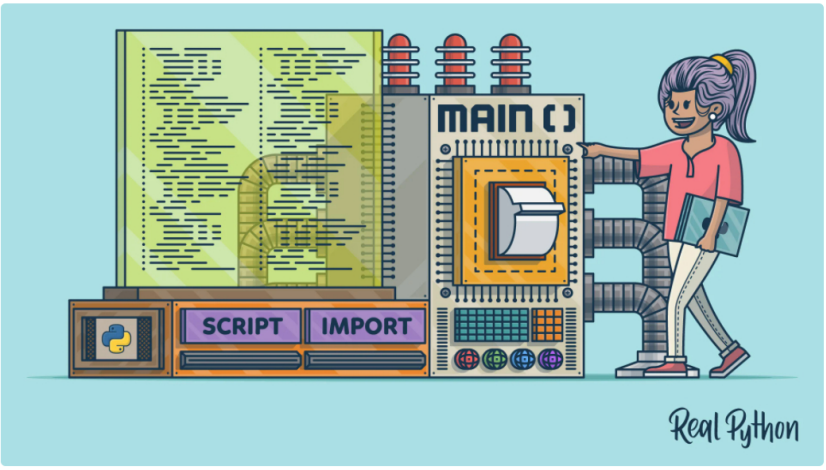

Defining Main Functions in Python

by Bryan Weber May 01, 2019 33 Comments best-practices intermediate

Mark as Completed

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Defining Main Functions in Python**

Many programming languages have a special function that is automatically executed when an operating system starts to run a program. This function is usually called `main()` and must have a specific return type and arguments according to the language standard. On the other hand, the Python interpreter executes scripts starting at the top of the file, and there is no specific function that Python automatically executes.

Nevertheless, having a defined starting point for the execution of a program is useful for understanding how a program works. Python programmers have come up with several conventions to define this starting point.

By the end of this article, you'll understand:

- What the special `__name__` variable is and how Python defines it
- Why you would want to use a `main()` in Python
- What conventions there are for defining `main()` in Python

Table of Contents
- A Basic Python main()
- Execution Modes in Python
- Best Practices for Python Main Functions
- Conclusion

Mark as Completed

Tweet Share Email

- What the best-practices are for what code to put into your `main()`

Free Download: Get a sample chapter from Python Tricks: The Book that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

A Basic Python main()

In some Python scripts, you may see a function definition and a conditional statement that looks like the example below:

Python

```
def main():
    print("Hello World!")

if __name__ == "__main__":
    main()
```

In this code, there is a function called `main()` that prints the phrase `Hello World!` when the Python interpreter executes it. There is also a conditional (or `if`) statement that checks the value of `__name__` and compares it to the string `"__main__"`. When the `if` statement evaluates to `True`, the Python interpreter executes `main()`. You can read more about conditional statements in Conditional Statements in Python.

This code pattern is quite common in Python files that you want to be **executed as a script** and **imported in another module**. To help understand how this code will execute, you should first understand how the Python interpreter sets `__name__` depending on how the code is being executed.

Execution Modes in Python

There are two primary ways that you can instruct the Python interpreter to execute or use code:

1. You can execute the Python file as a **script** using the command line.
2. You can **import** the code from one Python file into another file or into the interactive interpreter.

You can read a lot more about these approaches in How to Run Your Python Scripts. No matter which way of running your code you're using, Python defines a special variable called `__name__` that contains a string whose value depends on how the code is being used.

We'll use this example file, saved as `execution_methods.py`, to explore how the behavior of the code changes depending on the context:

Python

```
print("This is my file to test Python's execution methods.")
print("The variable __name__ tells me which context this file is running in.")
print("The value of __name__ is:", repr(__name__))
```

In this file, there are three calls to `print()` defined. The first two print some introductory phrases. The third `print()` will first print the phrase `The value of __name__ is`, and then it will print the representation of the `__name__` variable using Python's built-in `repr()`.

In Python, `repr()` displays the printable representation of an object. This example uses `repr()` to emphasize that the value of `__name__` is a string. You can read more about `repr()` in the Python documentation.

You'll see the words **file**, **module**, and **script** used throughout this article. Practically, there

isn't much difference between them. However, there are slight differences in meaning that emphasize the purpose of a piece of code:

1. **File:** Typically, a Python file is any file that contains code. Most Python files have the extension `.py`.
2. **Script:** A Python script is a file that you intend to execute from the command line to accomplish a task.
3. **Module:** A Python module is a file that you intend to import from within another module or a script, or from the interactive interpreter. You can read more about modules in Python Modules and Packages – An Introduction.

This distinction is also discussed in How to Run Your Python Scripts.

Executing From the Command Line

In this approach, you want to execute your Python script from the command line.

When you execute a script, you will not be able to interactively define the code that the Python interpreter is executing. The details of how you can execute Python from your command line are not that important for the purpose of this article, but you can expand the box below to read more about the differences between the command line on Windows, Linux, and macOS.

