments

**Parameterized functions**

The function's full power reveals itself when it can be equipped with an interface that is able to accept data provided by the invoker. Such data can modify the function's behavior, making it more flexible and adaptable to changing conditions.

A parameter is actually a variable, but there are two important factors that make parameters different and special:

* **parameters exist only inside functions in which they have been defined**, and the only place where the parameter can be defined is a space between a pair of parentheses in the def statement;
* **assigning a value to the parameter is done at the time of the function's invocation**, by specifying the corresponding argument.

def function(parameter):

###

Don't forget:

* **parameters live inside functions** (this is their natural environment)
* **arguments exist outside functions**, and are carriers of values passed to corresponding parameters.

There is a clear and unambiguous frontier between these two worlds.

Let's enrich the function above with just one parameter - we're going to use it to show the user the number of a value the function asks for.

We have to rebuild the def statement - this is how it looks now:

def message(number):

###

The definition specifies that our function operates on just one parameter named number. You can use it as an ordinary variable, but **only inside the function** - it isn't visible anywhere else.

Let's now improve the function's body:

def message(number):

print("Enter a number:", number)

We've made use of the parameter. Note: we haven't assigned the parameter with any value. Is it correct?

Yes, it is.

A value for the parameter will arrive from the function's environment.

Remember: **specifying one or more parameters in a function's definition** is also a requirement, and you have to fulfil it during invocation. You must **provide as many arguments as there are defined parameters**.

Failure to do so will cause an error.

**Parameterized functions: continued**

Try to run the code in the editor.

This is what you'll see in the console:

TypeError: message() missing 1 required positional argument: 'number'

**output**

This looks better, for sure:

def message(number):

print("Enter a number:", number)

message(1)

Moreover, it behaves better. The code will produce the following output:

Enter a number: 1

**output**

Can you see how it works? The value of the argument used during invocation (1) has been passed into the function, setting the initial value of the parameter named number.

We have to make you sensitive to one important circumstance.

It's legal, and possible, to have a **variable named the same as a function's parameter**.

The snippet illustrates the phenomenon:

def message(number):

print("Enter a number:", number)

number = 1234

message(1)

print(number)

A situation like this activates a mechanism called **shadowing**:

* parameter x shadows any variable of the same name, but...
* ... only inside the function defining the parameter.

The parameter named number is a completely different entity from the variable named number.

This means that the snippet above will produce the following output:

Enter a number: 1

1234

**output**

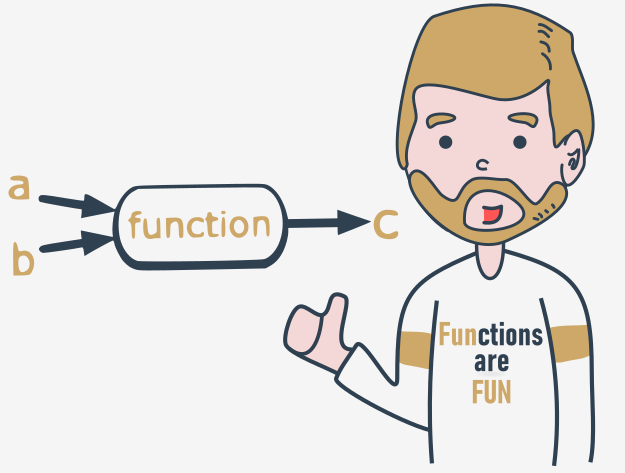
def message(number):

print("Enter a number:", number)

message()

# Parametrized functions: continued

A function can have **as many parameters as you want**, but the more parameters you have, the harder it is to memorize their roles and purposes.



Let's modify the function - it has **two parameters** now:

def message(what, number):

print("Enter", what, "number", number)

This also means that invoking the function will require **two arguments**.

The first new parameter is intended to carry the name of the desired value.

Here it is:

def message(what, number):

print("Enter", what, "number", number)

message("telephone", 11)

message("price", 5)

message("number", "number")

This is the output you're about to see:

Enter telephone number 11

Enter price number 5

Enter number number number

**output**

Run the code, modify it, add more parameters, and see how this affects the output.

def message(what, number):

print("Enter", what, "number", number)

# invoke the function

# Positional parameter passing

A technique which assigns the ith (first, second, and so on) argument to the ith (first, second, and so on) function parameter is called **positional parameter passing**, while arguments passed in this way are named **positional arguments**.

You've used it already, but Python can offer a lot more. We're going to tell you about it now.

def my\_function(a, b, c):

print(a, b, c)

my\_function(1, 2, 3)

Note: positional parameter passing is intuitively used by people in many social occasions. For example, it may be generally accepted that when we introduce ourselves we mention our first name(s) before our last name, e.g., "My name's John Doe."

Incidentally, Hungarians do it in reverse order.

Let's implement that social custom in Python. The following function will be responsible for introducing somebody:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction("Luke", "Skywalker")

introduction("Jesse", "Quick")

introduction("Clark", "Kent")

Can you guess the output? Run the code and find out if you were right.

Now imagine that the same function is being used in Hungary. In this case, the code would look like this:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction("Skywalker", "Luke")

introduction("Quick", "Jesse")

introduction("Kent", "Clark")

The output will look different. Can you guess it?

Run the code to see if you were right here, too. Are you surprised?

Can you make the function more culture-independent?

# Positional parameter passing

A technique which assigns the ith (first, second, and so on) argument to the ith (first, second, and so on) function parameter is called **positional parameter passing**, while arguments passed in this way are named **positional arguments**.

You've used it already, but Python can offer a lot more. We're going to tell you about it now.

def my\_function(a, b, c):

print(a, b, c)

my\_function(1, 2, 3)

Note: positional parameter passing is intuitively used by people in many social occasions. For example, it may be generally accepted that when we introduce ourselves we mention our first name(s) before our last name, e.g., "My name's John Doe."

Incidentally, Hungarians do it in reverse order.

Let's implement that social custom in Python. The following function will be responsible for introducing somebody:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction("Luke", "Skywalker")

introduction("Jesse", "Quick")

introduction("Clark", "Kent")

Can you guess the output? Run the code and find out if you were right.

Now imagine that the same function is being used in Hungary. In this case, the code would look like this:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction("Skywalker", "Luke")

introduction("Quick", "Jesse")

introduction("Kent", "Clark")

The output will look different. Can you guess it?

Run the code to see if you were right here, too. Are you surprised?

Can you make the function more culture-independent?

# Keyword argument passing

Python offers another convention for passing arguments, where **the meaning of the argument is dictated by its name**, not by its position - it's called **keyword argument passing**.

Take a look at the snippet:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction(first\_name = "James", last\_name = "Bond")

introduction(last\_name = "Skywalker", first\_name = "Luke")

The concept is clear - the values passed to the parameters are preceded by the target parameters' names, followed by the = sign.

The position doesn't matter here - each argument's value knows its destination on the basis of the name used.

You should be able to predict the output. Run the code to check if you were right.

Of course, you **mustn't use a non-existent parameter name**.

The following snippet will cause a runtime error:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction(surname="Skywalker", first\_name="Luke")

This is what Python will tell you:

TypeError: introduction() got an unexpected keyword argument 'surname'

**output**

Try it out yourself.

**Mixing positional and keyword arguments**

You can mix both fashions if you want - there is only one unbreakable rule: you have to put **positional arguments before keyword arguments**.

If you think for a moment, you'll certainly guess why.

To show you how it works, we'll use the following simple three-parameter function:

def adding(a, b, c):

print(a, "+", b, "+", c, "=", a + b + c)

Its purpose is to evaluate and present the sum of all its arguments.

The function, when invoked in the following way:

adding(1, 2, 3)

will output:

1 + 2 + 3 = 6

**output**

It was - as you may suspect - a pure example of **positional argument passing**.

Of course, you can replace such an invocation with a purely keyword variant, like this:

adding(c = 1, a = 2, b = 3)

Our program will output a line like this:

2 + 3 + 1 = 6

**output**

Note the order of the values.

Let's try to mix both styles now.

Look at the function invocation below:

adding(3, c = 1, b = 2)

Let's analyze it:

* the argument (3) for the a parameter is passed using the positional way;
* the arguments for c and b are specified as keyword ones.

This is what you'll see in the console:

3 + 2 + 1 = 6

**output**

Be careful, and beware of mistakes. If you try to pass more than one value to one argument, all you'll get is a runtime error.

Look at the invocation below - it seems that we've tried to set a twice:

adding(3, a = 1, b = 2)

Python's response:

TypeError: adding() got multiple values for argument 'a'

**output**

Look at the snipet below. A code like this is fully correct, but it doesn't make much sense:

adding(4, 3, c = 2)

Everything is right, but leaving in just one keyword argument looks a bit weird - what do you think?

def adding(a, b, c):

print(a, "+", b, "+", c, "=", a + b + c)

# Call the adding function here.

# Parametrized functions - more details

It happens at times that a particular parameter's values are in use more often than others. Such arguments may have their **default (predefined) values** taken into consideration when their corresponding arguments have been omitted.

They say that the most popular English last name is *Smith*. Let's try to take this into account.

The default parameter's value is set using clear and pictorial syntax:

def introduction(first\_name, last\_name="Smith"):

print("Hello, my name is", first\_name, last\_name)

You only have to extend the parameter's name with the = sign, followed by the default value.

Let's invoke the function as usual:

introduction("James", "Doe")

Can you guess the output of the program? Run it and check if you were right.

And? Everything looks the same, but when you invoke the function in a way that looks a bit suspicious at first sight, like this:

introduction("Henry")

or this:

introduction(first\_name="William")

there will be no error, and both invocations will succeed, while the console will show the following output:

Hello, my name is Henry Smith

Hello, my name is William Smith

**output**

Test it.

You can go further if it's useful. Both parameters have their default values now, look at the code below:

def introduction(first\_name="John", last\_name="Smith"):

print("Hello, my name is", first\_name, last\_name)

This makes the following invocation absolutely valid:

introduction()

And this is the expected output:

Hello, my name is John Smith

**output**

If you use one keyword argument, the remaining one will take the default value:

introduction(last\_name="Hopkins")

The output is:

Hello, my name is John Hopkins

**output**

Test it.

Congratulations - you have just learned the basic ways of communicating with functions.

def introduction(first\_name, last\_name="Smith"):

print("Hello, my name is", first\_name, last\_name)

# Call the function here.

**Key takeaways**

1. You can pass information to functions by using parameters. Your functions can have as many parameters as you need.

An example of a one-parameter function:

def hi(name):

print("Hi,", name)

hi("Greg")

An example of a two-parameter function:

def hi\_all(name\_1, name\_2):

print("Hi,", name\_2)

print("Hi,", name\_1)

hi\_all("Sebastian", "Konrad")

An example of a three-parameter function:

def address(street, city, postal\_code):

print("Your address is:", street, "St.,", city, postal\_code)

s = input("Street: ")

p\_c = input("Postal Code: ")

c = input("City: ")

address(s, c, p\_c)

2. You can pass arguments to a function using the following techniques:

* **positional argument passing** in which the order of arguments passed matters (Ex. 1),
* **keyword (named) argument passing** in which the order of arguments passed doesn't matter (Ex. 2),
* a mix of positional and keyword argument passing (Ex. 3).

Ex. 1

def subtra(a, b):

print(a - b)

subtra(5, 2) # outputs: 3

subtra(2, 5) # outputs: -3

Ex. 2

def subtra(a, b):

print(a - b)

subtra(a=5, b=2) # outputs: 3

subtra(b=2, a=5) # outputs: 3

Ex. 3

def subtra(a, b):

print(a - b)

subtra(5, b=2) # outputs: 3

subtra(5, 2) # outputs: 3

It's important to remember that **positional arguments mustn't follow keyword arguments**. That's why if you try to run the following snippet:

def subtra(a, b):

print(a - b)

subtra(5, b=2) # outputs: 3

subtra(a=5, 2) # Syntax Error

Python will not let you do it by signalling a SyntaxError.

3. You can use the keyword argument passing technique to **pre-define** a value for a given argument:

def name(first\_name, last\_name="Smith"):

print(first\_name, last\_name)

name("Andy") # outputs: Andy Smith

name("Betty", "Johnson") # outputs: Betty Johnson (the keyword argument replaced by "Johnson")

**Exercise 1**

What is the output of the following snippet?

def intro(a="James Bond", b="Bond"):

print("My name is", b + ".", a + ".")

intro()

Check

My name is Bond. James Bond.

