LEARN PYTHON WITH NO PROGRAMMING EXPERIENCE:

WHY, HOW, AND WHEN TO USE FUNCTIONS

JEAN PAUL KNIGHT

Learn Python With No Programming Experience: Why, How, and When to Use Functions

Jean Paul Knight

This book is for sale at http://leanpub.com/pythonbook

This version was published on 2018-08-02

Contents

Introduction

Welcome and congratulations!

You have made a very important step towards becoming a better Python programmer. In fact, the skills you're about to discover will make you a better programmer, overall! You will be able to pick up any programming language and apply this stuff.

You're also improving your chances of standing out from your competition. As you may have realized, many jobs require basic programming skills. In other jobs, automating your tasks can make your life much easier. Having these skills can give you a feeling of safety and security. Plus, you'll have more confidence in your own abilities.

Here is how it works

You'll first acquire the right mindset: The key to learning Python is having the right mindset. Once you grasp exactly what each concept is about, you'll find it much easier to apply it in real programs. More on that later…

You'll learn the actual syntax Of course, syntax is very important. So, you'll learn that too!

You'll use what you've learned in a practical way You'll get some practice using what you've learned. Some of the examples might seem simple on the surface. But, as you'll see, your biggest challenge will be making things simple. That's one of the most important keys.

The secret to your success…

…lies in taking your time. Here is a weird paradox: the slower you go, the faster you'll move. Becoming good is all about mastering the basics. The deeper you go into the basics, the more you'll be able to do with what you've got.

When you master the basics, a huge chunk of the advanced stuff will come to you naturally. You may actually find yourself using advanced concepts before you learn them from any books or course! It may be hard to believe right now, but once you start experiencing it, you'll never go back.

A quick note about JARGON

Jargon can stop a human in their tracks. I can't tell how many times I've glazed over documentation and then moved on. The same thing happens with some tutorials and even books. Seeing jargon can really put you off programming.

For this reason, I've done my best to explain things in layman's terms. Sadly, the tech world often likes to make things sound hard. I think it's something to do with wanting to appear smart. But, in reality, most jargon could (and should!) be simplified.

Enjoy your journey!

I wish you all the best. Grasping functions can have a big impact on your job and your career. Your life is never going to be the same once you master these skills. Have a good time and…

…let's begin!

Jean Paul PythonLove.com

P.S. Now, as you may know, this guide is a work in progress. I'm constantly working to improve it. Please take a moment right now to share what your biggest challenge is. Do it here: https://bit.ly/python_help[1]

You can be sure that I'll read your response. Then, I'll take what you've said and do my best to improve this guide. That way, you're more likely to get exactly what you need and want.

P.P.S. Again, share your concerns/worries/fears/challenges over here: https://bit.ly/python_help[2]

[1] https://bit.ly/python_help
[2] https://bit.ly/python_help

The Secret Mindset of a Programmer - How to Think And Other Mysteries

Learning Python is like learning magic. It's as if you're gathering powers that make the computer "do stuff". You're acquiring a skill possessed by only a small percentage of the population. However, the path to becoming really good with Python, is full of deadly traps. These traps can cause you to get stuck or to give up altogether. Yet, there is one thing that can speed up your journey.

There is a way that programmers think that is different from the rest of the world. If you want to understand functions and other programming concepts, you must have the right mindset.

Why is it important to think like a programmer?

Before we go deeper into functions and other Python concepts, we must look at a crucial mindset. The mindset of a programmer. When you have this mindset, you will find it much easier to bridge the gap between beginner and intermediate. You will also be able to quickly remove any mental block that cause you to struggle.

On the other hand, without the right mindset you may find yourself stuck way too often. You may find it hard to progress beyond tutorials. In particular, it may be hard to start making your own programs. Worse still, documentation (often badly written) will be enough to stop you in your tracks.

There is another reason why having the right mindset is important. If you are using Python to automate stuff, you'll be more likely to find quick solutions. If Python is required for you job, you will write code that impresses your peers, and more importantly your boss. Even if you code for yourself, the right kind of thinking will make your work easier, faster, and more efficient.

Don't skip the mindset bit! I believe the biggest reason why people get stuck in "newbie" mode is not understanding how programming works. Let's take a deeper look.

