Functions

Last updated on 2024-08-24 | Edit this page

Download Chapter notebook (ipynb)

OVERVIEW

Questions

- What are functions?
- How are functions created?
- What are optional arguments?
- What makes functions so powerful?

Objectives

- Understand how to develop and utilise functions.
- Understanding different ways of creating functions.
- Explaining input arguments.
- Understanding the interconnectivity of functions.

Function to create a dictionary

Transcription Function

This chapter assumes that you are familiar with the following concepts in Python:

PREREQUISITE

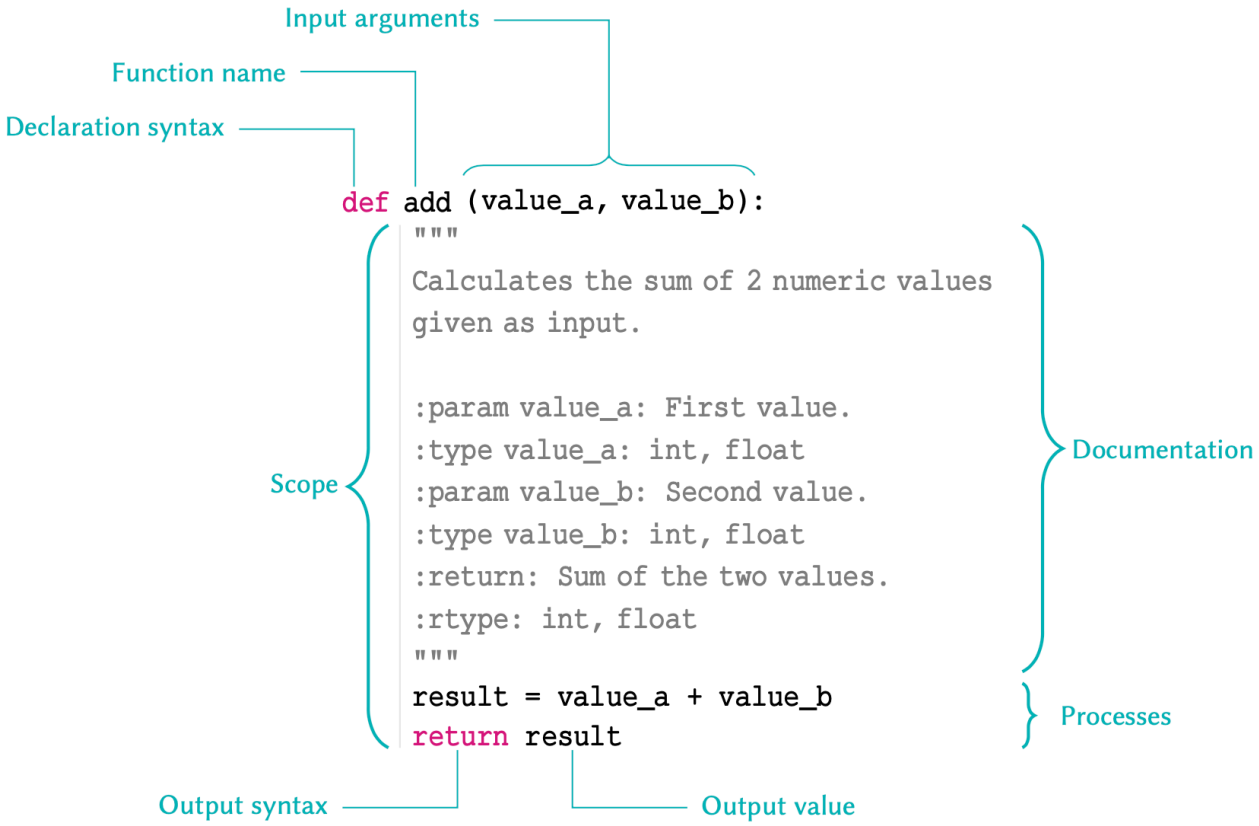
- Mathematical Operation
- Indentation Rule
- Conditional Statements
- Arrays
- Loops and Iterations

Functions

Defining Functions

In programming, functions are individual units or blocks of code that incorporate and perform specific tasks in a sequence defined and written by the programmer. As we learned in the first chapter (on outputs), a function usually takes in one or several variables or values, processes them, and produces a specific result. The variable(s) given to a function and those produced by it are referred to as *input arguments*, and *outputs* respectively.

