Create your own functions

Aksel Hiorth

University of Stavanger

Jan 11, 2024

Contents

1 What is a function?

As explained in the introduction, a function is several lines of code that perform a specific task. In many ways you can think of a function as a recipe, e.g. a cake recipe. To make a cake we need a certain input, eggs, flour, sugar, chocolate, then we follow a specific set of operations to produce the cake. A function in Python operates in the same way, it takes something as input (different variables), follow certain steps and returns a product (the cake).

You have already used a lot built in functions in Python, such as `print()`, `pandas.DataFrame()`, etc. We can define our own functions using the `def` keyword, it best illustrated with some examples

```
def my_func():
    print('My first function')
my_func()
```

You can pass arguments to the function, to make them more general

```
def my_func(name):
    print('My name is :' + name)
my_func('Bob')
```

2 When to define a function?

When to use functions? There is no particular rule, *but whenever you start to copy and paste code from one place to another, you should consider to use a function*. Functions makes the code easier to read. It is not easy to identify which part of a program is a good candidate for a function, it requires skill and experience. Most likely you will end up changing the function definitions as your program develops.

2.1 What is a good function?

Even if you only write code for yourself, you will quickly forget what the code does. That is why it is so important to write functions that are good, below are some suggestions

A function:

- should have a descriptive name.
- should only do one thing.
- should only be dependent on its input argument and give the same answer independent on how many times it is being called.
- should contain a doc string (see below for examples).

- should have a descriptive name, use small letters with an underscore to separate words.

DRY - Do not Repeat Yourself [1].

If you need to change the code in more than one place to extend it, you may forget to change everywhere and introduce bugs. The DRY principle also applies to *knowledge sharing*, it is not only about copy and paste code, but knowledge should only be represented in one place.

Exercise 1: Create a function from the following code

When I normally code I usually first write the code, to see that I am able to achieve what I want, and then I create a function based on that code.

Question: Create a function from the following code

```
# this code replace space and slash in names
name='16/1-12 Troldhaugen'
chars=[" ", "/"]
new_chars=["","_"]
new_name=name
for ch,nch in zip(chars,new_chars):
    new_name = new_name.replace(ch, nch)
```

```
def replace_chars(name):
    '''
    name: A string
    returns input strings where space is removed and slash is
    replaced with underscore
    '''
    chars=[" ", "/"]
    new_chars=["","_"]
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
name='16/1-12 Troldhaugen'
replace_chars(name) #prints 16_1-12Troldhaugen
```

Solution:

Docstring.

In the example above we added some text just after `def` statement. This is a *docstring*. A docstring is to tell people what the code does without

them having to read the code. If you type `help(replace_chars)`, Python will print out the docstring.

3 Making functions more general

In the above example, the function is already quite useful, but if you later decide that you e.g. do not want to remove spaces from names, you would have to write a new function. However we can achieve a more general function, by using *default arguments*.

```
def replace_chars(name,chars=[" ","/"],new_chars=["","_"]):
    '''
    name: A string
    returns input strings where space is removed and slash is
    replaced with underscore
    '''
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
```

Now you can use the same call signature as before `replace_chars(name)`, if you want to only replace `/`, you write

```
name='16/1-12 Troldhaugen'
replace_chars(name,["/"],["_"]) # prints 16_1-12 Troldhaugen
```

Positional arguments.