**Exercise 2**

What is the output of the following snippet?

def intro(a="James Bond", b="Bond"):

print("My name is", b + ".", a + ".")

intro(b="Sean Connery")

Check

My name is Sean Connery. James Bond.

**Exercise 3**

What is the output of the following snippet?

def intro(a, b="Bond"):

print("My name is", b + ".", a + ".")

intro("Susan")

Check

My name is Bond. Susan.

**Exercise 4**

What is the output of the following snippet?

def add\_numbers(a, b=2, c):

print(a + b + c)

add\_numbers(a=1, c=3)

Check

SyntaxError - a non-default argument (c) follows a default argument (b=2)

# Effects and results: the return instruction

All the previously presented functions have some kind of effect - they produce some text and send it to the console.

Of course, functions - like their mathematical siblings - may have results.

To get **functions to return a value** (but not only for this purpose) you use the return instruction.

This word gives you a full picture of its capabilities. Note: it's a Python **keyword**.

The return instruction has **two different variants** - let's consider them separately.

## return without an expression

The first consists of the keyword itself, without anything following it.

When used inside a function, it causes the **immediate termination of the function's execution, and an instant return (hence the name) to the point of invocation**.

Note: if a function is not intended to produce a result, **using the**return**instruction is not obligatory** - it will be executed implicitly at the end of the function.

Anyway, you can use it to **terminate a function's activities on demand**, before the control reaches the function's last line.

Let's consider the following function:

def happy\_new\_year(wishes = True):

print("Three...")

print("Two...")

print("One...")

if not wishes:

return

print("Happy New Year!")

When invoked without any arguments:

happy\_new\_year()

The function causes a little noise - the output will look like this:

Three...

Two...

One...

Happy New Year!

**output**

Providing False as an argument:

happy\_new\_year(False)

will modify the function's behavior - the return instruction will cause its termination just before the wishes - this is the updated output:

Three...

Two...

One...

**output**

## return with an expression

The second return variant is **extended with an expression**:

def function():

return expression

There are two consequences of using it:

* it causes the **immediate termination of the function's execution** (nothing new compared to the first variant)
* moreover, the function will **evaluate the expression's value and will return (hence the name once again) it as the function's result**.

Yes, we already know - this example isn't really sophisticated:

def boring\_function():

return 123

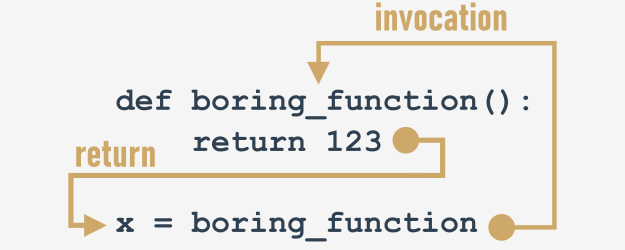
x = boring\_function()

print("The boring\_function has returned its result. It's:", x)

The snippet writes the following text to the console:

The boring\_function has returned its result. It's: 123

Let's investigate it for a while.

Analyze the figure below: 

The return instruction, enriched with the expression (the expression is very simple here), "transports" the expression's value to the place where the function has been invoked.

The result may be freely used here, e.g., to be assigned to a variable.

It may also be completely ignored and lost without a trace.

Note, we're not being too polite here - the function returns a value, and we ignore it (we don't use it in any way):

def boring\_function():

print("'Boredom Mode' ON.")

return 123

print("This lesson is interesting!")

boring\_function()

print("This lesson is boring...")

The program produces the following output:

This lesson is interesting!

'Boredom Mode' ON.

This lesson is boring...

**output**

Is it punishable? Not at all.

The only disadvantage is that the result has been irretrievably lost.

Don't forget:

* you are always **allowed to ignore the function's result**, and be satisfied with the function's effect (if the function has any)
* if a function is intended to return a useful result, it must contain the second variant of the return instruction.

Wait a minute - does this mean that there are useless results, too? Yes - in some sense.

**A few words about None**

Let us introduce you to a very curious value (to be honest, a none value) named None.

Its data doesn't represent any reasonable value - actually, it's not a value at all; hence, it **mustn't take part in any expressions**.

For example, a snippet like this:

print(None + 2)

will cause a runtime error, described by the following diagnostic message:

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

**output**

Note: None is a **keyword**.

There are only two kinds of circumstances when None can be safely used:

* when you **assign it to a variable** (or return it as a**function's result**)
* when you **compare it with a variable** to diagnose its internal state.

Just like here:

value = None

if value is None:

print("Sorry, you don't carry any value")

Don't forget this: if a function doesn't return a certain value using a return expression clause, it is assumed that it **implicitly returns**None.

Let's test it.

# A few words about None: continued

Take a look at the code in the editor.

It's obvious that the strangeFunction function returns True when its argument is even.

What does it return otherwise?

We can use the following code to check it:

print(strange\_function(2))

print(strange\_function(1))

This is what we see in the console:

True

None

**output**

Don't be surprised next time you see None as a function result - it may be the symptom of a subtle mistake inside the function.

def strange\_function(n):

if(n % 2 == 0):

return True

# Effects and results: lists and functions

There are two additional questions that should be answered here.

The first is: **may a list be sent to a function as an argument?**

Of course it may! Any entity recognizable by Python can play the role of a function argument, although it has to be assured that the function is able to cope with it.

So, if you pass a list to a function, the function has to handle it like a list.

A function like this one here:

def list\_sum(lst):

s = 0

for elem in lst:

s += elem

return s

and invoked like this:

print(list\_sum([5, 4, 3]))

will return 12 as a result, but you should expect problems if you invoke it in this risky way:

print(list\_sum(5))

Python's response will be unequivocal:

TypeError: 'int' object is not iterable

**output**

This is caused by the fact that a **single integer value mustn't be iterated through by the**for**loop**.

def list\_sum(lst):

s = 0

for elem in lst:

s += elem

return s

# Effects and results: lists and functions - continued

The second question is: **may a list be a function result?**

Yes, of course! Any entity recognizable by Python can be a function result.

Look at the code in the editor. The program's output will be like this:

[4, 3, 2, 1, 0]

**output**

Now you can write functions with and without results.

Let's dive a little deeper into the issues connected with variables in functions. This is essential for creating effective and safe functions.

def strange\_list\_fun(n):

strange\_list = []

for i in range(0, n):

strange\_list.insert(0, i)

return strange\_list

print(strange\_list\_fun(5))

**LAB**

## Estimated time

10-15 minutes

## Level of difficulty

Easy

## Objectives

Familiarize the student with:

* projecting and writing parameterized functions;
* utilizing the return statement;
* testing the functions.

## Scenario

Your task is to write and test a function which takes one argument (a year) and returns True if the year is a *leap year*, or False otherwise.

The seed of the function is already shown in the skeleton code in the editor.

Note: we've also prepared a short testing code, which you can use to test your function.

The code uses two lists ‒ one with the test data, and the other containing the expected results. The code will tell you if any of your results are invalid.

def is\_year\_leap(year):

#

# Write your code here.

#

test\_data = [1900, 2000, 2016, 1987]

test\_results = [False, True, True, False]

for i in range(len(test\_data)):

yr = test\_data[i]

print(yr,"->",end="")

result = is\_year\_leap(yr)

if result == test\_results[i]:

print("OK")

else:

print("Failed")

**LAB**

## Estimated time

15-20 minutes

## Level of difficulty

Medium

## Prerequisites

LAB 4.3.1.6

## Objectives

Familiarize the student with:

* projecting and writing parameterized functions;
* utilizing the return statement;
* utilizing the student's own functions.

## Scenario

Your task is to write and test a function which takes two arguments (a year and a month) and returns the number of days for the given month/year pair (while only February is sensitive to the year value, your function should be universal).

The initial part of the function is ready. Now, convince the function to return None if its arguments don't make sense.

Of course, you can (and should) use the previously written and tested function (LAB 4.3.1.6). It may be very helpful. We encourage you to use a list filled with the months' lengths. You can create it inside the function - this trick will significantly shorten the code.

We've prepared a testing code. Expand it to include more test cases.

def is\_year\_leap(year):

#

# Your code from LAB 4.3.1.6.

#

def days\_in\_month(year, month):

#

# Write your new code here.

#

test\_years = [1900, 2000, 2016, 1987]

test\_months = [2, 2, 1, 11]

test\_results = [28, 29, 31, 30]

for i in range(len(test\_years)):

yr = test\_years[i]

mo = test\_months[i]

print(yr, mo, "->", end="")

result = days\_in\_month(yr, mo)

if result == test\_results[i]:

print("OK")

else:

print("Failed")

**LAB**

## Estimated time

20-30 minutes

## Level of difficulty

Medium

## Prerequisites

LAB 4.3.1.6  
LAB 4.3.1.7

## Objectives

Familiarize the student with:

* projecting and writing parameterized functions;
* utilizing the return statement;
* building a set of utility functions;
* utilizing the student's own functions.

## Scenario

Your task is to write and test a function which takes three arguments (a year, a month, and a day of the month) and returns the corresponding day of the year, or returns None if any of the arguments is invalid.

Use the previously written and tested functions. Add some test cases to the code. This test is only a beginning.

def is\_year\_leap(year):

#

# Your code from LAB 4.3.1.6.

#

def days\_in\_month(year, month):

#

# Your code from LAB 4.3.1.7.

#

def day\_of\_year(year, month, day):

#

# Write your new code here.

#

print(day\_of\_year(2000, 12, 31))

**LAB**

## Estimated time

15-20 minutes

## Level of difficulty

Medium

## Objectives

* familiarizing the student with classic notions and algorithms;
* improving the student's skills in defining and using functions.

## Scenario

*A natural number is****prime****if it is greater than 1 and has no divisors other than 1 and itself.*

Complicated? Not at all. For example, 8 isn't a prime number, as you can divide it by 2 and 4 (we can't use divisors equal to 1 and 8, as the definition prohibits this).

On the other hand, 7 is a prime number, as we can't find any legal divisors for it.

Your task is to write a function checking whether a number is prime or not.

The function:

* is called is\_prime;
* takes one argument (the value to check)
* returns True if the argument is a prime number, and False otherwise.

Hint: try to divide the argument by all subsequent values (starting from 2) and check the remainder - if it's zero, your number cannot be a prime; think carefully about when you should stop the process.

If you need to know the square root of any value, you can utilize the \*\* operator. Remember: the square root of x is the same as x0.5

Complete the code in the editor.

Run your code and check whether your output is the same as ours.

## Expected output

2 3 5 7 11 13 17 19

def is\_prime(num):

#

# Write your code here.

#

for i in range(1, 20):

if is\_prime(i + 1):

print(i + 1, end=" ")

print()

**LAB**

## Estimated time

10-15 minutes

## Level of difficulty

Easy

## Objectives

* improving the student's skills in defining, using and testing functions.

## Scenario

A car's fuel consumption may be expressed in many different ways. For example, in Europe, it is shown as the amount of fuel consumed per 100 kilometers.

In the USA, it is shown as the number of miles traveled by a car using one gallon of fuel.

Your task is to write a pair of functions converting l/100km into mpg, and vice versa.

The functions:

* are named liters\_100km\_to\_miles\_gallon and miles\_gallon\_to\_liters\_100km respectively;
* take one argument (the value corresponding to their names)

Complete the code in the editor.

Run your code and check whether your output is the same as ours.

Here is some information to help you:

* 1 American mile = 1609.344 metres;
* 1 American gallon = 3.785411784 litres.

## Expected output

60.31143162393162

31.36194444444444

23.52145833333333

3.9007393587617467

7.490910297239916

10.009131205673757

def liters\_100km\_to\_miles\_gallon(liters):

#

# Write your code here.

#

def miles\_gallon\_to\_liters\_100km(miles):

#

# Write your code here

#

print(liters\_100km\_to\_miles\_gallon(3.9))

print(liters\_100km\_to\_miles\_gallon(7.5))

print(liters\_100km\_to\_miles\_gallon(10.))

print(miles\_gallon\_to\_liters\_100km(60.3))

print(miles\_gallon\_to\_liters\_100km(31.4))

print(miles\_gallon\_to\_liters\_100km(23.5))

# Key takeaways

1. You can use the return keyword to tell a function to return some value. The return statement exits the function, e.g.:

def multiply(a, b):

return a \* b

print(multiply(3, 4)) # outputs: 12

def multiply(a, b):

return

print(multiply(3, 4)) # outputs: None

2. The result of a function can be easily assigned to a variable, e.g.:

def wishes():

return "Happy Birthday!"

w = wishes()

print(w) # outputs: Happy Birthday!