There are different ways to create functions in Python. In the L2D, we will be using `def` to implement our functions. This is the simplest and most common method for declaring a function. The structure of a typical function defined using `def` is as follows:



REMEMBER

There are several points to remember relative to functions:

- The name of a function follows same principles as that of any other variable as discussed in variable names. The name must be in *lower-case* characters.
- The input arguments of a function — e.g. value_a and value_b in the above example; are essentially variables whose *scope* is the function. That is, they are *only* accessible *within* the function itself, and not from anywhere else in the code.
- Variables defined within a function, should *never* use the same name as variables defined outside of it; or they may override each other.
- A function declared using `def` should *always* be terminated with a `return` syntax. Any values or variables that follow `return` are regarded as the function's output.
- If we do not specify a return value, or fail to terminate a function using `return` altogether, the Python interpreter will automatically terminate that function with an implicit `return None`. Being an *implicit* process, this is generally regarded as a bad practice and should be avoided.

We implement functions to avoid repetition in our code. It is important that a function is *only* performing a very specific task, so that it can be context-independent. You should therefore avoid incorporating separable tasks inside a single function.

INTERESTING FACT

Functions are designed to perform specific tasks. That is why in the majority of cases, they are named using *verbs* — e.g. `add()` or `print()`. Verbs describe an action, a state, or an occurrence in the English language. Likewise, this type of nomenclature describes the action performed by a specific function. As we encourage with variable naming: sensible, short and descriptive names are best to consider, when naming a function.

Once you start creating functions for different purposes, you will eventually amass a library of ready-to-use functions which can individually address different needs. This is the primary principle of a popular programming paradigm known as functional programming.

So let us implement the example outline in the diagram:

PYTHON

```python
def add(value_a, value_b):
    """
    Calculates the sum of two numeric values
    given as inputs.

    :param value_a: First value.
    :type value_a: int, float
    :param value_b: Second value.
    :type value_b: int, float
    :return: Sum of the two values.
    :rtype: int, float
    """
    result = value_a + value_b
    return result
```

Once implemented, we can call and use the function we created. We can do so in the same way as we do with the *built-in* functions such as `max()` or `print()`:

PYTHON

```python
res = add(2, 5)

print(res)
```

OUTPUT

```
7
```

REMEMBER

When calling a function, we should always pass our *positional input arguments* in the order they are defined in the function definition: i.e. from left to right.

This is because in the case of *positional arguments*, as the name suggests, the Python interpreter relies on the *position* of each value to identify its *variable name* in the function signature. The function signature for our `add` function is as follows:

```
add(value_a, value_b)
```

So in the above example where we say add(2, 5), the value 2 is identified as the *input argument* for value_a, and not value_b. This happens automatically because in our function call, the value 2 is written in the first position: the position at which value_a is defined in our function declaration (signature).

Alternatively, we can use the name of each *input argument* to pass values onto them in any order. When we use the name of the *input argument* explicitly, we pass the values as *keyword arguments*. This is particularly useful in more complex functions where there are several *input arguments*.

Let us now use *keyword arguments* to pass values to our `add()` function:

PYTHON

```
res = add(value_a=2, value_b=5)

print(res)
```

OUTPUT

```
7
```

Now, even if we change the order of our arguments, the function would still be able to associate the values to the correct keyword argument:

PYTHON

```
res = add(value_b=2, value_a=5)

print(res)
```

OUTPUT

```
7
```

REMEMBER

Choose the order of your *input argument* wisely. This is important when your function can accept multiple *input arguments*.

Suppose we want to define a 'division' function. It makes sense to assume that the first number passed to the function will be divided by the second number:

PYTHON

```
def divide(a, b):
    return a / b
```

It is also much less likely for someone to use *keywords* to pass arguments to this function – that is, to say:

PYTHON

```
result = divide(a=2, b=4)
```

than it is for them to use positional arguments (without any keywords), that is:

PYTHON

```
result = divide(2, 4)
```

But if we use an arbitrary order, then we risk running into problems:

PYTHON

```
def divide_bad(denominator, numerator):
    return numerator / denominator
```

In which case, our function would perform perfectly well if we use *keyword arguments*; however, if we rely on positional arguments and common sense, then the result of the division would be calculated incorrectly.

PYTHON

```
result_a = divide_bad(numerator=2, denominator=4)
result_b = divide_bad(2, 4)

print(result_a == result_b)
```

OUTPUT

```
False
```

PRACTICE EXERCISE 1

Implement a function called find_tata that takes in one `str` argument called seq and looks for the TATA-box motif inside that sequence. Then:

- if found, the function should return the index for the TATA-box as output.
- if not found, the function should *explicitly* return `None`.