The variable `name` in the function definition of `replace_chars` is called a *positional argument*. On the other hand `chars=[" ","/"]` is called a *default argument*. In Python default argument, must always come *after* positional arguments. Hence, it is not allowed to write `def replace_chars(chars=[" ","/"],new_chars=["","_"],`

4 Improving robustness of functions

As time goes, you start to forget what a function does, and you can start using it wrong. A typical situation in the above example is that we could mix up the order of `new_chars` and `chars`, i.e. we do `replace_chars(name,["_"],["/"])` instead of `replace_chars(name,["/"],["_"])`, which would give the opposite effect. Python has a very neat syntax to avoid this behavior, by adding a `*` in the argument list

```
def replace_chars(name,*, chars=[" ","/"],new_chars=["","_"]):
    '''
    name: A string
    returns input strings where space is removed and slash is
    replaced with underscore
    '''
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
```

You can still call the function as before `replace_chars(name)`, but if you try to do

```
name='16/1-12 Troldhaugen'
replace_chars(name,["/"],["_"])
```

You will get an error

Warning.

```
TypeError: replace_chars() takes 1 positional argument but 3 were given
```

This error might be a bit hard to interpret, but basically `name` is a positional argument according to the definition and the two next arguments are default arguments. When we use the `*` in the function definition we have to explicitly enter the variable name of the default arguments

```
name='16/1-12 Troldhaugen'
replace_chars(name,chars=["/"],new_chars=["_"])
```

By forcing the user to use `chars` and `new_chars` it becomes less probable to mix them up.

5 Assert, raise and try statements

Still there are plenty of things that can go wrong with our function, and you could code for ever to if you want to catch every error. In many cases you would like to catch errors as quickly as possible, to help users to discover the errors. Lets look at an example, lets say we wrongly use our function

```
replace_chars(2)
```

We get

Warning.

```
new_name = name
           for ch,nch in zip(chars,new_chars):
---->          new_name = new_name.replace(ch, nch)
           return new_name

AttributeError: 'int' object has no attribute 'replace'
```

Note that the errors only occurs when we call `new_name.replace`, because we send in an integer and an integer does not have a function named `replace()`. The errors happens because the function `replace_chars` is used wrongly, thus it would be much better if we could catch the error at a very early stage and give the user an error message.

5.1 Try and Except

The most lazy, and not recommended, way of improving our function is to declare our function as

```
def replace_chars(name,*,chars=[" ", "/"],new_chars=["","_"]):
    ''' replace Norwegian characters and space in names'''
    try:
        new_name = name
        for ch,nch in zip(chars,new_chars):
            new_name = new_name.replace(ch, nch)
        return new_name
    except:
        print('Something went wrong in replace_chars')

replace_chars(2)# prints Something went wrong in replace_chars
```

Now the code prints out a message if something is not working. There are several drawbacks

1. The code is still running, even if something went wrong, the program should stop with an error message
2. We do not know exactly where something went wrong, it could be that `name` is not a string, but it could also be e.g. that the user enters a different length for `chars` and `new_chars`

5.2 Raise

We can raise errors by using the keyword `raise`. The `raise` keyword needs to be followed by a function from the `BaseExceptionclass`, typically you might use `raise Exception('Something went wrong in replace_chars')`

```
def replace_chars(name,*,chars=[" ", "/"],new_chars=["","_"]):
    ''' replace Norwegian characters and space in names'''
    try:
        new_name = name
        for ch,nch in zip(chars,new_chars):
            new_name = new_name.replace(ch, nch)
        return new_name
    except:
        raise Exception('Something went wrong in replace_chars')

replace_chars(2)# prints Something went wrong in replace_chars
```

Running the following code

```
replace_chars(2)
```

We get

Warning.

```
 except:
--->          raise Exception('Something went wrong in replace_chars')
     replace_chars(2)

Exception: Something went wrong in replace_chars
```

As compared to before we raise the exception at when our function is called and not when the `string.replace()` function is called.

Exceptions should be specific.

When raising errors, we should try to be as specific as possible to help the user as much as possible.

Let us extend our function using more specific raises

```
def replace_chars(name,*,chars=[" ", "/"],new_chars=["","_"]):
    ''' replace Norwegian characters and space in names'''
    if type(name)!=str:
        raise ValueError('replace_chars: name must be a string')
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
replace_chars(2)# ValueError: replace_chars: name must be a string
```

Exercise: Improve `replace_chars` using raise.