Look at the difference in output in the following two examples:

# Example 1

def wishes():

print("My Wishes")

return "Happy Birthday"

wishes() # outputs: My Wishes

# Example 2

def wishes():

print("My Wishes")

return "Happy Birthday"

print(wishes())

# outputs: My Wishes

# Happy Birthday

3. You can use a list as a function's argument, e.g.:

def hi\_everybody(my\_list):

for name in my\_list:

print("Hi,", name)

hi\_everybody(["Adam", "John", "Lucy"])

4. A list can be a function result, too, e.g.:

def create\_list(n):

my\_list = []

for i in range(n):

my\_list.append(i)

return my\_list

print(create\_list(5))

**Exercise 1**

What is the output of the following snippet?

def hi():

return

print("Hi!")

hi()

Check

the function will return an implicit None value

**Exercise 2**

What is the output of the following snippet?

def is\_int(data):

if type(data) == int:

return True

elif type(data) == float:

return False

print(is\_int(5))

print(is\_int(5.0))

print(is\_int("5"))

Check

True

False

None

**Exercise 3**

What is the output of the following snippet?

def even\_num\_lst(ran):

lst = []

for num in range(ran):

if num % 2 == 0:

lst.append(num)

return lst

print(even\_num\_lst(11))

Check

[0, 2, 4, 6, 8, 10]

**Exercise 4**

What is the output of the following snippet?

def list\_updater(lst):

upd\_list = []

for elem in lst:

elem \*\*= 2

upd\_list.append(elem)

return upd\_list

foo = [1, 2, 3, 4, 5]

print(list\_updater(foo))

Check

[1, 4, 9, 16, 25]

# Functions and scopes

Let's start with a definition:

The **scope of a name** (e.g., a variable name) is the part of a code where the name is properly recognizable.

For example, the scope of a function's parameter is the function itself. The parameter is inaccessible outside the function.

Let's check it. Look at the code in the editor. What will happen when you run it?

The program will fail when run. The error message will read:

NameError: name 'x' is not defined

**output**

def scope\_test():

x = 123

scope\_test()

print(x)

# Functions and scopes: continued

Let's start by checking whether or not a variable created outside any function is visible inside the functions. In other words, does a variable's name propagate into a function's body?

Look at the code in the editor. Our guinea pig is there.

The result of the test is positive - the code outputs:

Do I know that variable? 1

1

**output**

The answer is: **a variable existing outside a function has a scope inside the functions' bodies**.

This rule has a very important exception. Let's try to find it.

def my\_function():

print("Do I know that variable?", var)

var = 1

my\_function()

print(var)

# Functions and scopes: the global keyword

Hopefully, you should now have arrived at the following question: does this mean that a function is not able to modify a variable defined outside it? This would create a lot of discomfort.

Fortunately, the answer is *no*.

There's a special Python method which can **extend a variable's scope in a way which includes the functions' bodies** (even if you want not only to read the values, but also to modify them).

Such an effect is caused by a keyword named global:

global name

global name1, name2, ...

Using this keyword inside a function with the name (or names separated with commas) of a variable(s), forces Python to refrain from creating a new variable inside the function - the one accessible from outside will be used instead.

In other words, this name becomes global (it has a **global scope**, and it doesn't matter whether it's the subject of read or assign).

Look at the code in the editor.

We've added global to the function.

The code now outputs:

Do I know that variable? 2

2

**output**

This should be sufficient evidence to show that the global keyword does what it promises.

def my\_function():

global var

var = 2

print("Do I know that variable?", var)

var = 1

my\_function()

print(var)

# How the function interacts with its arguments

Now let's find out how the function interacts with its arguments.

The code in the editor should teach you something. As you can see, the function changes the value of its parameter. Does the change affect the argument?

Run the program and check.

The code's output is:

I got 1

I have 2

1

**output**

The conclusion is obvious - **changing the parameter's value doesn't propagate outside the function** (in any case, not when the variable is a scalar, like in the example).

This also means that a function receives the **argument's value**, not the argument itself. This is true for scalars.

Is it worth checking how it works with lists (do you recall the peculiarities of assigning list slices versus assigning lists as a whole?).

The following example will shed some light on the issue:

def my\_function(my\_list\_1):

print("Print #1:", my\_list\_1)

print("Print #2:", my\_list\_2)

my\_list\_1 = [0, 1]

print("Print #3:", my\_list\_1)

print("Print #4:", my\_list\_2)

my\_list\_2 = [2, 3]

my\_function(my\_list\_2)

print("Print #5:", my\_list\_2)

The code's output is:

Print #1: [2, 3]

Print #2: [2, 3]

Print #3: [0, 1]

Print #4: [2, 3]

Print #5: [2, 3]

**output**

It seems that the former rule still works.

Finally, can you see the difference in the example below:

def my\_function(my\_list\_1):

print("Print #1:", my\_list\_1)

print("Print #2:", my\_list\_2)

del my\_list\_1[0] # Pay attention to this line.

print("Print #3:", my\_list\_1)

print("Print #4:", my\_list\_2)

my\_list\_2 = [2, 3]

my\_function(my\_list\_2)

print("Print #5:", my\_list\_2)

We don't change the value of the parameter my\_list\_1 (we already know it will not affect the argument), but instead modify the list identified by it.

The output may be surprising. Run the code and check:

Print #1: [2, 3]

Print #2: [2, 3]

Print #3: [3]

Print #4: [3]

Print #5: [3]

**output**

Can you explain it?

Let's try:

* if the argument is a list, then changing the value of the corresponding parameter doesn't affect the list (remember: variables containing lists are stored in a different way than scalars),
* but if you change a list identified by the parameter (note: the list, not the parameter!), the list will reflect the change.

It's time to write some example functions. You'll do that in the next section.

def my\_function(n):

print("I got", n)

n += 1

print("I have", n)

var = 1

my\_function(var)

print(var)

# Key takeaways

1. A variable that exists outside a function has a scope inside the function body (Example 1) unless the function defines a variable of the same name (Example 2, and Example 3), e.g.:

Example 1:

var = 2

def mult\_by\_var(x):

return x \* var

print(mult\_by\_var(7)) # outputs: 14

Example 2:

def mult(x):

var = 5

return x \* var

print(mult(7)) # outputs: 35

Example 3:

def mult(x):

var = 7

return x \* var

var = 3

print(mult(7)) # outputs: 49

2. A variable that exists inside a function has a scope inside the function body (Example 4), e.g.:

Example 4:

def adding(x):

var = 7

return x + var

print(adding(4)) # outputs: 11

print(var) # NameError

3. You can use the global keyword followed by a variable name to make the variable's scope global, e.g.:

var = 2

print(var) # outputs: 2

def return\_var():

global var

var = 5

return var

print(return\_var()) # outputs: 5

print(var) # outputs: 5

**Exercise 1**

What will happen when you try to run the following code?

def message():

alt = 1

print("Hello, World!")

print(alt)

Check

The NameError exception will be thrown (NameError: name 'alt' is not defined)

**Exercise 2**

What is the output of the following snippet?

a = 1

def fun():

a = 2

print(a)

fun()

print(a)

Check

2

1

**Exercise 3**

What is the output of the following snippet?

a = 1

def fun():

global a

a = 2

print(a)

fun()

a = 3

print(a)

Check

2

3

**Exercise 4**

What is the output of the following snippet?

a = 1

def fun():

global a

a = 2

print(a)

a = 3

fun()

print(a)

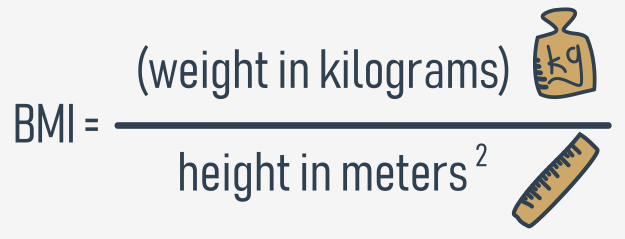
Check

2

2

# Some simple functions: evaluating the BMI

Let's get started on a function to evaluate the Body Mass Index (BMI).



As you can see, the formula gets two values:

* weight (originally in kilograms)
* height (originally in meters)

It seems that this new function will have **two parameters**. Its name will be bmi, but if you prefer any other name, use it instead.

Let's code the function:

def bmi(weight, height):

return weight / height \*\* 2

print(bmi(52.5, 1.65))

The result produced by the sample invocation looks as follows:

19.283746556473833

**output**

The function fulfils our expectations, but it's a bit simple - it assumes that the values of both parameters are always meaningful. It's definitely worth checking if they're trustworthy.

Let's check them both and return None if any of them looks suspicious.

def bmi(weight, height):

return weight / height \*\* 2

print(bmi(52.5, 1.65))

**Some simple functions: evaluating BMI and converting imperial units to metric units**

Look at the code in the editor. There are two things we need to pay attention to.

First, the test invocation ensures that the **protection** works properly - the output is:

None

**output**

Second, take a look at the way the **backslash** (\) symbol is used. If you use it in Python code and end a line with it, it will tell Python to continue the line of code in the next line of code.

It can be particularly useful when you have to deal with long lines of code and you'd like to improve code readability.

Okay, but there's something we omitted too easily - the imperial measurements. This function is not too useful for people accustomed to pounds, feet and inches.

What can be done for them?

We can write two simple functions to **convert imperial units to metric ones**. Let's start with pounds.

It is a well-known fact that 1 lb = 0.45359237 kg. We'll use this in our new function.

This is our helper function, named lb\_to\_kg:

def lb\_to\_kg(lb):

return lb \* 0.45359237

print(lb\_to\_kg(1))

The result of the test invocation looks good:

0.45359237

**output**

And now it's time for feet and inches: 1 ft = 0.3048 m, and 1 in = 2.54 cm = 0.0254 m.

The function we've written is named ft\_and\_inch\_to\_m:

def ft\_and\_inch\_to\_m(ft, inch):

return ft \* 0.3048 + inch \* 0.0254

print(ft\_and\_inch\_to\_m(1, 1))

The result of a quick test is:

0.3302

**output**

It looks as expected.

Note: we wanted to name the second parameter just in, not inch, but we couldn't. Do you know why?

in is a Python **keyword** - it cannot be used as a name.

Let's convert *six feet* into meters:

print(ft\_and\_inch\_to\_m(6, 0))

And this is the output:

1.8288000000000002

**output**

It's quite possible that sometimes you may want to use just feet without inches. Will Python help you? Of course it will.

We've modified the code a bit:

def ft\_and\_inch\_to\_m(ft, inch = 0.0):

return ft \* 0.3048 + inch \* 0.0254

print(ft\_and\_inch\_to\_m(6))

Now the inch parameter has its default value equal to 0.0.

The code produces the following output - this is what is expected:

1.8288000000000002

**output**

Finally, the code is able to answer the question: what is the BMI of a person 5'7" tall and weighing 176 lbs?

This is the code we have built:

def ft\_and\_inch\_to\_m(ft, inch = 0.0):

return ft \* 0.3048 + inch \* 0.0254

def lb\_to\_kg(lb):

return lb \* 0.45359237

def bmi(weight, height):

if height < 1.0 or height > 2.5 or weight < 20 or weight > 200:

return None

return weight / height \*\* 2

print(bmi(weight = lb\_to\_kg(176), height = ft\_and\_inch\_to\_m(5, 7)))

And the answer is:

27.565214082533313

**output**

Run the code and test it.

def bmi(weight, height):

if height < 1.0 or height > 2.5 or \

weight < 20 or weight > 200:

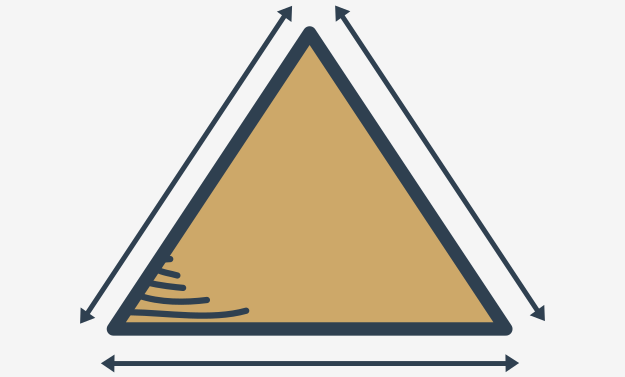
return None

return weight / height \*\* 2

print(bmi(352.5, 1.65))

**Some simple functions: continued**

Let's play with triangles now. We'll start with a function to check whether three sides of given lengths can build a triangle.



We know from school that *the sum of two arbitrary sides has to be longer than the third side*.

It won't be a hard challenge. The function will have **three parameters** - one for each side.

It will return True if the sides can build a triangle, and False otherwise. In this case, is\_a\_triangle is a good name for such a function.

Look at the code in the editor. You can find our function there. Run the program.

It seems that it works well - these are the results:

True

False

**output**

Can we make it more compact? It looks a bit wordy.

This is a more compact version:

def is\_a\_triangle(a, b, c):

if a + b <= c or b + c <= a or c + a <= b:

return False

return True

print(is\_a\_triangle(1, 1, 1))

print(is\_a\_triangle(1, 1, 3))

Can we compact it even more?

Yes, we can - look:

def is\_a\_triangle(a, b, c):

return a + b > c and b + c > a and c + a > b

print(is\_a\_triangle(1, 1, 1))

print(is\_a\_triangle(1, 1, 3))

We've negated the condition (reversed the relational operators and replaced ors with ands, receiving a **universal expression for testing triangles**).

Let's install the function in a larger program. It'll ask the user for three values and make use of the function.

def is\_a\_triangle(a, b, c):

if a + b <= c:

return False

if b + c <= a:

return False

if c + a <= b:

return False

return True

print(is\_a\_triangle(1, 1, 1))

print(is\_a\_triangle(1, 1, 3))

**Some simple functions: triangles and the Pythagorean theorem**

Look at the code in the editor. It asks the user for three values. Then it makes use of the is\_a\_triangle function. The code is ready to run.

In the second step, we'll try to ensure that a certain triangle is a **right-angle triangle**.

We will need to make use of the **Pythagorean theorem**:

**c2 = a2 + b2**

How do we recognize which of the three sides is the hypotenuse?

**The hypotenuse is the longest side**.

Here is the code:

def is\_a\_triangle(a, b, c):

return a + b > c and b + c > a and c + a > b

def is\_a\_right\_triangle(a, b, c):

if not is\_a\_triangle(a, b, c):

return False

if c > a and c > b:

return c \*\* 2 == a \*\* 2 + b \*\* 2

if a > b and a > c:

return a \*\* 2 == b \*\* 2 + c \*\* 2

print(is\_a\_right\_triangle(5, 3, 4))

print(is\_a\_right\_triangle(1, 3, 4))

Look at how we test the relationship between the hypotenuse and the remaining sides - we choose the longest side, and apply the **Pythagorean theorem** to check if everything is right. This requires three checks in total.

def is\_a\_triangle(a, b, c):

return a + b > c and b + c > a and c + a > b

a = float(input('Enter the first side\'s length: '))

b = float(input('Enter the second side\'s length: '))

c = float(input('Enter the third side\'s length: '))

if is\_a\_triangle(a, b, c):

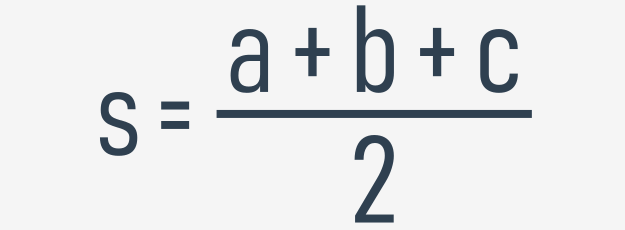
print('Yes, it can be a triangle.')

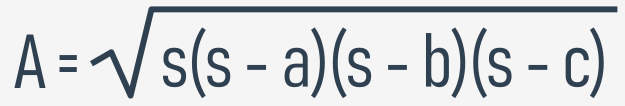
else:

print('No, it can\'t be a triangle.')

**Some simple functions: evaluating a triangle's area**

We can also evaluate a triangle's area. **Heron's formula** will be handy here:





We're going use the exponentiation operator to find the square root - it may seem strange, but it works:



This is the resulting code:

def is\_a\_triangle(a, b, c):

return a + b > c and b + c > a and c + a > b

def heron(a, b, c):

p = (a + b + c) / 2

return (p \* (p - a) \* (p - b) \* (p - c)) \*\* 0.5

def area\_of\_triangle(a, b, c):

if not is\_a\_triangle(a, b, c):

return None

return heron(a, b, c)

print(area\_of\_triangle(1., 1., 2. \*\* .5))

We try it with a right-angle triangle as a half of a square with one side equal to 1. This means that its area should be equal to 0.5.

It's odd - the code produces the following output:

0.49999999999999983

**output**

It's very close to 0.5, but it isn't exactly 0.5. What does it mean? Is it an error?

No, it isn't. This is **the specifics of floating-point calculations**. We'll tell you more about it soon.

def is\_a\_triangle(a, b, c):

return a + b > c and b + c > a and c + a > b

a = float(input('Enter the first side\'s length: '))

b = float(input('Enter the second side\'s length: '))

c = float(input('Enter the third side\'s length: '))

if is\_a\_triangle(a, b, c):

print('Yes, it can be a triangle.')

else:

print('No, it can\'t be a triangle.')

**Some simple functions: factorials**

Another function we're about to write is **factorials**. Do you remember how a factorial is defined?

0! = 1 (yes! it's true)  
1! = 1  
2! = 1 \* 2  
3! = 1 \* 2 \* 3  
4! = 1 \* 2 \* 3 \* 4  
:  
:  
n! = 1 \* 2 \* 3 \* 4 \* ... \* n-1 \* n

It's marked with an **exclamation mark**, and is equal to the **product** of all natural numbers from one up to its argument.

Let's write our code. We'll create a function and call it factorial\_function. Here is the code:

def factorial\_function(n):

if n < 0:

return None

if n < 2:

return 1

product = 1

for i in range(2, n + 1):

product \*= i

return product

for n in range(1, 6): # testing

print(n, factorial\_function(n))

Notice how we mirror step-by-step the mathematical definition, and how we use the for loop to **find the product**.

We add a simple testing code, and these are the results we get:

1 1

2 2

3 6

4 24

5 120

**output**

# Some simple functions: Fibonacci numbers

Are you familiar with **Fibonacci** numbers?

They are a **sequence of integer numbers** built using a very simple rule:

* the first element of the sequence is equal to one (**Fib1 = 1**)
* the second is also equal to one (**Fib2 = 1**)
* every subsequent number is the the\_sum of the two preceding numbers:  
  (**Fibi = Fibi-1 + Fibi-2**)

Here are some of the first Fibonacci numbers:

fib\_1 = 1  
fib\_2 = 1  
fib\_3 = 1 + 1 = 2  
fib\_4 = 1 + 2 = 3  
fib\_5 = 2 + 3 = 5  
fib\_6 = 3 + 5 = 8  
fib\_7 = 5 + 8 = 13

What do you think about **implementing this as a function**?

Let's create our fib function and test it. Here it is:

def fib(n):

if n < 1:

return None

if n < 3:

return 1

elem\_1 = elem\_2 = 1

the\_sum = 0

for i in range(3, n + 1):

the\_sum = elem\_1 + elem\_2

elem\_1, elem\_2 = elem\_2, the\_sum

return the\_sum

for n in range(1, 10): # testing

print(n, "->", fib(n))

Analyze the for loop body carefully, and find out how we **move the**elem\_1**and**elem\_2**variables through the subsequent Fibonacci numbers**.

The test part of the code produces the following output:

1 -> 1

2 -> 1

3 -> 2

4 -> 3

5 -> 5

6 -> 8

7 -> 13

8 -> 21

9 -> 34

**output**

**Some simple functions: recursion**

There's one more thing we want to show you to make everything complete - it's **recursion**.

This term may describe many different concepts, but one of them is especially interesting - the one referring to computer programming.

In this field, recursion is a **technique where a function invokes itself**.

These two cases seem to be the best to illustrate the phenomenon - factorials and Fibonacci numbers. Especially the latter.

**The Fibonacci numbers definition is a clear example of recursion**. We already told you that:

**Fibi = Fibi-1 + Fibi-2**

The definition of the ith number refers to the i-1 number, and so on, till you reach the first two.

Can it be used in the code? Yes, it can. It can also make the code shorter and clearer.

The second version of our fib() function makes direct use of this definition:

def fib(n):

if n < 1:

return None

if n < 3:

return 1

return fib(n - 1) + fib(n - 2)

The code is much clearer now.

But is it really safe? Does it entail any risk?

Yes, there is a little risk indeed. **If you forget to consider the conditions which can stop the chain of recursive invocations, the program may enter an infinite loop**. You have to be careful.

The factorial has a second, **recursive** side too. Look:

n! = 1 × 2 × 3 × ... × n-1 × n

It's obvious that:

1 × 2 × 3 × ... × n-1 = (n-1)!

So, finally, the result is:

n! = (n-1)! × n

This is in fact a ready recipe for our new solution.

Here it is:

def factorial\_function(n):

if n < 0:

return None

if n < 2:

return 1

return n \* factorial\_function(n - 1)

Does it work? Yes, it does. Try it for yourself.

Our short *functional* journey is almost over. The next section will take care of two curious Python data types: tuples and dictionaries.

def fib(n):

if n < 1:

return None

if n < 3:

return 1

elem\_1 = elem\_2 = 1

the\_sum = 0

for i in range(3, n + 1):

the\_sum = elem\_1 + elem\_2

elem\_1, elem\_2 = elem\_2, the\_sum

return the\_sum

for n in range(1, 10):

print(n, "->", fib(n))

**Key takeaways**

1. A function can call other functions or even itself. When a function calls itself, this situation is known as **recursion**, and the function which calls itself and contains a specified termination condition (i.e., the base case - a condition which doesn't tell the function to make any further calls to that function) is called a **recursive** function.

2. You can use recursive functions in Python to write **clean, elegant code, and divide it into smaller, organized chunks**. On the other hand, you need to be very careful as it might be **easy to make a mistake and create a function which never terminates**. You also need to remember that **recursive calls consume a lot of memory**, and therefore may sometimes be inefficient.

When using recursion, you need to take all its advantages and disadvantages into consideration.

The factorial function is a classic example of how the concept of recursion can be put in practice:

# Recursive implementation of the factorial function.

def factorial(n):

if n == 1: # The base case (termination condition.)

return 1

else:

return n \* factorial(n - 1)

print(factorial(4)) # 4 \* 3 \* 2 \* 1 = 24

**Exercise 1**

What will happen when you attempt to run the following snippet and why?

def factorial(n):

return n \* factorial(n - 1)

print(factorial(4))

Check

The factorial function has no termination condition (no base case) so Python will raise an exception (RecursionError: maximum recursion depth exceeded)

**Exercise 2**

What is the output of the following snippet?

def fun(a):

if a > 30:

return 3

else:

return a + fun(a + 3)

print(fun(25))

Check

56

# Sequence types and mutability

Before we start talking about **tuples** and **dictionaries**, we have to introduce two important concepts: **sequence types** and **mutability**.

A **sequence type is a type of data in Python which is able to store more than one value (or less than one, as a sequence may be empty), and these values can be sequentially (hence the name) browsed**, element by element.

As the for loop is a tool especially designed to iterate through sequences, we can express the definition as: **a sequence is data which can be scanned by the**for**loop**.

You've encountered one Python sequence so far - the list. The list is a classic example of a Python sequence, although there are some other sequences worth mentioning, and we're going to present them to you now.

The second notion - **mutability** - is a property of any of Python's data that describes its readiness to be freely changed during program execution. There are two kinds of Python data: **mutable** and **immutable**.

**Mutable data can be freely updated at any time** - we call such an operation in situ.

*In situ* is a Latin phrase that translates as literally *in position*. For example, the following instruction modifies the data in situ:

list.append(1)

**Immutable data cannot be modified in this way**.

Imagine that a list can only be assigned and read over. You would be able neither to append an element to it, nor remove any element from it. This means that appending an element to the end of the list would require the recreation of the list from scratch.

You would have to build a completely new list, consisting of the all elements of the already existing list, plus the new element.

The data type we want to tell you about now is a **tuple**. **A tuple is an immutable sequence type**. It can behave like a list, but it mustn't be modified in situ.

## What is a tuple?

The first and the clearest distinction between lists and tuples is the syntax used to create them - **tuples prefer to use parenthesis**, whereas lists like to see brackets, although it's also **possible to create a tuple just from a set of values separated by commas**.

