Create your own functions

Aksel Hiorth

University of Stavanger

Jan 17, 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 built in functions in Python, such as `print()`, `pandas.DataFrame()`, etc.

1.1 How to create a function in Python

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() #My first function
```

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

```
def greeting(name):
    print('My name is :', name)
greeting('Bob') # My name is Bob
```

It is also common practice to add a docstring to help people better understand what the function does.

```
def greeting(name):
    '''
    prints out a greeting
    '''
    print('My name is :', name)
greeting('Bob') # My name is Bob
```

If we do `help(greeting)`, Python will print out our docstring. A function can also return something

```
def add(x,y):
    '''
    adds two numbers
    '''
    return x+y
add(5,6) # 11
```

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 docstring (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.

3 Special use of functions

Before proceeding, and discuss how you best can build your own functions. It is worth noting that to use some libraries effectively, you sometimes need to interact with the library via functions. Let us consider a simple example, assume that you have a DataFrame that contains white spaces or some other symbols you would like to remove. Then you can define your own function that does what you want on a single element and use the `apply()` function in Pandas

```
# first create a simple DataFrame
dict={'col1':['aa', ' aa ', 'a b c '],'col2':[1,2,3]}
df=pd.DataFrame(dict)
print(df)
def remove_space(x):# only works for single elements
```

```
    return x.strip()
df['col1']=df['col1'].apply(remove_space)
print(df)
```

Note: There already exists a function in Pandas for removing spaces (`df['col1'].str.strip()`), but with the above example you can easily extend the function to do other things.

4 Pythons lambda function

The function defined in the example above is relatively small, and Python has a special syntax, that makes it possible to define small functions in one line. The functions can only have one expression. The syntax is `lambda <arguments>: <expression>`. Below are some examples

```
#remove white space
remove_space=lambda x : x.strip()
remove_space('  a a ')
```

The neat thing is that you can pass a lambda function directly to Pandas `apply()` function, without naming it

```
df['col1'].apply(lambda x: x.strip())
```

Another example from here[1].

```
# check if number is even or odd
result = lambda x : f"{x} is even" if x %2==0 else f"{x} is odd"
print(result(12)) # even
print(result(13)) # odd
```

5 Exercise: Create a function from the following code

It is a good exercise to take code you have already written and create one or several functions 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
```

[1] https://www.geeksforgeeks.org/how-to-use-if-else-elif-in-python-lambda-functions/

```
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
```

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.

6 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=["","`

7 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. This is also why we should try and use variable names like `chars` and `new_chars` that are descriptive.

8 Assert, raise and try statements

Still there are plenty of things that can go wrong with our function. In many cases you would like to catch errors as quickly as possible, to help users to discover where the errors occurs. 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 used an integer as an argument. An integer does not have a function named `replace()`, but the errors happens earlier because we used the function `replace_chars` 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.

8.1 Try and Except

Try and except, is actually as simple as it sounds. It is a way to tell Python to try a piece of code, if the piece of code fails, we move to the except statement. Following this a lazy, and crude 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`

8.2 Raise

We can raise errors by using the keyword `raise`. The `raise` keyword needs to be followed by a function from the `BaseExceptionclass`[2], 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

[2] https://docs.python.org/3/library/exceptions.html#exception-hierarchy

```
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 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
```

9 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.

References

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

Index