Question: How can we improve `replace_chars` to make sure that the length of `chars` and `new_chars` are of equal length?

```
def replace_chars(name,*,chars=[" ", "/"],new_chars=["","_"]):
    ''' replace Norwegian characters and space in names'''
    if type(name)!=str:
        raise ValueError('replace_chars: name must be a string')
    if len(chars) != len(new_chars):
        raise ValueError('replace_chars: chars and new_chars must same size')
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
replace_chars('2',chars=["/"])# ValueError: replace_chars: chars and new_chars must same size
```

Solution:

6 Using assert to test our code

When developing code it is extremely useful to design tests that checks that our code does what it is supposed to do. This is mainly done to make sure that the expected behavior of a function does not changes over time. For our small function we can use the specific keyword `assert`. The syntax is `assert <condition>,<error message>`

assert only works in debug mode.

If you for some reason compile your code or in other ways turn off the debug option in Python, assert will not work.

```
def test_replace_chars():
    assert replace_chars('  ') == ''
    assert replace_chars('//') == '__'
    assert replace_chars('G O//O D') =='GO__OD'
test_replace_chars()
```

Each time we start working and stop working on our code , we run all tests that we have defined. If nothing fails we know that no one has introduced errors before we start coding, and that we have not made any changes.

7 Advanced topics: Passing functions to functions

In many cases, you would also pass a function to another function to make your code more modular. Lets say we want to calculate the derivative of $\sin x$, using the most basic definition of a derivative $f'(x) = f(x + \Delta x) - f(x)/\Delta x$, we could do it as

```
def derivative_of_sine(x,delta_x):
    ''' returns the derivative of sin x '''
    return (np.sin(x+delta_x)-np.sin(x))/delta_x

print('The derivative of sinx at x=0 is :', derivative_of_sine(0,1e-3))
```

If we would like to calculate the derivative at multiple points, that is straightforward since we have used the Numpy version of $\sin x$.

```
x=np.array([0,.5,1])
print('Derivative of sinx at x=0,0.5,1 is :', derivative_of_sine(x,1e-3))
```

The challenge with our implementation is that if we want to calculate the derivative of another function we have to implement the derivative rule again for that function. It is better to have a separate function that calculates the derivative

```
def f(x):
    return np.sin(x)

def df(x,f,delta_x=1e-3):
    ''' returns the derivative of f '''
    return (f(x+delta_x)-f(x))/delta_x
print('Derivative of sinx at x=0 is :', df(0,f))
```

Note also that we have put `delta_x=1e-3` as a *default argument*. Default arguments have to come at the end of the argument lists, `df(x,delta_x=1e-3,f)` is not allowed. All of this looks well, but what you would experience is that your functions would not be as simple as $\sin x$. In many cases your functions need additional arguments to be evaluated e.g.:

```
def s(t,s0,v0,a):
    '''
    t  : time
    s0 : initial starting point
    v0 : initial velocity
    a  : acceleration
    returns the distance traveled
    '''
    return s0+v0*t+a*t*t*0.5 #multiplication (0.5)is general faster
                                    #than division (2)
```

How can we calculate the derivative of this function? If we try to do `df(1,s)` we will get the following message

```
TypeError: s() missing 3 required positional
           arguments: 's0', 'v0', and 'a'
```

This happens because the `df` function expect that the function we send into the argument list has a call signature `f(x)`. What many people do to avoid this error is to use global variable, that is to define `s0, v0`, and `a` at the top of the code. This is not always the best solution. Python has a special variable `*args` which can be used to pass multiple arguments to your function, thus if we rewrite `df` like this

```
def df(x,f,*args,delta_x=1e-3):
    ''' returns the derivative of f '''
    return (f(x+delta_x,*args)-f(x,*args))/delta_x
```

we can do (assuming `s0=0`, `v0=1`, and `a=9.8`)

```
print('The derivative of sinx at x=0 is :', df(0,f))
print('The derivative of s(t) at t=1 is :', df(0,s,0,1,9.8))
```