What does thinking like a programmer mean?

For the purpose of this guide, we will use a very simple defition.

> A programmers mindset is a way of thinking that focuses on creating useful and practical solutions to problems.

It means you grasp the basic fundamentals that make you a good programmer. The challenge is, that the programmers mindset doesn't have much to do with coding. That's what trips up many beginners. For instance, many people spend a lot of time on syntax.

It's kind of like putting on a stage show. The hard work happens during the reahersals, behind the scenes. Then, When it comes to programming, most of your work happens before you start up your favorite code editor.

The biggest mistake beginners make is...

...focusing too much on the code. Yet, there are two skills that will make you a master - neither of them involve writing code. They are:

1. Knowing how to learn
2. Knowing how to solve problems.

Great programmers ***always* keep learning**. There is no programmer who knows it all. So, as you progress you will always find yourself learning new libraries. You will also discover new concepts along the way. The key to becoming better is accepting that you'll always be a learner. For this reason, you need to have skills that allow you to learn quickly.

The other keys is knowing **how to solve problems**. When it comes to problem solving, the computer is not your best friend. In fact, the computer is your *servant* but we'll come back to that later. Your true best friend is a piece of paper (and a pen!)

> As a good programmer, you'll need to be able to solve problems using pen and paper. Then, once you've solved the problem, you use the solution as your guide when writing the code.

"OK, so what happens next?"

Next we're going to mix up learning Python with learning the right mindset. That way, you'll naturally absorb the concepts you need to grasp. As a result, you'll become better at using functions. At the same time, your overall programming skills will go to the next level.

Let's go! It's time to discover why you need functions in the first place...

Why You Need Functions

If you think you don't need functions, you're kind of right. In theory, you can do almost anything without using functions. The only trouble is, it will be a long, painful struggle. In practice, functions serve a very important role.

> The truth is, for small scripts, functions may not make a big difference. However, if you want to write full-fledged programs, functions play a very important role. In fact, the more you understand functions, the greater your ability to write complex programs will be.

Functions make it easier to manage complex programs and applications.

Let's face it: looking at a mass of code can be frustrating and intimidating. Badly organized code put a strain on everybody. It is hard to read and difficult to fix.

However, functions allow you to organize your code into neat "chunks". You can then rearrange them in any way you see fit.

Functions allow you to simplify your code

> Your brain finds it hard to grasp several complex concepts all at once. That's why you need to turn each complex part into something simple. For example, you may have a task that takes 20 lines of code. You can put all that code into one function. Then, the next time you need to do that task, you'll only need *one line* of code to make it work.

Thanks to functions, it's much easier to find and fix bugs - especially in large programs.

Programmers spend a lot of time fixing errors. The more lines of code you write, the more likely you are to encounter an error. The good news is, functions reduce the number of errors you have to deal with.

With functions, you only need to fix the code in one place. If the function's task is causing issues, you simply make a change in that function. This speeds up your work. You'll also have less frustration having to search for the same error, over and over again.

When you use functions, you don't have to write the same code over and over again.

A function acts as a self-contained unit for a specific task. Once you've figured out how to make the task work, simply use a function for it. You don't need to rewrite the code.

Functions make it easier to understand what your code does.

As a programmer, you'll often find yourself going back to your code. For example, you might want to add extra features to a script you've been using. It can be very confusing trying to understand what each part of your code does. That's where you'll find functions to be very handy.

Well named functions make it clear what they do. When you do write functions the right way, it becomes easier to read your code. So, you'll often be able to grasp what the code is about just by reading the function's name. You'll also find it much easier to fix any bugs, errors, or mistakes. That will save you the time you'd have to use to comb over the code. Thank you functions!

When solving a complex problem, functions make it easier.

When writing Python code, you'll constantly find yourself breaking things down. Smaller chunks make each part of the problem small enough to be solved. Functions can store each chunk so you don't have to think about it again.

With functions, you can separate details from the big picture.

In the simplest scenario, you write your functions and keep them in one file. Then you use them in another file, focusing on the main logic of your program. This will make your life much, much easier!

On the other hand...

###without functions... ...life is pure misery. Here are some reasons why:

Without functions, it's almost impossible to write full-fledged Python programs.