Example:

The function should behave, as follows:

```
sequence = 'GCAGTGTATAGTC'

res = find_tata(sequence)
```

Solution

PYTHON

```
def find_tata(seq):
    tata_box = 'TATA'
    result = seq.find(tata_box)

    return result
```

Documentation

It is *essential* to write short, informative documentation for a functions that you are defining. There is no single *correct* way to document a code. However, as a general rule, a sufficiently informative documentation should tell us:

- what a function does;
- the names of the input arguments, and what type each argument should be;
- the output, and its type.

This documentation string is referred to as the *docstring*. It is always written inside triple quotation marks. The *docstring* must be implemented on the *very first line*, immediately following the declaration of the function, in order for it to be recognised as documentation:

PYTHON

```python
def add(value_a, value_b):
    """
    Calculates the sum of two numeric values
    given as inputs.

    :param value_a: First value.
    :type value_a: int, float
    :param value_b: Second value.
    :type value_b: int, float
    :return: Sum of the two values.
    :rtype: int, float
    """
    result = value_a + value_b
    return result
```

REMEMBER

You might feel as though you would remember what your own functions do. Assuming this is often naive, however, as it is easy to forget the specifics of a function that you have written; particularly if it is complex and accepts multiple arguments. Functions that we implement tend to perform specialist, and often complex, interconnected processes. Whilst you might remember what a specific function does for a few days after writing it, you will likely have trouble remembering the details in a matter of months. And that is not even considering details regarding the type of the input argument(s) and those of the output. In addition, programmers often share their work with other fellow programmers; be it within their team or in the wider context of a publication, or even for distribution *via* public repositories, as a community contribution. Whatever the reason, there is one golden rule: a function should not exist unless it is documented.

Writing the *docstring* on the first line is important. Once a function is documented, we can use `help()`, which is a built-in function in Python, to access the documentations as follows:

PYTHON

```
help(add)
```

OUTPUT

```
Help on function add in module __main__:

add(value_a, value_b)
    Calculates the sum of two numeric values
    given as inputs.

    :param value_a: First value.
    :type value_a: int, float
    :param value_b: Second value.
    :type value_b: int, float
    :return: Sum of the two values.
    :rtype: int, float
```

For very simple functions – like the `add()` function that we implemented above, it is sufficient to simplify the docstring into something straightforward, and concise. This is because it is fairly obvious what are the input and output arguments are, and what their respective types are/should be. For example:

PYTHON

```
def add(value_a, value_b):
    """value_a + value_b -> number"""
    result = value_a + value_b
    return result
```

PYTHON

```
help(add)
```

OUTPUT

```
Help on function add in module __main__:

add(value_a, value_b)
    value_a + value_b -> number
```

PRACTICE EXERCISE 2

Re-implement the function you defined in the previous Practice Exercise 1 with appropriate documentation.

Solution

PYTHON

```python
def find_tata(seq):
    """
    Finds the location of the TATA-box,
    if one exists, in a polynucleotide
    sequence.

    :param seq: Polynucleotide sequence.
    :type seq: str
    :return: Start of the TATA-box.
    :rtype: int
    """
    tata_box = 'TATA'
    result = seq.find(tata_box)

    return result
```

Optional arguments

We already know that most functions accept one or more input arguments. Sometimes a function does not need all of the arguments in order to perform a specific task.

Such an example that we have already worked with is `print()`. We already know that this function may be utilised to display text on the screen. However, we also know that if we use the `file` argument, it will behave differently in that it will write the text inside a file instead of displaying it on the screen. Additionally, `print()` has other arguments such as `sep` or `end`, which have specific default values of `' '` (a single space) and `\n` (a linebreak) respectively.

REMEMBER

Input arguments that are necessary to call a specific function are referred to as *non-default arguments*. Those whose definition is not mandatory for a function to be called are known as *default* or *optional arguments*.

Optional arguments may *only* be defined *after* non-default arguments (if any). If this order is not respected, a `SyntaxError` will be raised.

ADVANCED TOPIC

The default value defined for *optional arguments* can theoretically be an instance of any type in Python. However, it is better and safer to only use *immutable* types (as demonstrated in Table) for default values. The rationale behind this principle is beyond the scope of this course, but you can read more about it in the official documentation.

In order to define functions with optional arguments, we need to assign a default value to them. Remember: input arguments are variables with a specific scope. As a result, we can treat our input argument as variables and assign them a value:

PYTHON

```
def prepare_seq(seq, name, upper=False):
    """
    Prepares a sequence to be displayed.

    :param seq: Sequence
    :type seq: str
    :param name: Name of the sequence.
    :type name: str
    :param upper: Convert sequence to uppercase characters (default: False)
    :type upper: bool
    :return: Formatted string containing the sequence.
    :rtype: str
    """
    template = 'The sequence of {} is: {}'

    if not upper:
        response = template.format(name, seq)
    else:
        seq_upper = seq.upper()
        response = template.format(name, seq_upper)

    return response
```

Now if we don't explicitly define `upper` when calling `prepare_seq()`, its value is automatically considered to be `False`:

PYTHON

```
sequence = 'TagCtGC'

prepped = prepare_seq(sequence, 'DNA')

print(prepped)
```

OUTPUT

```
The sequence of DNA is: TagCtGC
```

If we change the default value of `False` for `upper` and set to `True`, our sequence should be converted to upper case characters:

PYTHON

```
prepped = prepare_seq(sequence, 'DNA', upper=True)

print(prepped)
```

OUTPUT

```
The sequence of DNA is: TAGCTGC
```

PRACTICE EXERCISE 3

Modify the function from the previous Practice Exercise 2 to accept an *optional argument* called `upper`, with a default value of `False`. Thereafter:

- if `upper` is `False`, then the function should perform as it already does (similar to the previous Practice Exercise 2);
- if `upper` is `True`, then the function should convert the sequence to contain only uppercase characters, before it looks for the TATA-box.

Do not forget to update the *docstring* of your function.

Solution

PYTHON

```
def find_tata(seq, upper=False):
    """
    Finds the location of the TATA-box,
    if one exists, in a polynucleotide
    sequence.

    :param seq: Polynucleotide sequence.
    :type seq: str
    :param upper: Whether or not to
     homogenise the sequence
     to upper-case characters.
    :type upper: bool
    :return: Start of the TATA-box.
    :rtype: int
    """
    tata_box = 'TATA'

    if not upper:
        result = seq.find(tata_box)
    else:
        seq_prepped = seq.upper()
        result = seq_prepped.find(tata_box)

    return result
```

REMEMBER

It is not necessary to implement your functions in this way. It is, however, a common practice among programmers in any programming language. For this reason, you should be at least be familiar with the technique, as you will likely encounter it at some point.

It is important to note that it is also possible to have more than one `return` in a function. This is useful when we need to account for different outcomes; such as the one we saw in the previous example with `prepare_seq()`.

This means that we can simplify the process as follows:

PYTHON

```
def prepare_seq(seq, name, upper=False):
    """
    Prepares a sequence to be displayed.

    :param seq: Sequence
    :type seq: str
    :param name: Name of the sequence.
    :type name: str
    :param upper: Convert sequence to uppercase characters (default: False)
    :type upper: bool
    :return: Formated string containing the sequence.
    :rtype: str
    """
    template = 'The sequence of {} is: {}'

    if not upper:
        return template.format(name, seq)

    seq_upper = seq.upper()
    return template.format(name, seq_upper)
```

Notice that we got rid of response. Here is a description of what is happening:

- In this context, if the conditional statement holds — i.e. when `upper` is `False`— we enter the `if` block. In this case, we reach the first `return` statement. It is at this point, that function returns the corresponding results, and immediately terminates.
- Conversely, if the conditional statement does not hold — i.e. where `upper` is `True` — we skip the `if` block altogether and proceed. It is only then that we arrive at the second `return` statement where the alternative set of results are prepared.

This does not alter the functionality of the function, in any way. However, in complex functions which can be called repetitively (*e.g.* inside `for` loop), this technique may improve the performance of the function.

Now if we call our function, it will behave in exactly the same way as it did before:

PYTHON

```
sequence = 'TagCtGC'

prepped = prepare_seq(sequence, 'DNA')

print(prepped)
```

OUTPUT

```
The sequence of DNA is: TagCtGC
```

PYTHON

```
prepped = prepare_seq(sequence, 'DNA', upper=True)

print(prepped)
```

OUTPUT

```
The sequence of DNA is: TAGCTGC
```

Interconnectivity of functions

Functions can also call other functions. This is what makes them extremely powerful tools that may be utilised to address an unlimited number of problems.

This allows us to devise a network of functions that can all call each other to perform different tasks at different times. This network of functions can then collectively contribute to the production of a single, final answer.

REMEMBER

Functions should have specialist functionalities.They should ideally be written to perform one task, and one task only.

So in instances where more operations are required, it is advised not to write more code to execute these, into one function. This would defy the ethos of functional programming. Instead, consider writing more functions that contain less code, and perform more specialist functionalities.

EXAMPLE: A MINI TOOLBOX FOR STATISTICS

Suppose we want to calculate the variance of an array of numbers. This is a generalisable task, which means that we could, and should write a function that is able to calculate the variance of *any* array of numbers.

We start off by breaking down the process of calculating the variance, which we know is defined as follows:

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

which breaks down into:

1. calculating the mean or μ,
2. using the mean to calculate the numerator,
3. dividing the numerator by the length of the array.

At least one of these processes is not unique to the calculation of variance; and that is the process by which we work out the *mean*. To that end, we should separate this calculation from our *variance* function, and define it independently:

PYTHON

```
def mean(arr):
    """
    Calculates the mean of an array.

    :param arr: Array of numbers.
    :type arr: list, tuple, set
    :return: Mean of the values in the array.
    :rtype: float
    """
    summation = sum(arr)
    length = len(arr)

    result = summation / length

    return result
```

Now that we have function to calculate the *mean*, we can go ahead and write a function to calculate the variance; which itself relies on *mean*:

PYTHON

```python
def variance(arr):
    """
    Calculates the variance of an array.

    :param arr: Array of numbers.
    :type arr: list, tuple, set
    :return: Variance of the values in the array.
    :rtype: float
    """
    arr_mean = mean(arr)
    denominator = len(arr)

    numerator = 0

    for num in arr:
        numerator += (num - arr_mean) ** 2

    result = numerator / denominator

    return result
```

Now we have two functions, which can be used to calculate the *variance*, or the *mean*, for any array of numbers.

Remember that testing a function is inherent to successful design. So let's test our functions

PYTHON

```python
numbers = [1, 5, 0, 14.2, -23.344, 945.23, 3.5e-2]
```

PYTHON

```python
numbers_mean = mean(numbers)

print(numbers_mean)
```

OUTPUT

```
134.58871428571427
```

PYTHON

```
numbers_variance = variance(numbers)

print(numbers_variance)
```

OUTPUT

```
109633.35462420408
```

Now that we have a function to calculate the *variance*, we can easily proceed to calculate the *standard deviation*, as well.

The standard deviation is calculated from the square root of variance. We can easily implement this in a new function as follows:

PYTHON

```
def stan_dev(arr):
    """
    Calculates the standard deviation of an array.

    :param arr: Array of numbers.
    :type arr: list, tuple, set
    :return: Standard deviation of the values in the array.
    :rtype: float
    """
    from math import sqrt

    var = variance(arr)

    result = sqrt(var)

    return result
```

Now let's see how it works, in practice:

PYTHON

```
numbers_std = stan_dev(numbers)

print(numbers_std)
```

OUTPUT

```
331.1092789762982
```

PRACTICE EXERCISE 4

Write a function that — given an array of any values — produces a dictionary containing the values within the array as *keys*, and the count of those values in the original array (their frequencies), as *values*.

Example:

For the following array:

PYTHON

```
values = [1, 1.3, 1, 1, 5, 5, 1.3, 'text', 'text', 'something']
```

the function should return the above dictionary:

Suggestion: You can add this as a new tool to the statistics mini toolbox.

Solution

PYTHON

```
def count_values(arr):
    """
    Converts an array into a dictionary of
    the unique members (as keys) and their
    counts (as values).

    :param arr: Array containing repeated
                members.
    :type arr: list, tuple
    :return: Dictionary of unique members
                         with counts.
    :rtype: dict
    """
    unique = set(arr)
    arr_list = list(arr)

    result = dict()

    for num in unique:
        result[num] = arr_list.count(num)

    return result
```

Exercises

END OF CHAPTER EXERCISES

Write a function with the following features:

- Call the function `get_basic_stats()` and let it take one input argument which may contain any number of input arrays, *e.g.* a tuple of arrays.
- Using a for loop, for each of the arrays calculate the mean and the variance for each of the arrays using the functions 'mean' and 'variance', given above, *i.e.* call these functions from within the function `get_basic_stats()`.
- Calculate the standard deviation for each array as the square root of the variance. You will have to import the function `sqrt` from module `math`.
- Return a single array containing (in that order) the mean, the variance and the standard deviation for each array.

To test the function, combine three arrays in a tuple as follows:

PYTHON

```
my_arrays = (
    [1, 2, 3, 4, 5],
    [7, 7, 7, 7],
    [1.0, 0.9, 1.2, 1.12, 0.95, 0.76],
)
```

Call the function `get_basic_stats()` with this tuple as an argument, and write the output to a variable. Display the results in the following form:

```
STD of array' index, ':' STD
```

The result for the above arrays should be:

```
STD of array 0 :  1.4142135623730951
STD of array 1 :  0.0
STD of array 2 :  0.14357537702854514
```

Solution

KEY POINTS

- Functions can help to make repetitive tasks efficient, allowing the passing of values into whole blocks of code, with a simple function call.
- Keyword `def` is used to write a function.
- Optional arguments do not require prior definition.
- The potential interconnectivity of functions can make them very powerful.