Command line environments Show/Hide

Now you should execute the `execution_methods.py` script from the command line, as shown below:

Shell

```
$ python3 execution_methods.py
This is my file to test Python's execution methods.
The variable __name__ tells me which context this file is running in.
The value of __name__ is: '__main__'
```

In this example, you can see that `__name__` has the value `'__main__'`, where the quote symbols (`'`) tell you that the value has the string type.

Remember that, in Python, there is no difference between strings defined with single quotes (`'`) and double quotes (`"`). You can read more about defining strings in Basic Data Types in Python.

You will find identical output if you include a shebang line in your script and execute it directly (`./execution_methods.py`), or use the `%run` magic in IPython or Jupyter Notebooks.

You may also see Python scripts executed from within packages by adding the `-m` argument to the command. Most often, you will see this recommended when you're using `pip`: `python3 -m pip install package_name`.

Adding the `-m` argument runs the code in the `__main__.py` module of a package. You can find more information about the `__main__.py` file in How to Publish an Open-Source Python Package to PyPI.

In all three of these cases, `__name__` has the same value: the string `'__main__'`.

Technical detail: The Python documentation defines specifically when `__name__` will have the value `'__main__'`:

> A module's `__name__` is set equal to `'__main__'` when read from standard input, a script, or from an interactive prompt. (Source)

`__name__` is stored in the global namespace of the module along with the `__doc__`, `__package__`, and other attributes. You can read more about these attributes in the Python Data Model documentation and, specifically for modules and packages, in the

Python Import documentation.

Importing Into a Module or the Interactive Interpreter

Now let's take a look at the second way that the Python interpreter will execute your code: imports. When you are developing a module or script, you will most likely want to take advantage of modules that someone else has already built, which you can do with the `import` keyword.

During the import process, Python executes the statements defined in the specified module (but only the *first* time you import a module). To demonstrate the results of importing your `execution_methods.py` file, start the interactive Python interpreter and then import your `execution_methods.py` file:

```
Python >>>

>>> import execution_methods
This is my file to test Python's execution methods.
The variable __name__ tells me which context this file is running in.
The value of __name__ is: 'execution_methods'
```

In this code output, you can see that the Python interpreter executes the three calls to `print()`. The first two lines of output are exactly the same as when you executed the file as a script on the command line because there are no variables in either of the first two lines. However, there is a difference in the output from the third `print()`.

When the Python interpreter imports code, the value of `__name__` is set to be the same as the name of the module that is being imported. You can see this in the third line of output above. `__name__` has the value `'execution_methods'`, which is the name of the `.py` file that Python is importing from.

Note that if you `import` the module again without quitting Python, there will be no output.

> **Note:** For more information on how importing works in Python, check out Python import: Advanced Techniques and Tips as well as Absolute vs Relative Imports in Python.

Best Practices for Python Main Functions

Now that you can see the differences in how Python handles its different execution modes, it's useful for you to know some best practices to use. These will apply whenever you want to write code that you can run as a script *and* import in another module or an interactive session.

You will learn about four best practices to make sure that your code can serve a dual purpose:

1. Put most code into a function or class.
2. Use `__name__` to control execution of your code.
3. Create a function called `main()` to contain the code you want to run.
4. Call other functions from `main()`.

Put Most Code Into a Function or Class

Remember that the Python interpreter executes all the code in a module when it imports the module. Sometimes the code you write will have side effects that you want the user to control, such as:

- Running a computation that takes a long time
- Writing to a file on the disk

- Writing to a file on the disk
- Printing information that would clutter the user's terminal

In these cases, you want the user to control triggering the execution of this code, rather than letting the Python interpreter execute the code when it imports your module.

Therefore, the best practice is to **include most code inside a function or a class**. This is because when the Python interpreter encounters the `def` or `class` keywords, it only stores those definitions for later use and doesn't actually execute them until you tell it to.

Save the code below to a file called `best_practices.py` to demonstrate this idea:

Python

```python
from time import sleep

print("This is my file to demonstrate best practices.")

def process_data(data):
    print("Beginning data processing...")
    modified_data = data + " that has been modified"
    sleep(3)
    print("Data processing finished.")
    return modified_data
```

In this code, you first import `sleep()` from the `time` module.

`sleep()` pauses the interpreter for however many seconds you give as an argument and will produce a function that takes a long time to run for this example. Next, you use `print()` to print a sentence describing the purpose of this code.

Then, you define a function called `process_data()` that does five things:

1. Prints some output to tell the user that the data processing is starting
2. Modifies the input data
3. Pauses the execution for three seconds using `sleep()`
4. Prints some output to tell the user that the processing is finished
5. Returns the modified data

Execute the Best Practices File on the Command Line

Now, what will happen when you execute this file as a script on the command line?

The Python interpreter will execute the `from time import sleep` and `print()` lines that are outside the function definition, then it will create the definition of the function called `process_data()`. Then, the script will exit without doing anything further, because the script does not have any code that executes `process_data()`.