A big, complex program is simply a collection of small solutions to various problems. Each solution is connected in a way that makes it useful to you and other people. Without functions, all you have is a single huge mass of code - rather than manageable chunks.

If you don't use functions, your skills will likely remain at the scripting level (as opposed to actual programming).

Advanced programming is all about solving problems and managing complexity. Without a good grasp of functions, you will have a limited ability to create useful solutions. This will affect the level of competency at which you can use your Python skills.

When you don't grasp functions and how they work, it's much harder to advance as a Python programmer.

Since functions are a basic part of Python, you'll be expected to understand them during job interviews. There's no way around it. Remember, a pro is someone who can use basic concepts without having to think about them.

Lack of deep understanding of functions make it harder to learn other concepts such as OOP (Object Oriented Programming).

As you keep using Python, you'll keep coming across invisible walls. The way to cross these walls is by developing your skills. For instance, without OOP it's difficult to handle many problems. But, if you can't use functions you won't be able to learn OOP.

Without functions, your code is boring.

Yep. I said it. Without functions, your code becomes a boring blob of code. It's very hard to understand what your code does. However, a well-named function is like a picture; it's worth a thousand words!

Without functions you have to work way too hard.

When you make a mistake, you'll have to correct in many places. This could lead to even more mistakes that are harder to find. Yet, with functions, you can fix it in one place and you're good to go!

Writing code without functions wastes time.

Without functions, you have to copy and paste the code to reuse it. But, once you have a function, you can keep using it as many times as you want. You can share that code with your friends and let them use it in their programs!

It looks like we're ready for the next step. It's time to answer the questions of the day: "**What in the world is a function?**" (You may need this answer in case someone asks you!)

So... What is a Function Exactly?

You may be surprised that you already use functions in your life. Have you ever followed a recipe or any other set of instructions? That's basically how functions work. Let's look at an example.

Let's start with an imaginary friend named Tom. You want to teach Tom to make a cup of tea. For the sake of this example, let's assume that your friend Tom is really bad at remembering how to do stuff.

After a hard day's work, you decide to have a nap. Out of the blue, your phone rings. It's Tom! He wants you to tell him how to make a cup of tea. As annoyed as you feel, you still want to be a good friend. You decide to help him out.

You may tell him something like this:

1. *Boil the water in the kettle.*
2. *Pour the water into a cup.*
3. *Insert a teabag.*
4. *Put in some sugar.*
5. *Mix it all up!*

So, your friend goes off to make the cup of tea. He follows the instructions. But...

... the next time he wants to make a cup of tea, he calls you again. It gets worse.

Tom has four visitors. He keeps calling you any time a visitor wants some tea. This is annoying because you have to repeat yourself over and over again. And, you don't get to have your nap. The good news is, you're smart. You're very smart.

You have a good solution to this problem. You write down what to do on a piece of paper. You name that piece of paper "make tea". You put "make tea" on top of the instructions. Then, you get Tom to stick it on his fridge.

So, anytime Tom wants to "make tea" he picks up that piece of paper. Then, he follows the instructions. You don't have to repeat the steps every time - Tom can refer to the piece of paper called "make tea".

In a similar way, when you write a program, you often have to do certain tasks over and over again. The more complex the program is, the more this happens. Now, instead of copying and pasting the code, you can simply put that code inside of a function. Then, you can get that code to work in any part of your program by "calling" the function.

Jargon alert!

"**Calling**" a function simply means getting it to do its job. In Python, you simply write the name of the function followed by parenthesis (brackets). Another name for doing this is to "**invoke**" the function.

Example:

```
make_tea() #This will call/invoke the function

make_tea #Without the brackets/parenthesis at the end, the code will do nothing.
```

Based on the above, we can arrive at a simple description of what a function is.

What is a function?

In most simple terms, a function is like a recipe that your program can use to perform a specific task. It is a self-contained set of instructions used to perform the task. Any time you need that task in your program, you simply invoke or call the function. As mentioned above, without a function, you'd have to write the same code over and over again.

How to Create a Function In Python (Part I)

> **Before we get to the syntax**...
>
> ...there is one thing you should do before you create a function. Write down what you want your function to do. At the very least, have a rough idea. This will help you along the way. As practice, write down: "The function *make_tea* will let me know when the tea is ready."

Creating functions in Python is relatively simple.

1. First, type `def`
2. Then, type in the name of your function. In this case, we will call it `make_tea`
3. Then, type in parentheses `()`. The parentheses play an important role - we'll talk about that in just a bit.
4. As your next step, type in a colon `:`. This lets Python know that you're about to start the body of the function.

> As a convention, Python function names are written in lower case. The name should describe what the function does or *returns* (we will talk about the `return` statement in a bit.)If there is more than one word, you can space the words using the underscore. Here are a few examples:
>
> total_purchase store_info show_me_the_secret_to_makeing_million_dollars_pretty_please
>
> **Function body** refers to the actual code contained by your function.

At this point, your code should look something like this:

```
1  def make_tea():
```

> **IMPORTANT!** Your function will most likely through an error if it contains nothing. Don't worry, the `pass` statement comes to the rescue. If you're not yet sure what you'll put in the function, just type `pass` as your function body. The `pass` statement does nothing but at least your function is not empty!

We will use the `pass` statement as a placeholder. So your code will look something like this:

```
def make_tea():
    pass
```

Now, notice that we have there are four spaces before the `pass` statement. This is known as *indenting* your code. All the code in your function has to be indented by four spaces. This tells Python that the code belongs to the function. If you don't ident the function body, you will get errors. Plus, Python will not know which parts of your code belong to the function. (You may notice that your code will align with the function name.

Alright, so we now have the basic syntax for functions. Let's put in some actual code. For now, we will simply get our function to say "Tea is ready!". To accomplish that, we will use the `print` function.

> The `print` function comes with Python. You use it to display text.

Type in the following:

```
def make_tea():
    print("Tea is ready!")
```

If you run this code, you will notice that it does... absolutely nothing! That's because functions only work when you tell them to. This is known as *calling* or *invoking* a function.

> **calling** or **invoking** the function lets Python know you want the function to do its job. It's kind of like ringing up your friend Tom and saying "Make tea!"

The way you call a function in Python is simply by using the function name followed by the parentheses. In our case, you'd want to do this:

```
make_tea()
```

Keep in mind that the function name on its own does not work. You MUST add the parentheses. So, for instance, doing this

```
make_tea
```

will not cause the function to run. This is important, you'll be surprised how often people get stuck just because they forgot to add the parentheses.

OK, so now it's time to make our functions a little bit more interesting. We will look at parameters and their cousins; arguments.

How to Create a Function In Python (Part II)

###The Truth Revealed: Parameters vs Arguments

In the world of programming, there is a question that keeps coming up. That question is: “What’s the difference between parameters and arguments”. We’re about to settle the… erm… argument.

If you don’t have any idea what any of the above means, that’s OK. You will, in just a little while.

So, where were we?

Oh, yes. When you create a function, sometimes you will want to use some data with it. For instance, you might want to tell their tea is ready. How do you let the function know whose tea will be served next?

First, the function has to be set up in a way that lets it accept data. This will be done by creating placeholders in the parentheses. This is known as adding parameters to a function.

> A *parameter* is simply a placeholder for data that will be used in a function. You place the parameter inside the parenthesis.

Let’s take a look at how this will work with our `make_tea` function. Type in the following code:

```
def make_tea(name):
    print("Your tea is ready, " + name)
```

Now, we can call our function with data. The actual data we pass into a function is known as *arguments*.

> An *argument* is a piece of data passed into a function when you call that function

Let’s see how that might work in our example.

```
name = "Jason X"
# we will now call our function
make_tea(name)
```

This will print out `Your tea is ready, Jason X`

Try changing `name` to something else, perhaps your name. (I'm assuming your name is not Jason X, in which case you can just go with the example.)

So…

The difference between arguments and parameters is as follows.

> **Parameters** represent the data your function is expecting. You use *parameters* when you are creating your function.
>
> On the other hand… **Arguments** are the actual data you pass to the function. You use *arguments* when you're calling a function.

So, the next time you come across this question, you know how to answer it. (Don't forget to share where you got it from!) Whew! Now that you know the difference between parameters and arguments, it's time to step up the game. We're moving up to…

The *return* of the... oh, wait...

It's time for the return. I mean, it's time for the `return` statement. It has a very simple yet important role.

> The `return` statement provides the result of a function's process to the rest of the program. You can then assign that result to a variable.

As you learned earlier, a function is meant to perform a specific task. In most cases, your function will work on some data. Once it's finished with the data, you can use the `return` statement to send that data to a variable. Here is a code example:

```
1  def add_numbers(a , b):
2      c = a + b
3      return c
4
5  total = add_numbers(5, 6) #This assigns the result to the `total` variable
6
7  print(total)
```

This will print out `11`. Here is what happened.

We created our function `add_numbers`. This function accepts two parameters `a` and `b`. Inside the function, `a` and `b` are added to each other. The resulting number is assigned to `c`. Finally, the function returns `c`.

To get the result of the function, we created a variable called `total`. We then assigned the value of `add_numbers` to the variable `total`.

That's it! So anytime you want to use the result of a function, you use `return`. Here are a few tips.

- In most cases, you will only need to return one value.
- If you need to return more than one value, turn that into a list or a dictionary.
- Not every function needs to have a `return` statement. It all depends on what you're doing at the time.

*args - Now, what's that?

You may have come across something called `*args`. Even if you haven't, you will so pay close attention. You're about to discover one of the most important concepts about creating and using functions.

You see, in certain situations, you may not be sure how many arguments your function will need. Yet, you may need your program to be ready to handle such a scenario. In this case, all you need to do is to place an asterisk `*` before the parameter. The most common way to do it is to use `args` as the parameter name. However, it doesn't matter what the name is. As long as you place an asterisk, Python will know what to do.

Let's use our `make_tea` example. We will use the function to use any number of guests' names.

```
def make_tea(*names):
        # So our function will inform each guest that their tea is ready
        for name in names:
                print("Your tea is ready, " + name)
```

So now we can put in as many guest names as we want! Let's try!

```
make_tea ("Jason X", "Peter Pan", "Han Solo", "Heisenberg")
```

This will print out

```
Your tea is ready, Jason X
Your tea is ready, Peter Pan
Your tea is ready, Han Solo
Your tea is ready, Heisenberg
```

How does that happen? Python will take each of the arguments, in this case guest names, and put it into a list. The list, in this case, will be called `names` - because we put an asterisk before the `names` parameter. If we used `*coders` the list would have been called coders.

Then, our function goes through each item in the list. It prints out our message for each guest. That's it!

To recap: * You can use an asterisk `*` before a parameter - this turns that parameter into a list. * When you call the function, the arguments you use in place of that parameter will be stored in a list * By convention we use `*args` as a placeholder for an unknown number of arguments. However, as long as you place the asterisk before the parameter, you're fine. E.g. `*names`, `*space_ships`, `*magic_wands`

On this note, it's time to meet the cousin of our dear `*args`...

**kwargs - OK, My Jargon Radar is Beeping!

> Remember, **jargon** likes to intimidate. Whenever you come across a confusing word, pause for a moment. It's probably not as hard as it looks. At the same time, don't rush it. For the most part, try to avoid learning too many new terms at once. In the beginning, focus on learning 1 - 2 new jargon terms in a day.

"Hehe, actually `**kwargs` are kind of cool!" said no one ever! (until they finished reading this chapter, that is.)

Well, well, well. It's time to deal with the famous `**kwargs`. And deal with it we will.

So... the term `**kwargs` is just a fancy way to say `keyword arguments`. What's `keyword arguments`? We'll start by looking at an example.

```
make_tea(sweetener="honey", flavoring="lemon")
```

As you may have guessed, keyword arguments are simply pieces of data you pass to your functions - each piece having its own name. In the above example, we have sweetener passed as honey. The flavor is "lemon".

Using `**kwargs` allows you to pass as many keyword arguments as you wish. The arguments you pass are then stored as a dictionary.