7.1 Scope of variables

In small programs you would not care about scope, but once you have several functions, you will easily get into trouble if you do not consider the scope of a variable. By scope of a variable we mean where the variable is available, first some simple examples

A variable created inside a function is only available within the function: “

```
def f(x):
    a=10
    b=20
    return a*x+b
```

Doing `print(a)` outside the function will throw an error: `name 'a' is not defined`. What happens if we define variable `a` outside the function?

```
a=2
def f(x):
    a=10
    b=20
    return a*x+b
```

If we first call the function `f(0)`, and then do `print(a)` Python would give the answer `2`, *not* `10`. A *local* variable `a` is created inside `f(x)`, that does not interfere with the variable `a` defined outside the function.

The `global` keyword can be used to pass and access variables in functions: “

```
global a
a=2
def f(x):
    global a
    a=10
    b=20
    return a*x+b
```

In this case `print(a)` *before* calling `f(x)` will give the answer `2` and *after* calling `f(x)` will give `10`.

Use of global variables.

Sometimes global variables can be very useful, and help you to make the code simpler. But make sure to use a *naming convention* for them, e.g. end all the global variables with an underscore. In the example above we would write `global a_`. A person reading the code would then know that all variables ending with an underscore are global, and can potentially be modified by several functions.

7.2 Passing arrays and lists to functions

In the previous section, we looked at some simple examples regarding the scope of variables, and what happened with that variable inside and outside a function. The examples used integer or floats. However in most applications you will pass an array or a list to a function, and then you need to be aware that the behavior is not always would you might expect.

Unexpected behavior.

Sometimes functions do not do what you expect, this might be because the function does not treat the arguments as you might think. The best advice is to make a very simple version of your function and test it for yourself. Is the behavior what you expect? Try to understand why or why not.

Let us look at some examples, and try to understand what is going on and why.

```
x=3
def f(x):
    x = x*2
    return x
print('x =',x)
print('f(x) returns ', f(x))
```

```
print('x is now ', x)
```

In the example above we can use `x=3`, `x=[3]`, `x=np.array([3])`, and after execution `x` is unchanged (i.e. same value as before `f(x)`) was called. Based on what we have discussed before, this is maybe what you would expect, but if we now do

```
x=[3]
def append_to_list(x):
    return x.append(1)
print('x = ',x)
print('append_to_list(x) returns ', append_to_list(x))
print('x is now ', x)
```

(Clearly this function will only work for lists, due to the append command.) After execution, we get the result

```
x = [3]
append_to_list(x) #returns  [3 1], x is now  [3, 1]
```

Even if this might be exactly what you wanted your function to do, why does `x` change here when it is a list and not in the previous case when it is a float? Before we explain this behavior let us rewrite the function to work with Numpy arrays

```
x=np.array([3])
def append_to_np(x):
    return np.append(x,1)
print('x = ',x)
print('append_to_np(x) returns ', append_to_np(x))
print('x is now ', x)
```

The output of this code is

```
x =  np.array([3])
append_to_np(x) #returns  [3 1], x is now  [3]
```

This time `x` was not changed, what is happening here? It is important to understand what is going on because it deals with how Python handles variables in the memory. If `x` contains million of values, it can slow down your program, if we do

```
N=1000000
x=[3]*N
%timeit append_to_list(x)
x=np.array([3]*N)
%timeit append_to_np(x)
```

On my computer I found that `append_to_list` used 76 nano seconds, and `append_to_np` used 512 micro seconds, the Numpy function was about 6000 times slower! To add to the confusion consider the following functions

```
x=np.array([3])
def add_to_np(x):
    x=x+3
    return x

def add_to_np2(x):
    x+=3
    return x
print('x = ',x)
print('add_to_np(x) returns ', add_to_np(x))
print('x is now ', x)

print('x = ',x)
print('add_to_np2(x) returns ', add_to_np2(x))
print('x is now ', x)
```

The output is

```
x =  np.array([3])
add_to_np(x) #returns [6], x is now [3]
x =  np.array([3])
add_to_np2(x) #returns  [6], x is now  [6]
```

In both cases the function returns what you expect, but it has an unexpected (or at least a different) behavior regarding the variable `x`. What about speed?

```
N=10000000
x=np.array([3]*N)
%timeit add_to_np(x)
x=np.array([3]*N)
%timeit add_to_np2(x)
```

`add_to_np` is about twice as slow as `add_to_np2`. In the next section we will try to explain the difference in behavior.

Avoiding unwanted behavior of functions.

The examples in this section are meant to show you that if you pass an array to a function, the array can be altered outside the scope of the function. If this is not what you want, it could lead to bugs that are hard to detect. Thus, if you experience unwanted behavior pick out the part of function involving list or array operations and test one by one in the editor.

7.3 Call by value or call by reference

For anyone that has programmed in C or C++ call by reference or value is something one need to think about constantly. When we pass a variable to a

function there are two choices, should we pass a copy of the variable or should we pass information about where the variable is stored in memory?

Value and reference.

In C and C++ pass by value means that we are making a copy in the memory of the variable we are sending to the function, and pass by reference means that we are sending the actual parameter or more specific the address to the memory location of the parameter. In Python all variables are passed by object reference.

In C and C++ you always tell in the function definition if the variables are passed by value or reference. Thus if you would like a change in a variable outside the function definition, you pass the variable by reference, otherwise by value. In Python we always pass by (object) reference.

Floats and integers. To gain a deeper understanding, we can use the `id` function, the `id` function gives the unique id to a variable. In C this would be the actual memory address, lets look at a couple of examples

```
a=10.0
print(id(a)) #gives on my computer 140587667748656
a += 1
print(id(a)) #gives on my computer 140587667748400
```

Thus, after adding 1 to `a`, `a` is assigned *a new place in memory*. This is very different from C or C++, in C or C++ the variable, once it is created, *always has the same memory address*. In Python this is not the case, it works in the opposite way. The statement `a=10.0`, is executed so that *first* 10.0 is created in memory, secondly `x` is assigned the reference to 10.0. The assignment operator `=` indicates that `a` should point to whatever is on the right hand side. Another example is

```
a=10.0
b=10.0
print(a is b) # prints False
b=a
print(a is b ) # prints True
```

In this case 10.0 is created in two different places in the memory and a different reference is assigned to `a` and `b`. However if we put `b=a`, `b` points to the same object as `a` is pointing on. More examples

```
a=10
b=a
print(a is b) # True
a+=2
```

```
print(a is b) # False
```

When we add 2 to `a`, we actually add 2 to the value of 10, the number 12 is assigned a new place in memory and `a` will be assigned that object, whereas `b` would still points the old object 10.

Lists and arrays. Yous should think of lists and arrays as containers (or a box). If we do

```
x=[0,1,2,3,4]
print(id(x))
x[0]=10
print(id(x)) # same id value as before and x=[10,1,2,3,4]
```

First, we create a list, which is basically a box with the numbers 0, 1, 2, 3, 4. The variable `x` points to *the box*, and `x[0]` points to 0, and `x[1]` to 1 etc. Thus if we do `x[0]=10`, that would be the same as picking 0 out of the box and replacing it with 10, but *the box stays the same*. Thus when we do `print(x)`, we print the content of the box. If we do

```
x=[0,1,2,3,4]
y=x
print(x is y) # True
x.append(10)  # x is now [0,1,2,3,4,10]
print(y)      # y=[0,1,2,3,4,10]
print(x is y) # True
```

What happens here is that we create a box with the numbers 0, 1, 2, 3, 4, `x` is referenced that box. Next, we do `y=x` so that `y` is referenced the *same box* as `x`. Then, we add the number 10 to that box, and `x` and `y` still points to the same box.