Look at the example:

tuple\_1 = (1, 2, 4, 8)

tuple\_2 = 1., .5, .25, .125

There are two tuples, both containing **four elements**.

Let's print them:

tuple\_1 = (1, 2, 4, 8)

tuple\_2 = 1., .5, .25, .125

print(tuple\_1)

print(tuple\_2)

This is what you should see in the console:

(1, 2, 4, 8)

(1.0, 0.5, 0.25, 0.125)

**output**

Note: **each tuple element may be of a different type** (floating-point, integer, or any other not-as-yet-introduced kind of data).

## How to create a tuple?

It is possible to create an empty tuple - parentheses are required then:

empty\_tuple = ()

If you want to create a **one-element tuple**, you have to take into consideration the fact that, due to syntax reasons (a tuple has to be distinguishable from an ordinary, single value), you must end the value with a comma:

one\_element\_tuple\_1 = (1, )

one\_element\_tuple\_2 = 1.,

Removing the commas won't spoil the program in any syntactical sense, but you will instead get two single variables, not tuples.

**How to use a tuple?**

If you want to get the elements of a tuple in order to read them over, you can use the same conventions to which you're accustomed while using lists.

Take a look at the code in the editor.

The program should produce the following output - run it and check:

1

1000

(10, 100, 1000)

(1, 10)

1

10

100

1000

**output**

The similarities may be misleading - **don't try to modify a tuple's contents**! It's not a list!

All of these instructions (except the topmost one) will cause a runtime error:

my\_tuple = (1, 10, 100, 1000)

my\_tuple.append(10000)

del my\_tuple[0]

my\_tuple[1] = -10

This is the message that Python will give you in the console window:

AttributeError: 'tuple' object has no attribute 'append'

**output**

my\_tuple = (1, 10, 100, 1000)

print(my\_tuple[0])

print(my\_tuple[-1])

print(my\_tuple[1:])

print(my\_tuple[:-2])

for elem in my\_tuple:

print(elem)

# How to use a tuple: continued

What else can tuples do for you?

* the len() function accepts tuples, and returns the number of elements contained inside;
* the + operator can join tuples together (we've shown you this already)
* the \* operator can multiply tuples, just like lists;
* the in and not in operators work in the same way as in lists.

The snippet in the editor presents them all.

The output should look as follows:

9

(1, 10, 100, 1000, 10000)

(1, 10, 100, 1, 10, 100, 1, 10, 100)

True

True

**output**

One of the most useful tuple properties is their ability to **appear on the left side of the assignment operator**. You saw this phenomenon some time ago, when it was necessary to find an elegant tool to swap two variables' values.

Take a look at the snippet below:

var = 123

t1 = (1, )

t2 = (2, )

t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)

It shows three tuples interacting - in effect, the values stored in them "circulate" - t1 becomes t2, t2 becomes t3, and t3 becomes t1.

Note: the example presents one more important fact: a **tuple's elements can be variables**, not only literals. Moreover, they can be expressions if they're on the right side of the assignment operator.

my\_tuple = (1, 10, 100)

t1 = my\_tuple + (1000, 10000)

t2 = my\_tuple \* 3

print(len(t2))

print(t1)

print(t2)

print(10 in my\_tuple)

print(-10 not in my\_tuple)

# What is a dictionary?

The **dictionary** is another Python data structure. It's **not a sequence** type (but can be easily adapted to sequence processing) and it is **mutable**.

To explain what the Python dictionary actually is, it is important to understand that it is literally a dictionary.

The Python dictionary works in the same way as **a bilingual dictionary**. For example, you have an English word (e.g., cat) and need its French equivalent. You browse the dictionary in order to find the word (you may use different techniques to do that - it doesn't matter) and eventually you get it. Next, you check the French counterpart and it is (most probably) the word "chat".



In Python's world, the word you look for is named a key. The word you get from the dictionary is called a value.

This means that a dictionary is a set of **key-value** pairs. Note:

* each key must be **unique** - it's not possible to have more than one key of the same value;
* a key may be **any immutable type of object**: it can be a number (integer or float), or even a string, but not a list;
* a dictionary is not a list - a list contains a set of numbered values, while a **dictionary holds pairs of values**;
* the len() function works for dictionaries, too - it returns the numbers of key-value elements in the dictionary;
* a dictionary is a **one-way tool** - if you have an English-French dictionary, you can look for French equivalents of English terms, but not vice versa.

Now we can show you some working examples.

## How to make a dictionary?

If you want to assign some initial pairs to a dictionary, you should use the following syntax:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

phone\_numbers = {'boss': 5551234567, 'Suzy': 22657854310}

empty\_dictionary = {}

print(dictionary)

print(phone\_numbers)

print(empty\_dictionary)

In the first example, the dictionary uses keys and values which are both strings. In the second one, the keys are strings, but the values are integers. The reverse layout (keys → numbers, values → strings) is also possible, as well as number-number combination.

The list of pairs is **surrounded by curly braces**, while the pairs themselves are **separated by commas**, and the **keys and values by colons**.

The first of our dictionaries is a very simple English-French dictionary. The second - a very tiny telephone directory.

The empty dictionaries are constructed by an **empty pair of curly braces** - nothing unusual.

The dictionary as a whole can be printed with a single print() invocation. The snippet **may** produce the following output:

{'dog': 'chien', 'horse': 'cheval', 'cat': 'chat'}

{'Suzy': 5557654321, 'boss': 5551234567}

{}

**output**

Have you noticed anything surprising? The order of the printed pairs is different than in the initial assignment. What does that mean?

First of all, it's a confirmation that **dictionaries are not lists** - they don't preserve the order of their data, as the order is completely meaningless (unlike in real, paper dictionaries). The order in which a dictionary **stores its data is completely out of your control**, and your expectations. That's normal. (\*)

NOTE

(\*) In Python 3.6x dictionaries have become **ordered** collections by default. Your results may vary depending on what Python version you're using.

# How to use a dictionary?

If you want to get any of the values, you have to deliver a valid key value:

print(dictionary['cat'])

print(phone\_numbers['Suzy'])

Getting a dictionary's value resembles indexing, especially thanks to the brackets surrounding the key's value.

Note:

* if the key is a string, you have to specify it as a string;
* **keys are case-sensitive**: 'Suzy' is something different from 'suzy'.

The snippet outputs two lines of text:

chat

22657854310

**output**

And now the most important news: you **mustn't use a non-existent key**. Trying something like this:

print(phone\_numbers['president'])

will cause a runtime error. Try to do it.

Fortunately, there's a simple way to avoid such a situation. The in operator, together with its companion, not in, can salvage this situation.

The following code safely searches for some French words:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

words = ['cat', 'lion', 'horse']

for word in words:

if word in dictionary:

print(word, "->", dictionary[word])

else:

print(word, "is not in dictionary")

The code's output looks as follows:

cat -> chat

lion is not in dictionary

horse -> cheval

**output**

NOTE

When you write a big or lengthy expression, it may be a good idea to keep it vertically aligned. This is how you can make your code more readable and more programmer-friendly, e.g.:

# Example 1:

dictionary = {

"cat": "chat",

"dog": "chien",

"horse": "cheval"

}

# Example 2:

phone\_numbers = {'boss': 5551234567,

'Suzy': 22657854310

}

This kind of formatting is called a **hanging indent**.

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

phone\_numbers = {'boss' : 5551234567, 'Suzy' : 22657854310}

empty\_dictionary = {}

# Print the values here.

# How to use a dictionary: the keys()

Can dictionaries be **browsed** using the for loop, like lists or tuples?

No and yes.

No, because a dictionary is **not a sequence type** - the for loop is useless with it.

Yes, because there are simple and very effective tools that can **adapt any dictionary to the**for**loop requirements** (in other words, building an intermediate link between the dictionary and a temporary sequence entity).

The first of them is a method named keys(), possessed by each dictionary. The method **returns an iterable object consisting of all the keys gathered within the dictionary**. Having a group of keys enables you to access the whole dictionary in an easy and handy way.

Just like here:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for key in dictionary.keys():

print(key, "->", dictionary[key]

The code's output looks as follows:

horse -> cheval

dog -> chien

cat -> chat

**output**

## The sorted() function

Do you want it **sorted**? Just enrich the for loop to get such a form:

for key in sorted(dictionary.keys()):

The sorted() function will do its best - the output will look like this:

cat -> chat

dog -> chien

horse -> cheval

**output**

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for key in dictionary.keys():

print(key, "->", dictionary[key])

**How to use a dictionary: The items() and values() methods**

Another way is based on using a dictionary's method named items(). The method **returns tuples** (this is the first example where tuples are something more than just an example of themselves) **where each tuple is a key-value pair**.

This is how it works:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for english, french in dictionary.items():

print(english, "->", french)

Note the way in which the tuple has been used as a for loop variable.

The example prints:

cat -> chat

dog -> chien

horse -> cheval

**output**

There is also a method named values(), which works similarly to keys(), but **returns values**.

Here is a simple example:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for french in dictionary.values():

print(french)

As the dictionary is not able to automatically find a key for a given value, the role of this method is rather limited.

Here is the expected output:

cheval

chien

chat

**output**

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for english, french in dictionary.items():

print(english, "->", french)

# How to use a dictionary: modifying and adding values

Assigning a new value to an existing key is simple - as dictionaries are fully **mutable**, there are no obstacles to modifying them.

We're going to replace the value "chat" with "minou", which is not very accurate, but it will work well with our example.

Look:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary['cat'] = 'minou'

print(dictionary)

The output is:

{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}

**output**

## Adding a new key

Adding a new key-value pair to a dictionary is as simple as changing a value - you only have to assign a value to a new, **previously non-existent key**.

Note: this is very different behavior compared to lists, which don't allow you to assign values to non-existing indices.

Let's add a new pair of words to the dictionary - a bit weird, but still valid:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary['swan'] = 'cygne'

print(dictionary)

The example outputs:

{'cat': 'chat', 'dog': 'chien', 'horse': 'cheval', 'swan': 'cygne'}

**output**

**EXTRA**

You can also insert an item to a dictionary by using the update() method, e.g.:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary.update({"duck": "canard"})

print(dictionary)

## Removing a key

Can you guess how to remove a key from a dictionary?

Note: removing a key will always cause the **removal of the associated value**. **Values cannot exist without their keys**.

This is done with the del instruction.

Here's the example:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

del dictionary['dog']

print(dictionary)

Note: **removing a non-existing key causes an error**.

The example outputs:

{'cat': 'chat', 'horse': 'cheval'}

**output**

**EXTRA**

To remove the last item in a dictionary, you can use the popitem() method:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary.popitem()

print(dictionary) # outputs: {'cat': 'chat', 'dog': 'chien'}

In the older versions of Python, i.e., before 3.6.7, the popitem() method removes a random item from a dictionary.

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary['cat'] = 'minou'

print(dictionary)

# Tuples and dictionaries can work together

We've prepared a simple example, showing how tuples and dictionaries can work together.

Let's imagine the following problem:

* you need a program to evaluate the students' average scores;
* the program should ask for the student's name, followed by her/his single score;
* the names may be entered in any order;
* entering an empty name finishes the inputting of the data (note 1: entering an empty score will raise the ValueError exception, but don't worry about that now, you'll see how to handle such cases when we talk about exceptions in the second part of the Python Essentials course series)
* a list of all names, together with the evaluated average score, should be then emitted.

Look at the code in the editor. This is how to do it.

Now, let's analyze it line by line:

* **line 1**: create an empty dictionary for the input data; the student's name is used as a key, while all the associated scores are stored in a tuple (the tuple may be a dictionary value - that's not a problem at all)
* **line 3**: enter an "infinite" loop (don't worry, it'll break at the right moment)
* **line 4**: read the student's name here;
* **line 5-6**: if the name is an empty string (), leave the loop;
* **line 8**: ask for one of the student's scores (an integer from the range 0-10)
* **line 9-10**: if the score entered is not within the range from 0 to 10, leave the loop;
* **line 12-13**: if the student's name is already in the dictionary, lengthen the associated tuple with the new score (note the += operator)
* **line 14-15**: if this is a new student (unknown to the dictionary), create a new entry - its value is a one-element tuple containing the entered score;
* **line 17**: iterate through the sorted students' names;
* **line 18-19**: initialize the data needed to evaluate the average (sum and counter)
* **line 20-22**: we iterate through the tuple, taking all the subsequent scores and updating the sum, together with the counter;
* **line 23**: evaluate and print the student's name and average score.

This is a record of the conversation we had with our program:

Enter the student's name: Bob

Enter the student's score (0-10): 7

Enter the student's name: Andy

Enter the student's score (0-10): 3

Enter the student's name: Bob

Enter the student's score (0-10): 2

Enter the student's name: Andy

Enter the student's score (0-10): 10

Enter the student's name: Andy

Enter the student's score (0-10): 3

Enter the student's name: Bob

Enter the student's score (0-10): 9

Enter the student's name:

Andy : 5.333333333333333

Bob : 6.0

**output**

school\_class = {}

while True:

name = input("Enter the student's name: ")

if name == '':

break

score = int(input("Enter the student's score (0-10): "))

if score not in range(0, 11):

break

if name in school\_class:

school\_class[name] += (score,)

else:

school\_class[name] = (score,)

for name in sorted(school\_class.keys()):

adding = 0

counter = 0

for score in school\_class[name]:

adding += score

counter += 1

print(name, ":", adding / counter)

**Key takeaways: tuples**

1. **Tuples** are ordered and unchangeable (immutable) collections of data. They can be thought of as immutable lists. They are written in round brackets:

my\_tuple = (1, 2, True, "a string", (3, 4), [5, 6], None)

print(my\_tuple)

my\_list = [1, 2, True, "a string", (3, 4), [5, 6], None]

print(my\_list)

Each tuple element may be of a different type (i.e., integers, strings, booleans, etc.). What is more, tuples can contain other tuples or lists (and the other way round).

2. You can create an empty tuple like this:

empty\_tuple = ()

print(type(empty\_tuple)) # outputs: <class 'tuple'>

3. A one-element tuple may be created as follows:

one\_elem\_tuple\_1 = ("one", ) # Brackets and a comma.

one\_elem\_tuple\_2 = "one", # No brackets, just a comma.

If you remove the comma, you will tell Python to create a **variable**, not a tuple:

my\_tuple\_1 = 1,

print(type(my\_tuple\_1)) # outputs: <class 'tuple'>

my\_tuple\_2 = 1 # This is not a tuple.

print(type(my\_tuple\_2)) # outputs: <class 'int'>

4. You can access tuple elements by indexing them:

my\_tuple = (1, 2.0, "string", [3, 4], (5, ), True)

print(my\_tuple[3]) # outputs: [3, 4]

5. Tuples are **immutable**, which means you cannot change their elements (you cannot append tuples, or modify, or remove tuple elements). The following snippet will cause an exception:

my\_tuple = (1, 2.0, "string", [3, 4], (5, ), True)

my\_tuple[2] = "guitar" # The TypeError exception will be raised.

However, you can delete a tuple as a whole:

my\_tuple = 1, 2, 3,

del my\_tuple

print(my\_tuple) # NameError: name 'my\_tuple' is not defined

6. You can loop through a tuple elements (Example 1), check if a specific element is (not)present in a tuple (Example 2), use the len() function to check how many elements there are in a tuple (Example 3), or even join/multiply tuples (Example 4):

# Example 1

tuple\_1 = (1, 2, 3)

for elem in tuple\_1:

print(elem)

# Example 2

tuple\_2 = (1, 2, 3, 4)

print(5 in tuple\_2)

print(5 not in tuple\_2)

# Example 3

tuple\_3 = (1, 2, 3, 5)

print(len(tuple\_3))

# Example 4

tuple\_4 = tuple\_1 + tuple\_2

tuple\_5 = tuple\_3 \* 2

print(tuple\_4)

print(tuple\_5)

**EXTRA**

You can also create a tuple using a Python built-in function called tuple(). This is particularly useful when you want to convert a certain iterable (e.g., a list, range, string, etc.) to a tuple:

my\_tuple = tuple((1, 2, "string"))

print(my\_tuple)

my\_list = [2, 4, 6]

print(my\_list) # outputs: [2, 4, 6]

print(type(my\_list)) # outputs: <class 'list'>

tup = tuple(my\_list)

print(tup) # outputs: (2, 4, 6)

print(type(tup)) # outputs: <class 'tuple'>

By the same fashion, when you want to convert an iterable to a list, you can use a Python built-in function called list():

tup = 1, 2, 3,

my\_list = list(tup)

print(type(my\_list)) # outputs: <class 'list'>

**Key takeaways: dictionaries**

1. Dictionaries are unordered**\***, changeable (mutable), and indexed collections of data. (**\***In Python 3.6x dictionaries have become ordered by default.

Each dictionary is a set of *key: value* pairs. You can create it by using the following syntax:

my\_dictionary = {

key1: value1,

key2: value2,

key3: value3,

}

2. If you want to access a dictionary item, you can do so by making a reference to its key inside a pair of square brackets (ex. 1) or by using the get() method (ex. 2):

pol\_eng\_dictionary = {

"kwiat": "flower",

"woda": "water",

"gleba": "soil"

}

item\_1 = pol\_eng\_dictionary["gleba"] # ex. 1

print(item\_1) # outputs: soil

item\_2 = pol\_eng\_dictionary.get("woda")

print(item\_2) # outputs: water

3. If you want to change the value associated with a specific key, you can do so by referring to the item's key name in the following way:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

pol\_eng\_dictionary["zamek"] = "lock"

item = pol\_eng\_dictionary["zamek"]

print(item) # outputs: lock

4. To add or remove a key (and the associated value), use the following syntax:

phonebook = {} # an empty dictionary

phonebook["Adam"] = 3456783958 # create/add a key-value pair

print(phonebook) # outputs: {'Adam': 3456783958}

del phonebook["Adam"]

print(phonebook) # outputs: {}

You can also insert an item to a dictionary by using the update() method, and remove the last element by using the popitem() method, e.g.:

pol\_eng\_dictionary = {"kwiat": "flower"}

pol\_eng\_dictionary.update({"gleba": "soil"})

print(pol\_eng\_dictionary) # outputs: {'kwiat': 'flower', 'gleba': 'soil'}

pol\_eng\_dictionary.popitem()

print(pol\_eng\_dictionary) # outputs: {'kwiat': 'flower'}

5. You can use the for loop to loop through a dictionary, e.g.:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

for item in pol\_eng\_dictionary:

print(item)

# outputs: zamek

# woda

# gleba

6. If you want to loop through a dictionary's keys and values, you can use the items() method, e.g.:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

for key, value in pol\_eng\_dictionary.items():

print("Pol/Eng ->", key, ":", value)

7. To check if a given key exists in a dictionary, you can use the in keyword:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

if "zamek" in pol\_eng\_dictionary:

print("Yes")

else:

print("No")

8. You can use the del keyword to remove a specific item, or delete a dictionary. To remove all the dictionary's items, you need to use the clear() method:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

print(len(pol\_eng\_dictionary)) # outputs: 3

del pol\_eng\_dictionary["zamek"] # remove an item

print(len(pol\_eng\_dictionary)) # outputs: 2

pol\_eng\_dictionary.clear() # removes all the items

print(len(pol\_eng\_dictionary)) # outputs: 0

del pol\_eng\_dictionary # removes the dictionary

9. To copy a dictionary, use the copy() method:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

copy\_dictionary = pol\_eng\_dictionary.copy()

**Key takeaways: tuples and dictionaries**

**Exercise 1**

What happens when you attempt to run the following snippet?

my\_tup = (1, 2, 3)

print(my\_tup[2])

Check

The program will print 3 to the screen.

**Exercise 2**

What is the output of the following snippet?

tup = 1, 2, 3

a, b, c = tup

print(a \* b \* c)

Check

The program will print 6 to the screen. The tup tuple elements have been "unpacked" in the a, b, and c variables.

**Exercise 3**

Complete the code to correctly use the count() method to find the number of duplicates of 2 in the following tuple.

tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9

duplicates = # Write your code here.

print(duplicates) # outputs: 4

Check

tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9

duplicates = tup.count(2)

print(duplicates) # outputs: 4

**Exercise 4**

Write a program that will "glue" the two dictionaries (d1 and d2) together and create a new one (d3).

d1 = {'Adam Smith': 'A', 'Judy Paxton': 'B+'}

d2 = {'Mary Louis': 'A', 'Patrick White': 'C'}

d3 = {}

for item in (d1, d2):

# Write your code here.

print(d3)

Check

Sample solution:  
d1 = {'Adam Smith': 'A', 'Judy Paxton': 'B+'}

d2 = {'Mary Louis': 'A', 'Patrick White': 'C'}

d3 = {}

for item in (d1, d2):

d3.update(item)

print(d3)

**Exercise 5**

Write a program that will convert the my\_list list to a tuple.

my\_list = ["car", "Ford", "flower", "Tulip"]

t = # Write your code here.

print(t)

Check

Sample solution:  
my\_list = ["car", "Ford", "flower", "Tulip"]

t = tuple(my\_list)

print(t)

**Exercise 6**

Write a program that will convert the colors tuple to a dictionary.

colors = (("green", "#008000"), ("blue", "#0000FF"))

# Write your code here.

print(colors\_dictionary)

Check

Sample solution:  
colors = (("green", "#008000"), ("blue", "#0000FF"))

colors\_dictionary = dict(colors)

print(colors\_dictionary)

**Exercise 7**

What will happen when you run the following code?

my\_dictionary = {"A": 1, "B": 2}

copy\_my\_dictionary = my\_dictionary.copy()

my\_dictionary.clear()

print(copy\_my\_dictionary)

Check

The program will print {'A': 1, 'B': 2} to the screen.

**Exercise 8**

What is the output of the following program?

colors = {

"white": (255, 255, 255),

"grey": (128, 128, 128),

"red": (255, 0, 0),

"green": (0, 128, 0)

}

for col, rgb in colors.items():

print(col, ":", rgb)

Check

white : (255, 255, 255)

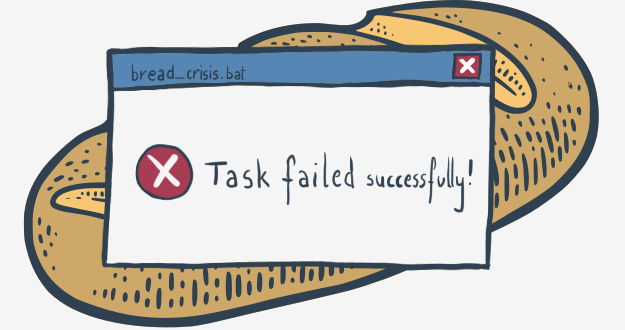
grey : (128, 128, 128)

red : (255, 0, 0)

green : (0, 128, 0)

**Errors – the developer's daily bread**

It seems indisputable that all programmers (including you) want to write error-free code and do their best to achieve this goal. Unfortunately, nothing is perfect in this world and software is no exception. Pay attention to the word **exception** as we’ll see it again very soon in a meaning that has nothing in common with the absolute.



To err is human. It's impossible to make no mistakes, and it's impossible to write error-free code. Don't get us wrong – we don't want to convince you that writing messy and faulty programs is a virtue. We rather want to explain that even the most careful programmer is not able to avoid minor or major defects. It's only those who do nothing that make no mistakes.

Paradoxically, accepting this difficult truth can make you a better programmer and may improve your code quality.

"How could this be possible?", you may ask.

We'll try to show you.

**Errors in data vs. errors in code**

Dealing with programming errors has (at least) two sides. The one appears when you get into trouble because your – apparently correct – code is fed with bad data. For example, you expect the code will input an integer value, but your careless user enters some random letters instead.

It may happen that your code will be terminated then, and the user will be left alone with a terse and ambiguous error message on the screen. The user will be unsatisfied, and you should be unsatisfied, too.

We're going to show you how to protect your code from this kind of failure and how not to provoke the user's anger.

The other side of dealing with programming errors reveals itself when undesirable code behavior is caused by mistakes you made when you were writing your program. This kind of error is commonly called a “bug”, which is a manifestation of a well-established belief that if a program works badly, it must be caused by malicious bugs which live inside the computer hardware and cause short circuits or other interference.

This idea is not as mad as it may look – such incidents were common in times when computers occupied large halls, consumed kilowatts of electricity, and produced enormous amounts of heat. Fortunately or not, these times are gone forever and the only bugs which can spoil your code are those you sowed in the code yourself. Therefore, we will try to show you how to find and eliminate your bugs, in other words, how to debug your code.

Let's start the journey through the land of errors and bugs.