In case you're not familiar with dictionaries, here is a quick run through:

> A **dictionary** is a way of grouping data using keys and values. The `key` acts as a name for a piece of data. A `value` is the actual value for that piece of data. Here is an example of a dictionary:

```
phone_dictionary = {
    'name' : 'iPhone',
    'color' : 'black',
    'year' : '2016'
}
```

> Notice the format of a dictionary. Curly brackets `{}` mark the beginning and the end of a dictionary. Then, each key and value are connected using a colon `:` like this `'key' : 'value'`. Finally, we separate each key-value pair with a comma.

Let's say you want to access a piece of data from a dictionary. To do that, you will use the name of that dictionary followed by the key. The key will be enclosed in square brackets like this `name_of_dictionary['key']`.

Example:

```
print(phone_dictionary['name'])
```

This will print out `iPhone`. Likewise:

```
print(phone_dictionary['color'])
```

will print out `black`. Go ahead and try it!

Ok, so now that we know what dictionaries are all about, we're ready to use `**kwargs`. Let's get back to our tea example.

Everybody likes their tea differently. Thanks to `**kwargs`, each guest can now have their tea just the way they like it. Here is our new `make_tea` function.

```
def make_tea(**kwargs):
    for key, value in kwargs.items():
        print("This tea uses " + value + " as " + key)
```

Now, we will pass the arguments to the function as keywords. These arguments will then be turned into a dictionary. Here is how:

```
make_tea(sweetener='sugar', leafs="Green tea", flavoring="nothing")
```

This code will then print out:

```
This tea uses sugar as sweetener
This tea uses Green tea as leafs
This tea uses nothing as flavoring
```

Here are some tips for you to keep in mind.

- As long as you put a double asterisk `**` before a parameter, the function will expect keyword arguments in its place. You don't have to call it `kwargs`
- The dictionary created will be named after the parameter. So if you use `**swords`, inside the function, the dictionary will be called `swords`
- Don't worry too much about `*args` and `**kwargs`. As you keep learning Python, you will know when the time comes for these concepts. In many cases, you won't have any need for them.

APPENDIX I - Python Functions Cheat Sheet

Why we need functions

- Functions make it easier to manage complex programs and applications.
- Thanks to functions, it's much easier to find and fix bugs - especially in large programs.
- When you use functions, you don't have to write the same code over and over again.
- Functions make it easier to understand what you code does.
- When solving a complex problem, functions make it easier to break down that problem into smaller chunks.

On the other hand...

- Without functions, it's almost impossible to write full-fledged Python programs.
- If you don't use functions, your skills will likely remain at the scripting level (as opposed to actual programming).
- When you don't grasp functions and how they work, it's much harder to advance as a Python programmer.
- Lack of deep understanding of functions make it harder to learn other concepts such as OOP (Object Oriented Programming).

What is a function?

In most simple terms, a function is like a recipe that your program can use to perform a specific task. Any time you need that task in your program, you simply invoke the function. As mentioned above, without a function, you'd have to write the same code over and over again.

Anatomy of a function

```
#`def` tells Python you're about to define a function

def register_user(name, email): #Here, `register_user` is the name of our function.
#The colon `:` at the end of the line, tells Python the function body is following n\
ext.

        #The lines below form the function body
    print("Registering user...")
    print("The user's name is: " + name)
    print("The user's email is: " + email)

    user_id = 10

    return user_id #In this example, `user_id` is the return value
```

Let's bust the jargon...

parameters - Parameters are placeholders for the data your function needs to do its job. (**MAJOR KEY ALERT**: Not all functions have parameters). You decide on what your parameters are when creating the function.

`return` - what `return` does is to make data available to the rest of your program. In most cases, `return` provides the result of what the function was designed to do. (**MAJOR KEY ALERT**: Not all functions return a value)

arguments - This refers to the actual data you pass to the function when the function is being gcggused.

Using the earlier example:

```
name = "John Doe" #This will be passed as an argument when you use the function
email = 'johndoe@example.com' #This will be passed as an argument when you use the f\
unction
registration_number = register_user(name, email) # This will assign 10 to registrati\
on_number
```

function body - the code that your function contains

pass an argument - provide actual data for the function to use. You pass the data as the parenthesis, inside the brackets that follow the function name.

> **Questions?** Get in touch: https://leanpub.com/pythonbook/email_author/new[3]

[3] https://leanpub.com/pythonbook/email_author/new