Numpy arrays behave differently, and that is basically because if we want to add a number to a Numpy array we have to do `x=np.array(x,10)`. Because of the assignment operator `=` , we take the content of the original box add 10 and put it into a *new* box

```
x=np.array([0,1,2,3,4])
y=x
print(x is y)     # True
x=np.append(x,10) # x is now [0,1,2,3,4,10]
print(y)          # y=[0,1,2,3,4]
print(x is y)     # False
```

The reason for this behavior is that the elements in Numpy arrays (contrary to lists) have to be continuous in the memory, and the only way to achieve this is to create a new box that is large enough to also contain the new number. This also explains that if you use the `np.append(x,some_value)` inside a function where `x` is large it could slow down your code, because the program has to delete

`x` and create a new very large box each time it would want to add a new element. A better way to do it is to create `x` *large enough* in the beginning, and then just assign values `x[i]=a`.

7.4 Mutable and immutable objects

What we have explained in the previous section is related to what is known as mutable and immutable objects. These terms are used to describe objects that have an internal state that can be changed (mutable), and objects that have an internal state that cannot be changed after they have been created. Example of mutable objects are lists, dictionaries, and arrays. Examples of immutable objects are floats, ints, tuples, and strings. Thus if we create the number 10 its value cannot be changed (and why would we do that?). Note that this is *not the same as saying that* `x=10` and that the internal state of `x` cannot change, this is *not* true. We are allowed to make `x` reference another object. If we do `x=10`, then `x is 10` will give true and they will have the same value if we use the `id` operator on `x` and `10`. If we later say that `a=11` then `a is 10` will give false and `id(a)` and `id(10)` give different values, but * `id(10)` will have the same value as before*.

Lists are mutable objects, and once a list is created, we can change the content without changing the reference to that object. That is why the operations `x=[]` and `x.append(1)`, does not change the id of x, and also explain that if we put `y=x`, `y` would change if `x` is changed. Contrary to immutable objects if `x=[]`, and `y=[]` then `x is y` will give false. Thus, whenever you create a list it will be an unique object.

A final tip.

You are bound to get into strange, unwanted behavior when working with lists, arrays and dictionaries (mutable) objects in Python. Whenever, you are unsure, just make a simple version of your lists and perform some of the operations on them to investigate if the behavior is what you want.

Finally, we show some "unexpected" behavior, just to demonstrate that it is easy to do mistakes and one should always test code on simple examples.

```
x_old=[]
x  = [1, 2, 3]
x_old[:] = x[:] # x_old = [1, 2, 3]
x[0] = 10
print(x_old) # "expected" x_old = [10, 2, 3], actual [1, 2, 3]
```

Comment: We put the *content* of the `x` container into `x_old`, but `x` and `x_old` reference different containers.

```
def add_to_list(x,add_to=[])
    add_to.append(x)
    return add_to

print(add_to_list(1)) # "expected" [1] actual [1]
print(add_to_list(2)) # "expected" [2] actual [1, 2]
print(add_to_list(3)) # "expected" [3] actual [1, 2, 3]
```

Comment: `add_to=[]` is a default argument and it is created once when the program starts and not each time the function is called.

```
x = [10]
y = x
y = y + [1]
print(x, y) # prints [10] [10, 1]

x = [10]
y = x
y += [1]
print(x, y) # prints [10, 1] [10, 1]
```

Comment: In the first case `y + [1]` creates a new object and the assignment operator `=` assign `y` to that object, thus `x` stays the same. In the second case the `+=` adds `[1]` to the `y` container without changing the container, and thus `x` also changes.

References

[1] David Thomas and Andrew Hunt. *The Pragmatic Programmer: Your Journey to Mastery*. Addison-Wesley Professional, 2019.

Index