**When data is not what it should be**

Let's write a piece of extremely trivial code – it will read a natural number (a non-negative integer) and print its reciprocal. In this way, 2 will turn into 0.5 (1/2) and 4 into 0.25 (1/4). Here’s the program:

Is there anything that can go wrong with it? The code is so brief and so compact that it doesn't seem like we’ll find any trouble there.

It seems that you already know where we are going. Yes, you're right – entering data that is not an integer (which also includes entering nothing at all) will completely ruin the program execution. This is what the code's user will see:

Traceback (most recent call last):

File "code.py", line 1, in

value = int(input('Enter a natural number: '))

ValueError: invalid literal for int() with base 10: ''

**output**

All the lines Python shows you are meaningful and important, but the last line seems to be the most valuable. The first word in the line is the name of the **exception** which causes your code to stop. It's ValueError here. The rest of the line is just a brief explanation which more precisely specifies the cause of the occurred exception.

How do you deal with it? How do you protect your code from termination, the user from disappointment, and yourself from the user's dissatisfaction?

The very first thought that can come to your mind is to check if the data provided by the user is valid and to refuse to cooperate if the data is incorrect. In this case, the check can rely on the fact that we expect the input string to contain digits only.

You should already be able to implement this check and write it yourself, shouldn’t you? It is also possible to check if the value variable's type is an int (Python has a special means for these kinds of checks – it's an operator named is. The check itself may look like this:

type(value) is int

and evaluates to true if the current value variable's type is int.

Please forgive us if we don't spend any more time on it now – you will find more detailed explanations of the is operator in a course module devoted to Object-Oriented Programming.

You may be surprised to learn that we don't want you to do any preliminary data validation. Why? Because this is not the way Python recommends. Really.

value = int(input('Enter a natural number: '))

print('The reciprocal of', value, 'is', 1/value)

# The *try-except* branch

In the Python world, there is a rule that says: *"It’s better to beg for forgiveness than to ask for permission"*.

Let's stop here for a moment. Don't get us wrong – we don't want you to apply the rule in your everyday life. Don't take anyone's car without permission in the hope that you can be so convincing that you will avoid conviction. The rule is about something else.

Actually, the rule reads: *"it's better to handle an error when it happens than to try to avoid it"*.

*"Okay,"* you may say now, *'but how should I beg for forgiveness when the program is terminated and there is nothing left that can be done?"* This is where the **exception** comes on the scene.

Look at the code in the editor.

You can see two branches here:

* first, starting with the try keyword – this is the place where you put the code you suspect is risky and may be terminated in case of error; note: this kind of error is called an **exception**, while the exception occurrence is called **raising** – we can say that an exception is (or was) raised;

* second, the part of the code starting with the except keyword is designed to handle the exception; it's up to you what you want to do here: you can clean up the mess or you can just sweep the problem under the carpet (although we would prefer the first solution).

So, we could say that these two blocks work like this:

* the try keyword marks the place where you try to do something without permission;
* the except keyword starts a location where you can show off your apology talents.

As you can see, this approach accepts errors (treats them as a normal part of the program's life) instead of escalating efforts to avoid errors at all.

try:

# It's a place where

# you can do something

# without asking for permission.

except:

# It's a spot dedicated to

# solemnly begging for forgiveness.

# The exception proves the rule

Let's rewrite the code to adopt the Python approach to life:

Let us summarize what we talked about:

* any part of the code placed between try and except is executed in a very special way – any error which occurs here **won't terminate program execution**. Instead, the control will immediately jump to the first line situated after the except keyword, and no other part of the try branch is executed;

* the code in the except branch is activated only when an exception has been encountered inside the try block. There is no way to get there by any other means;

* when either the try block or the except block is executed successfully, the control returns to the normal path of execution, and any code located beyond in the source file is executed as if nothing happened.

Now we want to ask you an innocent question: is ValueError the only way the control could fall into the except branch?

Analyze the code carefully and think over your answer!

try:

value = int(input('Enter a natural number: '))

print('The reciprocal of', value, 'is', 1/value)

except:

print('I do not know what to do.')

# How to deal with more than one exception

The answer is obviously "no" – there is more than one possible way to raise an exception. For example, a user may enter zero as an input – can you predict what will happen next?

Yes, you're right – the division placed inside the print() function invocation will raise the ZeroDivisionError. As you may expect, the code's behavior will be the same as in the previous case – the user will see the *"I do not know what to do..."* message, which seems to be quite reasonable in this context, but it's also possible that you would want to handle this kind of problem in a bit different way.

Is it possible? Of course, it is. There are at least two approaches you can implement here.

The first of them is simple and complicated at the same time: you can just add two separate try blocks, one including the input() function invocation where the ValueError may be raised, and the second devoted to handling possible issues induced by the division. Both these try blocks would have their own except branches, and in effect you will gain full control over two different errors.

This solution is good, but it is a bit lengthy – the code becomes unnecessarily bloated. Moreover, it's not the only danger that awaits you. Note that leaving the first try-except block leaves a lot of uncertainty – you will have to add extra code to ensure that the value the user has entered is safe to use in division. This is how a seemingly simple solution becomes overly complicated.

Fortunately, Python offers a simpler way to deal with this kind of challenge.

## Two exceptions after one *try*

Look at the code in the editor. As you can see, we've just introduced the second except branch. This is not the only difference – note that both branches have exception **names** specified. In this variant, each of the expected exceptions has its own way of handling the error, but it must be emphasized that **only one** of all branches can intercept the control – **if one of the branches is executed, all the other branches remain idle**.

Additionally, the number of except branches is not limited – you can specify as many or as few of them as you need, but don't forget that **none of the exceptions can be specified more than once**.

But this still isn't the last Python word on exceptions. Stay tuned.

try:

value = int(input('Enter a natural number: '))

print('The reciprocal of', value, 'is', 1/value)

except ValueError:

print('I do not know what to do.')

except ZeroDivisionError:

print('Division by zero is not allowed in our Universe.')

**The default exception and how to use it**

The code has changed again – can you see the difference?

We've added a third except branch, but this time it **has no exception name specified** – we can say it's **anonymous** or (what is closer to its actual role) it's the **default**. You can expect that when an exception is raised and there is no except branch dedicated to this exception, it will be handled by the default branch.

**Note:**

The default except branch must be the last except branch. Always!

try:

value = int(input('Enter a natural number: '))

print('The reciprocal of', value, 'is', 1/value)

except ValueError:

print('I do not know what to do.')

except ZeroDivisionError:

print('Division by zero is not allowed in our Universe.')

except:

print('Something strange has happened here... Sorry!')

# Some useful exceptions

Let’s discuss in more detail some useful (or rather, the most common) exceptions you may experience.

## ZeroDivisionError

This appears when you try to force Python to perform any operation which provokes division in which the divider is zero, or is indistinguishable from zero. Note that there is more than one Python operator which may cause this exception to raise. Can you guess them all?

Yes, they are: /, //, and %.

## ValueError

Expect this exception when you're dealing with values which may be inappropriately used in some context. In general, this exception is raised when a function (like int() or float()) receives an argument of a proper type, but its value is unacceptable.

## TypeError

This exception shows up when you try to apply a data whose type cannot be accepted in the current context. Look at the example:

short\_list = [1]

one\_value = short\_list[0.5]

You're not allowed to use a float value as a list index (the same rule applies to tuples, too). TypeError is an adequate name to describe the problem, and an adequate exception to raise.

## AttributeError

This exception arrives – among other occasions – when you try to activate a method which doesn't exist in an item you're dealing with. For example:

short\_list = [1]

short\_list.append(2)

short\_list.depend(3)

The third line of our example attempts to make use of a method which isn’t contained in the lists. This is the place where AttributeError is raised.

## SyntaxError

This exception is raised when the control reaches a line of code which violates Python's grammar. It may sound strange, but some errors of this kind cannot be identified without first running the code. This kind of behavior is typical of interpreted languages – the interpreter always works in a hurry and has no time to scan the whole source code. It is content with checking the code which is currently being run. An example of such a category of issues will be presented very soon.

It's a bad idea to handle this exception in your programs. You should produce code that is free of syntax errors, instead of masking the faults you’ve caused.

# Why you can’t avoid testing your code

Although we're going to wrap up our *exceptional* considerations here, don't think it's all Python can offer to help you with begging for forgiveness. Python's exception machinery is far more complex, and its capabilities allow you to build expanded error handling strategies. We'll return to these issues – we promise. Feel free to conduct your experiments and to dive into exceptions yourself.

Now we want to tell you about the second side of the never-ending struggle with errors – the inevitable destiny of a developer's life. As you are not able to avoid making bugs in your code, you must always be ready to seek out and destroy them. Don't bury your head in the sand – ignoring errors won't make them disappear.

An important duty for developers is to test the newly created code, but you must not forget that testing isn't a way to prove that the code is error-free. Paradoxically, the only proof testing can provide is that your code contains errors. Don’t think you can relax after a successful test.

The second important aspect of software testing is strictly psychological. It's a truth known for years that authors – even those who are reliable and self-aware – **aren't able to objectively evaluate and verify their works**.

This is why each novelist needs an editor and each programmer needs a tester. Some say – a little spitefully but truthfully – that developers test the code to show their perfection, not to find problems that may frustrate them. Testers are free of such dilemmas, and this is why their work is more effective and profitable.

Of course, this doesn't absolve you from being attentive and careful. Test your code as best you can. Don't make the testers' work too easy.

Your primary duty is to **ensure that you’ve checked all execution paths** your code can go through. Does that sound mysterious? Nothing of the kind!

## Tracing the execution paths

Now look at the code in the editor. Suppose you've just finished writing it.

There are three independent execution paths in the code – can you see them? They are determined by the if-elif-else statements. Of course, the execution paths can be built by many other statements, like loops, or even try-except blocks.

If you're going to test your code fairly and you want to sleep soundly and to dream without nightmares (nightmares about bugs can be devastating for a developer’s performance) you are obliged to prepare a test data set that will force your code to negotiate all possible paths.

In our example, the set should contain at least three float values: one positive, one negative, and zero.

temperature = float(input('Enter current temperature:'))

if temperature > 0:

print("Above zero")

elif temperature < 0:

print("Below zero")

else:

print("Zero")

# When Python closes its eyes

Such a test is crucial. We want to show you why you mustn't skip it. Look at the code in the editor.

We intentionally introduced an error into the code – we hope your watchful eyes noticed it immediately. Yes, we removed just one letter and in effect, the valid print() function invocation turns into the obviously invalid clause "prin()". There is no such function as "prin()" in our program's scope, but is it really obvious for Python?

Run the code and enter 0.

As you can see, the code finishes its execution without any obstacles.

How is that possible? Why does Python **overlook** such an evident developer mistake?

Can you find the answers to these fundamental questions?

temperature = float(input('Enter current temperature:'))

if temperature > 0:

print("Above zero")

elif temperature < 0:

prin("Below zero")

else:

print("Zero")

# Tests, testing, and testers

The answer is simpler than you may expect, and a bit disappointing, too. Python – as you know for sure – is an **interpreted** language. This means that the source code is parsed and executed at the same time. Consequently, Python may not have time to analyze the code lines which aren't subject to execution. As an old developer's saying states: *"it's a feature, not a bug"* (please don't use this phrase to justify your code's weird behavior).

Do you understand now why passing through all execution paths is so vital and inevitable?

Let’s assume that you complete your code and the tests you've made are successful. You deliver your code to the testers and – fortunately! – they found some bugs in it. We’re using the word *"fortunately"* completely consciously. You need to accept that, firstly, testers are the developer’s best friends – don't treat the bugs they discover as an offense or a malignancy; and, secondly, each bug the testers find is a bug that won't affect the users. Both factors are valuable and worth your attention.

You already know that your code contains a bug or bugs (the latter is more likely). How do you locate them and how do you fix your code?

## Bug vs. debug

The basic measure a developer can use against bugs is – unsurprisingly – a **debugger**, while the process during which bugs are removed from the code is called **debugging**. According to an old joke, debugging is a complicated mystery game in which you are simultaneously the murderer, the detective, and – the most painful part of the intrigue – the victim. Are you ready to play all these roles? Then you must arm yourself with a debugger.

A debugger is a specialized piece of software that can control how your program is executed. Using the debugger, you can execute your code line-by-line, inspect all the variables' states and change their values on demand without modifying the source code, stop program execution when certain conditions are or aren't met, and do lots of other useful tasks.



We can say that every IDE is equipped with a more or less advanced debugger. Even IDLE has one, although you may find its handling a bit complicated and troublesome. If you want to make use of IDLE's integrated debugger, you should activate it using the “Debug” entry in the main IDLE window menu bar. It's the start point for all debugger facilities.

[Click here to see the screenshots](https://www.cs.uky.edu/~keen/help/debug-tutorial/debug.html) that show the IDLE debugger during a simple debugging session. (Thank you, University of Kentucky!)

You can see how the debugger visualizes variables and parameter values, and note the call stack which shows the chain of invocations leading from the currently executed function to the interpreter level.

If you want to know more about the IDLE debugger, consult [the IDLE documentation](https://docs.python.org/3/library/idle.html).

***print* debugging**

This form of debugging, which can be applied to your code using any kind of debugger, is sometimes called **interactive debugging**. The meaning of the term is self-explanatory – the process needs your (the developer's) interaction to be performed.

Some other debugging techniques can be used to hunt bugs. It's possible that you aren't able or don't want to use a debugger (the reasons may vary). Are you helpless then? Absolutely not!

You may use one of the simplest and the oldest (but still useful) debugging tactics known as **print debugging**. The name speaks for itself – you just insert several additional print() invocations inside your code to output data which illustrates the path your code is currently negotiating. You can output the values of the variables which may affect the execution.

These printouts may output meaningful text like *"I am here"*, *"I entered the foo() function"*, *"The result is 0"*, or they may contain sequences of characters that are legible only to you. Please don't use obscene or indecent words for the purpose, even though you may feel a strong temptation – your reputation can be ruined in a moment if these antics leak to the public.

As you can see, this kind of debugging isn't really interactive at all, or is interactive only to a small extent, when you decide to apply the input() function to stop or delay code execution.

After the bugs are found and removed, the additional printouts may be commented out or removed – it's up to you. Don't let them be executed in the final code – they may confuse both testers and users, and bring bad karma down upon you.



# Some useful tips

Here are some tips which may help you to find and eliminate the bugs. None of them is either ultimate or definitive. Use them flexibly and rely on your intuition. Don't believe yourself – check everything twice.

1. **Try to tell someone** (for example, your friend or coworker) what your code is expected to do and how it actually behaves. Be concrete and don't omit details. Answer all questions your helper asks. You'll likely realize the cause of the problem while telling your story, as speaking activates these parts of your brain which remain idle during coding. If no human can help you with the problem, use a yellow rubber duck instead. We're not kidding – consult the Wikipedia article to learn more about this commonly used technique: [Rubber Duck Debugging](https://en.wikipedia.org/wiki/Rubber_duck_debugging).

1. **Try to isolate the problem**. You can extract the part of your code that is suspected of being responsible for your troubles and run it separately. You can comment out parts of the code that obscure the problem. Assign concrete values to variables instead of reading them from the input. Test your functions by applying predictable argument values. Analyze the code carefully. Read it aloud.

1. If the bug has appeared recently and didn't show up earlier, **analyze all the changes you've introduced into your code** – one of them may be the reason.

1. **Take a break**, drink a cup of coffee, take your dog and go for a walk, read a good book for a moment or two, make a phone call to your best friend – you'll be surprised how often it helps.

1. **Be optimistic** – you'll find the bug eventually; we promise you this.

# Unit testing – a higher level of coding

There is also one important and widely used programming technique that you will have to adopt sooner or later during your developer career – it's called unit testing. The name may a bit confusing, as it's not only about testing the software, but also (and most of all) about how the code is written.

To make a long story short – unit testing assumes that tests are inseparable parts of the code and preparing the test data is an inseparable part of coding. This means that when you write a function or a set of cooperating functions, you're also obliged to create a set of data for which your code's behavior is predictable and known.

Moreover, you should equip your code with an interface that can be used by an automated testing environment. In this approach, any amendment made to the code (even the least significant) should be followed by the execution of all the unit tests accompanied by your source.

To standardize this approach and make it easier to apply, Python provides a dedicated module named unittest. We're not going to discuss it here – it's a broad and complex topic.

Therefore, we’ve prepared a separate course and certification path for this subject. It is called “Testing Essentials with Python”, and we invite you to participate in it.

See you soon!

# Key takeaways – Exceptions

1. In Python, there is a distinction between two kinds of errors:

* **syntax errors** (parsing errors), which occur when the parser comes across a statement that is incorrect. For example:

Trying to execute the following line:

print("Hello, World!)

will cause a *SyntaxError*, and result in the following (or similar) message being displayed in the console:

File "main.py", line 1

print("Hello, World!)

^

SyntaxError: EOL while scanning string literal

**output**

Pay attention to the arrow – it indicates the place where the Python parser has run into trouble. In our case, it's the missing double quote. Did you notice it?

* **exceptions**, which occur even when a statement/expression is syntactically correct; these are the errors that are detected during execution when your code results in an error which is not *uncoditionally fatal*. For example:

Trying to execute the following line:

print(1/0)

will cause a *ZeroDivisionError* exception, and result in the following (or similar) message being displayed in the console:

Traceback (most recent call last):

File "main.py", line 1, in

print(1/0)

ZeroDivisionError: division by zero

**output**

Pay attention to the last line of the error message – it actually tells you what happened. There are many different types of exceptions, such as *ZeroDivisionError*, *NameError*, *TypeError*, and many more; and this part of the message informs you of what type of exception has been raised. The preceding lines show you the context in which the exception has occured.

2. You can "catch" and handle exceptions in Python by using the *try-except* block. So, if you have a suspicion that any particular snippet may raise an exception, you can write the code that will gracefully handle it, and will not interrupt the program. Look at the example:

while True:

try:

number = int(input("Enter an integer number: "))

print(number/2)

break

except:

print("Warning: the value entered is not a valid number. Try again...")

The code above asks the user for input until they enter a valid integer number. If the user enters a value that cannot be converted to an int, the program will print Warning: the value entered is not a valid number. Try again..., and ask the user to enter a number again. What happens in such a case?

1. The program enters the *while* loop.
2. The try block/clause is executed. The user enters a wrong value, for example: hello!.
3. An exception occurs, and the rest of the try clause is skipped. The program jumps to the except block, executes it, and then continues running after the *try-except* block.

If the user enters a correct value and no exception occurs, the subsequent instructions in the *try* block are executed.

3. You can handle multiple exceptions in your code block. Look at the following examples:

while True:

try:

number = int(input("Enter an int number: "))

print(5/number)

break

except ValueError:

print("Wrong value.")

except ZeroDivisionError:

print("Sorry. I cannot divide by zero.")

except:

print("I don't know what to do...")

You can use multiple *except* blocks within one *try* statement, and specify particular exception names. If one of the except branches is executed, the other branches will be skipped. Remember: you can specify a particular built-in exception only once. Also, don't forget that the **default** (or generic) exception, that is the one with no name specified, should be placed **at the bottom of the branch** (use the more specific exceptions first, and the more general last).

You can also specify and handle multiple built-in exceptions within a single *except* clause:

while True:

try:

number = int(input("Enter an int number: "))

print(5/number)

break

except (ValueError, ZeroDivisionError):

print("Wrong value or No division by zero rule broken.")

except:

print("Sorry, something went wrong...")

4. Some of the most useful Python built-in exceptions are: *ZeroDivisionError*, *ValueError*, *TypeError*, *AttributeError*, and *SyntaxError*. One more exception that, in our opinion, deserves your attention is the *KeyboardInterrupt* exception, which is raised when the user hits the interrupt key (CTRL-C or Delete). Run the code above and hit the key combination to see what happens.

To learn more about the Python built-in exceptions, consult the official [Python documentation](https://docs.python.org/3/library/exceptions.html#bltin-exceptions).

5. Last but not least, you should remember about testing and debugging your code. Use such debugging techniques as *print* debugging; if possible – ask someone to read your code and help you to find bugs in it or to improve it; try to isolate the fragment of code that is problematic and susceptible to errors: **test your functions** by applying predictable argument values, and try to **handle** the situations when someone enters wrong values; **comment out** the parts of the code that obscure the issue. Finally, take breaks and come back to your code after some time with a fresh pair of eyes.

**Exercise**

What is the output of the following program if the user enters 0?

try:

value = int(input("Enter a value: "))

print(value/value)

except ValueError:

print("Bad input...")

except ZeroDivisionError:

print("Very bad input...")

except:

print("Booo!")

Check

The program will output: Very bad input....

**PROJECT**

## Estimated time

30-120 minutes

## Level of difficulty

Medium/Hard

## Objectives

* perfecting the student's skills in using Python for solving complex problems;
* integrating programming techniques in one program consisting of many various parts.

## Scenario

Your task is to write **a simple program which pretends to play *tic-tac-toe* with the user**. To make it all easier for you, we've decided to simplify the game. Here are our assumptions:

* the computer (i.e., your program) should play the game using 'X's;
* the user (e.g., you) should play the game using 'O's;
* the first move belongs to the computer − it always puts its first 'X' in the middle of the board;
* all the squares are numbered row by row starting with 1 (see the example session below for reference)
* the user inputs their move by entering the number of the square they choose − the number must be valid, i.e., it must be an integer, it must be greater than 0 and less than 10, and it cannot point to a field which is already occupied;
* the program checks if the game is over − there are four possible verdicts: the game should continue, the game ends with a tie, you win, or the computer wins;
* the computer responds with its move and the check is repeated;
* don't implement any form of artificial intelligence − a random field choice made by the computer is good enough for the game.

The example session with the program may look as follows:

+-------+-------+-------+

| | | |

| 1 | 2 | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | 6 |

| | | |

+-------+-------+-------+

| | | |

| 7 | 8 | 9 |

| | | |

+-------+-------+-------+

Enter your move: 1

+-------+-------+-------+

| | | |

| O | 2 | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | 6 |

| | | |

+-------+-------+-------+

| | | |

| 7 | 8 | 9 |

| | | |

+-------+-------+-------+

+-------+-------+-------+

| | | |

| O | X | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | 6 |

| | | |

+-------+-------+-------+

| | | |

| 7 | 8 | 9 |

| | | |

+-------+-------+-------+

Enter your move: 8

+-------+-------+-------+

| | | |

| O | X | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | 6 |

| | | |

+-------+-------+-------+

| | | |

| 7 | O | 9 |

| | | |

+-------+-------+-------+

+-------+-------+-------+

| | | |

| O | X | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | X |

| | | |

+-------+-------+-------+

| | | |

| 7 | O | 9 |

| | | |

+-------+-------+-------+

Enter your move: 4

+-------+-------+-------+

| | | |

| O | X | 3 |

| | | |

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| 7 | O | 9 |

| | | |

+-------+-------+-------+

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| 7 | O | 9 |

| | | |

+-------+-------+-------+

Enter your move: 7

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| O | O | 9 |

| | | |

+-------+-------+-------+

You won!

## Requirements

Implement the following features:

* the board should be stored as a three-element list, while each element is another three-element list (the inner lists represent rows) so that all of the squares may be accessed using the following syntax:  
    
  board[row][column]
* each of the inner list's elements can contain 'O', 'X', or a digit representing the square's number (such a square is considered free)
* the board's appearance should be exactly the same as the one presented in the example.
* implement the functions defined for you in the editor.

Drawing a random integer number can be done by utilizing a Python function called randrange(). The example program below shows how to use it (the program prints ten random numbers from 0 to 8).

Note: the from-import instruction provides access to the randrange function defined within an external Python module callled random.

from random import randrange

for i in range(10):

print(randrange(8))

def display\_board(board):

# The function accepts one parameter containing the board's current status

# and prints it out to the console.

def enter\_move(board):

# The function accepts the board's current status, asks the user about their move,

# checks the input, and updates the board according to the user's decision.

def make\_list\_of\_free\_fields(board):

# The function browses the board and builds a list of all the free squares;

# the list consists of tuples, while each tuple is a pair of row and column numbers.

def victory\_for(board, sign):

# The function analyzes the board's status in order to check if

# the player using 'O's or 'X's has won the game

def draw\_move(board):

# The function draws the computer's move and updates the board.

# Congratulations! You have completed Module 4.

Well done! You've reached the end of Module 4 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and become familiar with in Module 4:

* the defining and using of functions − their rationale, purpose, conventions, and traps;
* the concept of passing arguments in different ways and setting their default values, along with the mechanisms of returning the function's results;
* name scope issues;
* new data aggregates: tuples and dictionaries, and their role in data processing;
* the concept of exceptions, and the importance of code testing and debugging.

You are now ready to take the module quiz and attempt the final challenge: Module 4 Test, which will help you gauge what you've learned so far.