The code block below shows the result of running this file as a script:

Shell

```
$ python3 best_practices.py
This is my file to demonstrate best practices.
```

The output that we can see here is the result of the first `print()`. Notice that importing from `time` and defining `process_data()` produce no output. Specifically, the outputs of the calls to `print()` that are inside the definition of `process_data()` are not printed!

Import the Best Practices File in Another Module or the Interactive Interpreter

When you import this file in an interactive session (or another module), the Python interpreter will perform exactly the same steps as when it executes file as a script.

Once the Python interpreter imports the file, you can use any variables, classes, or functions defined in the module you've imported. To demonstrate this, we will use the interactive Python interpreter. Start the interactive interpreter and then type `import best_practices`:

Python >>>

```
>>> import best_practices
```

```
This is my file to demonstrate best practices.
```

The only output from importing the `best_practices.py` file is from the first `print()` call defined outside `process_data()`. Importing from `time` and defining `process_data()` produce no output, just like when you executed the code from the command line.

Use `if __name__ == "__main__"` to Control the Execution of Your Code

What if you want `process_data()` to execute when you run the script from the command line but not when the Python interpreter imports the file?

You can **use the `if __name__ == "__main__"` idiom to determine the execution context** and conditionally run `process_data()` only when `__name__` is equal to `"__main__"`. Add the code below to the bottom of your `best_practices.py` file:

Python

```
11 if __name__ == "__main__":
12     data = "My data read from the Web"
13     print(data)
14     modified_data = process_data(data)
15     print(modified_data)
```

In this code, you've added a conditional statement that checks the value of `__name__`. This conditional will evaluate to `True` when `__name__` is equal to the string `"__main__"`. Remember that the special value of `"__main__"` for the `__name__` variable means the Python interpreter is executing your script and not importing it.

Inside the conditional block, you have added four lines of code (lines 12, 13, 14, and 15):

- **Lines 12 and 13:** You are creating a variable `data` that stores the data you've acquired from the Web and printing it.
- **Line 14:** You are processing the data.
- **Line 15:** You are printing the modified data.

Now, run your `best_practices.py` script from the command line to see how the output will change:

Shell

```
$ python3 best_practices.py
This is my file to demonstrate best practices.
My data read from the Web
Beginning data processing...
Data processing finished.
My data read from the Web that has been modified
```

First, the output shows the result of the `print()` call outside of `process_data()`.

After that, the value of `data` is printed. This happened because the variable `__name__` has the value `"__main__"` when the Python interpreter executes the file as a script, so the conditional statement evaluated to `True`.

Next, your script called `process_data()` and passed `data` in for modification. When `process_data()` executes, it prints some status messages to the output. Finally, the value of `modified_data` is printed.

Now you should check what happens when you import the `best_practices.py` file from the interactive interpreter (or another module). The example below demonstrates this situation:

Python >>>

```
>>> import best_practices
```

```
This is my file to demonstrate best practices.
```

Notice that you get the same behavior as before you added the conditional statement to the end of the file! This is because the `__name__` variable had the value `"best_practices"`, so Python did not execute the code inside the block, including `process_data()`, because the conditional statement evaluated to `False`.

Create a Function Called main() to Contain the Code You Want to Run

Now you are able to write Python code that can be run from the command line as a script and imported without unwanted side effects. Next, you are going to learn about how to write your code to make it easy for other Python programmers to follow what you mean.

Many languages, such as C, C++, Java, and several others, define a special function that must be called `main()` that the operating system automatically calls when it executes the compiled program. This function is often called the **entry point** because it is where execution enters the program.

By contrast, Python does not have a special function that serves as the entry point to a script. You can actually give the entry point function in a Python script any name you want!

Although Python does not assign any significance to a function named `main()`, the best practice is to **name the entry point function `main()`** anyways. That way, any other programmers who read your script immediately know that this function is the starting point of the code that accomplishes the primary task of the script.

In addition, `main()` should contain any code that you want to run when the Python interpreter executes the file. This is better than putting the code directly into the conditional block because a user can reuse `main()`if they import your module.

Change the `best_practices.py` file so that it looks like the code below:

Python

```python
from time import sleep

print("This is my file to demonstrate best practices.")

def process_data(data):
    print("Beginning data processing...")
    modified_data = data + " that has been modified"
    sleep(3)
    print("Data processing finished.")
    return modified_data

def main():
    data = "My data read from the Web"
    print(data)
    modified_data = process_data(data)
    print(modified_data)

if __name__ == "__main__":
    main()
```

In this example, you added the definition of `main()` that includes the code that was previously inside the conditional block. Then, you changed the conditional block so that it executes `main()`. If you run this code as a script or import it, you will get the same output as in the previous section.

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can

including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or another user) to re-use a few of the steps and ignore the ones you don't want. Then you can

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or another user) to re-use a few of the steps and ignore the ones you don't want. Then you can

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or another user) to re-use a few of the steps and ignore the ones you don't want. Then you can

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or another user) to re-use a few of the steps and ignore the ones you don't want. Then you can

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API

2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following: