

## ✓ Writing Reusable Code Using Functions in Python

### Part 4 of "A Gentle Introduction to Programming with Python"

This tutorial is the fourth in a series on introduction to programming using the Python language. These tutorials take a practical coding-based approach, and the best way to learn the material is to execute the code and experiment with the examples. Check out the full series here:

1. [First Steps with Python and Jupyter](#)
2. [A Quick Tour of Variables and Data Types](#)
3. [Branching using Conditional Statements and Loops](#)
4. [Writing Reusable Code Using Functions](#)
5. [Reading from and Writing to Files](#)
6. [Object Oriented Programming with Classes](#)

### How to run the code

This tutorial hosted on [Jovian.ml](#), a platform for sharing data science projects online. You can "run" this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your own computer*.

This tutorial is a [Jupyter notebook](#) - a document made of "cells", which can contain explanations in text or code written in Python. Code cells can be executed and their outputs e.g. numbers, messages, graphs, tables, files etc. can be viewed within the notebook, which makes it a really powerful platform for experimentation and analysis. Don't afraid to experiment with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top of the notebook.

#### Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing this notebook is to click the "Run" button at the top of this page, and select "Run on Binder". This will run the notebook on [mybinder.org](#), a free online service for running Jupyter notebooks. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

#### Option 2: Running on your computer locally

You'll need to install Python and download this notebook on your computer to run in locally. We recommend using the [Conda](#) distribution of Python. Here's what you need to do to get started:

1. Install Conda by [following these instructions](#). Make sure to add Conda binaries to your system PATH to be able to run the `conda` command line tool from your Mac/Linux terminal or Windows command prompt.
2. Create and activate a [Conda virtual environment](#) called `intro-to-python` which you can use for this tutorial series:

```
conda create -n intro-to-python -y python=3.8
conda activate intro-to-python
```

You'll need to create the environment only once, but you'll have to activate it every time want to run the notebook. When the environment is activated, you should be able to see a prefix (`intro-to-python`) within your terminal or command prompt.

3. Install the required Python libraries within the environmebt by the running the following command on your terminal or command prompt:

```
pip install jovian jupyter numpy pandas matplotlib seaborn --upgrade
```

4. Download the notebook for this tutorial using the `jovian clone` command:

```
jovian clone aakashns/python-functions-and-scope
```

The notebook is downloaded to the directory `python-functions-and-scope`.

5. Enter the project directory and start the Jupyter notebook:

```
cd python-functions-and-scope
jupyter notebook
```

6. You can now access Jupyter's web interface by clicking the link that shows up on the terminal or by visiting <http://localhost:8888> on your browser. Click on the notebook `python-functions-and-scope.ipynb` to open it and run the code. If you want to type out the code yourself, you can also create a new notebook using the "New" button.

## ✓ Creating and using functions

A function is a reusable set of instructions. A function takes one or more inputs, performs certain operations, and often returns an output. Python provides many in-built functions like `print`, and also allows you to define your own functions.

You can define a new function using the `def` keyword.

```
def say_hello():  
    print('Hello there!')  
    print('How are you?')
```

Note the round brackets or parantheses `()` and colon `:` after the function's name. Both are essential parts of the syntax for defining a function. The *body* of the function can contain one or more statements which are to be executed when the function is called. Similar to conditional statements and loops, the statements must be indented by 4 spaces.

The statements inside a function's body are not executed when a function is defined. To execute the statements, we need to *call* or *invoke* the function.

```
say_hello()
```


```
⇒ Hello there!  
   How are you?
```

## ✓ Function arguments

Functions can also accept one or more values as *inputs* (also known as *arguments* or *parameters*). Arguments help us write flexible functions which can perform the same operation on different values. Further, functions can also return a value as a result using the `return` keyword, which can be stored in a variable or used in other expressions.

Here's a function that filters out the even numbers from a list.

```
def filter_even(number_list):  
    result_list = []  
    for number in number_list:  
        if number % 2 == 0:  
            result_list.append(number)  
    return result_list  
  
even_list = filter_even([1, 2, 3, 4, 5, 6, 7])  
  
even_list
```

 [2, 4, 6]

## ✓ Writing great functions in Python

As a programmer, you will spend most of your time writing and using functions, and Python offers many features to make your functions powerful and flexible. Let's explore some of these by solving a problem:

Radha is planning to buy a house that costs \$1,260,000. She is considering two options to finance her purchase:

- Option 1: Make an immediate down payment of \$300,000, and take loan 8-year loan with an interest rate of 10% (compounded monthly) for the remaining amount.
- Option 2: Take a 10-year loan with an interest rate of 8% (compounded monthly) for the entire amount.


Both these loans have to be paid back in equal monthly installments (EMIs). Which loan has a lower EMI among the two?

Since we need to compare the EMIs for two loan options, it might be helpful to define a function to calculate the EMI for a loan, given inputs like the cost of the house, the down payment, duration of the loan, rate of interest etc. We'll build this function step by step.

To begin, let's write a simple function that calculates the EMI on the entire cost of the house, assuming that the loan has to be paid back in one year, and there is no interest or down payment.

```
def loan_emi(amount):  
    emi = amount / 12  
    print('The EMI is {}'.format(emi))
```

```
loan_emi(1260000)
```

 The EMI is \$105000.0

## ✓ Local variables and scope

Let's add a second argument to account for the duration of the loan, in months.

```
def loan_emi(amount, duration):  
    emi = amount / duration  
    print('The EMI is {}'.format(emi))
```

Note that the variable `emi` defined inside the function is not accessible outside the function. The same is true for the parameters `amount` and `duration`. These are all *local variables* that lie within the *scope* of the function.

**Scope:** Scope refers to the region within the code where a certain variable is visible. Every function (or class definition) defines a scope within Python. Variables defined in this scope are called *local variables*. Variables that are available everywhere are called *global variables*. Scope rules allow you to use the same variable names in different functions without sharing values from one to the other.

`emi`



`amount`



`duration`



We can now compare a 6-year loan vs. a 10-year loan (assuming no down payment or interest).

```
loan_emi(1260000, 8*12)
```



The EMI is \$13125.0

```
loan_emi(1260000, 10*12)
```



The EMI is \$10500.0

## ✓ Return values

As you might expect, the EMI for the 6-year loan is higher compared to the 10-year loan. Right now we're printing out the result, but it would be better to return it and store the results in variables for easier comparison. We can do this using the `return` statement

```
def loan_emi(amount, duration):  
    emi = amount / duration  
    return emi
```

```
emi1 = loan_emi(1260000, 8*12)
```

```
emi2 = loan_emi(1260000, 10*12)
```

```
emi1
```

```
⇒ 13125.0
```

```
emi2
```

```
⇒ 10500.0
```

## ✓ Optional arguments

Let's now add another argument to account for the immediate down payment. We'll make this an *optional argument*, with a default value of 0.

```
def loan_emi(amount, duration, down_payment=0):  
    loan_amount = amount - down_payment  
    emi = loan_amount / duration  
    return emi
```

```
emi1 = loan_emi(1260000, 8*12, 3e5)
```

```
emi1
```

```
⇒ 10000.0
```

```
emi2 = loan_emi(1260000, 10*12)
```

```
emi2
```

```
⇒ 10500.0
```

Next, let's add the interest calculation into the function. Here's the formula used to calculate the EMI for a loan:

$$EMI = \frac{P \times r \times (1 + r)^n}{(1 + r)^n - 1}$$

where:

- P is the loan amount (principal)

```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    return emi
```

Note that while defining the function, required arguments like `cost`, `duration` and `rate` must appear before optional arguments like `down_payment`.

Let's calculate the EMI for Option 1

```
loan_emi(1260000, 8*12, 0.1/12, 3e5)
```

⇒ 14567.19753389219

While calculating the EMI for Option 2, we need not include the `down_payment` argument.

```
loan_emi(1260000, 10*12, 0.08/12)
```

⇒ 15287.276888775077

## ✓ Named arguments

Invoking a function with many arguments can often get confusing, and is prone to human errors. Python provides the option of invoking functions with *named* arguments, for better clarity. Function invocation can also be split into multiple lines.

```
emi1 = loan_emi(
    amount=1260000,
    duration=8*12,
```

```

    rate=0.1/12,
    down_payment=3e5
)

```

```
emi1
```

```
⇒ 14567.19753389219
```

```
emi2 = loan_emi(amount=1260000, duration=10*12, rate=0.08/12)
```

```
emi2
```

```
⇒ 15287.276888775077
```

## ✓ Modules and library functions

We can already see that the EMI for Option 1 seems to be lower than the EMI for Option 2. However, it would be nice to round up the amount to full dollars, rather than including digits after the decimal. To achieve this, we might want to write a function which can take a number and round it up to the next integer (e.g. 1.2 is rounded up to 2). That would be a good exercise to try out!

However, since rounding numbers is a fairly common operation, Python provides a function for it (along with thousands of other functions) as part of the [Python Standard Library](#). Functions are organized into *modules*, which need to be imported in order to use the functions they contain.

**Modules:** Modules are files containing Python code (variables, functions, classes etc.). They provide a way of organizing the code for large Python projects into files and folders. The key benefit offered by modules is *namespaces* - a module or a specific function/class/variable from a module has to be imported before it can be used within a Python script or notebook. This provides *encapsulation* and avoids naming conflicts between your code vs. a module, or across modules.

For rounding up our EMI amounts, we can use the `ceil` function (short for *ceiling*) from the `math` module. Let's import the module and use it to round up the number 1.2 .

```
import math
```

```
help(math.ceil)
```

```
⇒ Help on built-in function ceil in module math:
```

```

ceil(x, /)
    Return the ceiling of x as an Integral.

```



This is the smallest integer  $\geq x$ .

```
math.ceil(1.2)
```

↩ 2

Let's now use the `math.ceil` function within the `home_loan_emi` function to round up the EMI amount.

Using function to build other functions is a great way to reuse code and implement complex business logic while still keeping the code small, understandable and manageable. Ideally, one a fuction should do one thing, and one thing only. If you find yourself doing too many things within a single function, you should consider splitting it into 2 or more smaller, independent functions. As a rule of thumb, try to limit your functions to 10 lines of code or less. Good programmers always write small, simple and readable functions.

```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    emi = math.ceil(emi)
    return emi
```

```
emi1 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12,
    down_payment=3e5
)
```

```
emi1
```

↩ 14568

```
emi2 = loan_emi(amount=1260000, duration=10*12, rate=0.08/12)
```

```
emi2
```

↩ 15288

Let's compare the EMIs and display a message for the option with the lower EMI.

```

if emi1 < emi2:
    print("Option 1 has the lower EMI: {}".format(emi1))
else:
    print("Option 2 has the lower EMI: {}".format(emi2))

```

⇒ Option 1 has the lower EMI: \$14568

## ✓ Reusing and improving functions

Now we know for certain that "Option 1" has the lower EMI among the two options. But what's even better is that we now have a handy function `loan_emi` that can be used to solve many other similar problems with just a few lines of code. Let's try it with a couple more problems.

Q: Shaun is currently paying back a home loan for a house a few years go. The cost of the house was \$800,000. Shaun made a down payment of 25% of the cost, and financed the remaining amount using a 6-year loan with an interest rate of 7% per annum (compounded monthly). Shaun is now buying a car worth \$60,000, which he is planning to finance using a 1-year loan with an interest rate of 12% per annum. Both loans are paid back in EMIs. What is the total monthly payment Shaun makes towards loan repayment?

This question is now straightforward to solve, using the `loan_emi` function we've already defined.

```

cost_of_house = 800000
home_loan_duration = 6*12 # months
home_loan_rate = 0.07/12 # monthly
home_down_payment = .25 * 800000

emi_house = loan_emi(amount=cost_of_house,
                      duration=home_loan_duration,
                      rate=home_loan_rate,
                      down_payment=home_down_payment)

```

emi\_house

⇒ 10230

```

cost_of_car = 60000
car_loan_duration = 1*12 # months
car_loan_rate = .12/12 # monthly

emi_car = loan_emi(amount=cost_of_car,
                   duration=car_loan_duration,
                   rate=car_loan_rate)

```

```
emi_car
```

```
↳ 5331
```

```
print("Shaun makes a total monthly payment of ${} towards loan repayments.".format(e
```

```
↳ Shaun makes a total monthly payment of $15561 towards loan repayments.
```

## ✓ Exceptions and try-except

Q: If you borrow \$100,000 using a 10-year loan with an interest rate of 9% per annum, what is the total amount you end up paying as interest?

One way to solve this problem is to compare the EMIs for two loans: one with the given rate of interest, and another with a 0% rate of interest. The total interest paid is then simply the sum of monthly differences over the duration of the loan.

```
emi_with_interest = loan_emi(amount=100000, duration=10*12, rate=0.09/12)
emi_with_interest
```

```
↳ 1267
```

```
emi_without_interest = loan_emi(amount=100000, duration=10*12, rate=0./12)
emi_without_interest
```

```
↳
```

Something seems to have gone wrong! If you look at the error message above carefully, Python tells us exactly what is gone wrong. Python *throws* a `ZeroDivisionError` with a message indicating that we're trying to divide a number by zero. This is an *exception* that stops further execution of the program.

**Exception:** Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions. We refer to exceptions as being typically stop further execution of the program, unless they are handled within the program using `try - except` statements.

Python provides many built-in exceptions that are *thrown* when built-in operators, functions or methods are used in an incorrect manner: [https://docs.python.org/3/library/exceptions.html#built-](https://docs.python.org/3/library/exceptions.html#built-in-exceptions)

[in-exceptions](#) . You can also define your own custom exception by extending `Exception` class (more on that later).

You can use the `try` and `except` statements to *handle* an exception. Here's an example:

```
try:
    print("Now computing the result..")
    result = 5 / 0
    print("Computation was completed successfully")
except ZeroDivisionError:
    print("Failed to compute result because you were trying to divide by zero")
    result = None

print(result)
```

```
➞ Now computing the result..
Failed to compute result because you were trying to divide by zero
None
```

When an exception occurs in the code inside a `try` block, the rest of the statements in the block are skipped, and `except` statement is executed. If the type of exception thrown matches the type of exception being handled by the `except` statement, then the code inside the `except` block is executed and the program execution then returns to the normal flow.

You can also handle more than one type of exception using multiple `except` statements. Learn more about exceptions here: [https://www.w3schools.com/python/python\\_try\\_except.asp](https://www.w3schools.com/python/python_try_except.asp) .

Let's enhance the `loan_emi` function to use `try - except` to handle the scenario where the rate of interest is 0%. It's common practice to make changes/enhancements to functions over time, as new scenarios and use cases come up. It makes functions more flexible & powerful.

```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    try:
        emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    except ZeroDivisionError:
        emi = loan_amount / duration
    emi = math.ceil(emi)
    return emi
```

We can use the updated `loan_emi` function to solve our problem.

Q: If you borrow \$100,000 using a 10-year loan with an interest rate of 9% per annum, what is the total amount you end up paying as interest?

```
emi_with_interest = loan_emi(amount=100000, duration=10*12, rate=0.09/12)
emi_with_interest
```

1267

```
emi_without_interest = loan_emi(amount=100000, duration=10*12, rate=0)
emi_without_interest
```

834

```
total_interest = (emi_with_interest - emi_without_interest) * 10*12
```

```
print("The total interest paid is ${}".format(total_interest))
```

The total interest paid is \$51960.

## ✓ Documenting functions using Docstrings


We can add some documentation within our function using a *docstring*. A docstring is simply a string that appears as the first statement within the function body, and is used by the `help` function. A good docstring describes what the function does, and provides some explanation about the arguments.

```
def loan_emi(amount, duration, rate, down_payment=0):
    """Calculates the equal montly installment (EMI) for a loan.

    Arguments:
        amount - Total amount to be spent (loan + down payment)
        duration - Duration of the loan (in months)
        rate - Rate of interest (monthly)
        down_payment (optional) - Optional intial payment (deducted from amount)
    """
    loan_amount = amount - down_payment
    try:
        emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    except ZeroDivisionError:
        emi = loan_amount / duration
    emi = math.ceil(emi)
    return emi
```

In the docstring above, we've provided some additional information that the `duration` and `rate` are both measured in months. You might even consider naming the arguments `duration_months` and `rate_monthly`, to avoid any confusion whatsoever. Can you think of some other ways in which the function can be improved?

```
help(loan_emi)
```

 Help on function loan\_emi in module \_\_main\_\_:

```
loan_emi(amount, duration, rate, down_payment=0)
    Calculates the equal montly installment (EMI) for a loan.
```

Arguments:

```
    amount - Total amount to be spent (loan + down payment)
    duration - Duration of the loan (in months)
    rate - Rate of interest (monthly)
    down_payment (optional) - Optional intial payment (deducted from amount)
```

## ✓ Save and upload your notebook

Whether you're running this Jupyter notebook on an online service like Binder or on your local machine, it's important to save your work from time, so that you can access it later, or share it online. You can upload this notebook to your [Jovian.ml](https://jovian.ml) account using the `jovian` Python library.

```
# Instal the library
!pip install jovian --upgrade --quiet
```

```
# Import the jovian module
import jovian
```

```
jovian.commit(project='python-functions-and-scope')
```



## Summary and Further Reading

With this we complete our discussion of functions in Python. We've covered the following topics in this tutorial:

- Creating and using functions
- Functions with one or more arguments
- Local variables and scope
- Returning values using `return`
- Using default arguments to make a function flexible
- Using named arguments while invoking a function
- Importing modules and using library functions

- Reusing and improving functions to handle new use cases
- Handling exceptions with `try - except`
- Documenting functions using docstrings

This is by no means an exhaustive or comprehensive tutorial on functions in Python. Here are few more topics to learn about:

- Functions with an arbitrary number of arguments using (`*args` and `**kwargs`)
- Defining functions inside functions (and closures)
- A function that invokes itself (recursion)
- Functions that accept other functions as arguments or return other functions
- Functions that enhance other functions (decorators)

Following are some resources to learn about more functions in Python:

- Python Tutorial at W3Schools: <https://www.w3schools.com/python/>
- Practical Python Programming: <https://dabeaz-course.github.io/practical-python/Notes/Contents.html>
- Python official documentation: <https://docs.python.org/3/tutorial/index.html>

You are ready to move on to the next tutorial: ["Reading from and writing to files using Python"](#).