**\_\_main\_\_ and \_\_name\_\_ in Python**

A Program written in languages of C family (C, C++, Java, C# etc.) needs the main() function to indicate the starting point of execution.

In Python, on the other hand, there is no concept of the main() function, as it is an interpreter based language and can be equally used in an [interactive shell](https://www.tutorialsteacher.com/python/python-interective-shell). The Python program file with .py extension contains multiple statements. The execution of the Python program file starts from the first statement.

Python includes the special variable called \_\_name\_\_ that contains the scope of the code being executed as a string. \_\_main\_\_ is the name of the top-level scope in which top-level code executes.

For example, the scope of the code executed in the interpreter shell will be \_\_main\_\_, as shown below.

Python Shell

>>>\_\_name\_\_

'\_\_main\_\_'

All the functions and modules will be executed in the top-level scope \_\_main\_\_\_ in the interpreter shell.

Python Shell

>>> def f1():

print(\_\_name\_\_)

>>> f1()

Even the inner functions are executed in the top-level scope \_\_main\_\_:

Python Shell

>>> def f1():

print(\_\_name\_\_)

def f2():

print(\_\_name\_\_)

f2()

>>> f1()

\_\_main\_\_

\_\_main\_\_

A Python file can contain multiple functions and statements that can be executed independently. For example, consider the following addition.py:

addition.py

def add(x,y):

z=x+y

print('add() executed under the scope: ', \_\_name\_\_)

return z

x=input('Enter the first number to add: ')

y=input('Enter the secode number to add: ')

result = add(int(x),int(y))

print(x, '+', y,'=', result)

print('Code executed under the scope: ', \_\_name\_\_)

Python program file can be executed in the following ways:

1. Use the command prompt/terminal to execute the Python file as a script.
2. Import Python code from one file to another using the import statement

C:\Python37> python addition.py   
Enter the first number to add: 3   
Enter the secode number to add: 3   
add() executed under the scope: \_\_main\_\_   
3 + 3 = 6   
Code executed under the scope: \_\_main\_\_

As you can see, the addition.py executed under the top-level scope \_\_main\_\_.

The addition.py file can be used as a module in another file or in interactive shell by importing it.

Let's see what happens when you import the addition module in the interactive shell.

Python Shell

>>> import addition

Enter the first number to add: 3

Enter the secode number to add: 3

add() executed under the scope: addition

3 + 3 = 6

Code executed under the scope: addition

Above, the import statement starts executing from the first statement. But, we only want to use the add() method and don't want to execute the other statements.

Here we can use the special variable \_\_name\_\_ to check the scope and execute the statements of the addition.py file only when it executes from the command prompt/terminal independently but not when imported it in some other file/module. Rewrite the addition.py, as shown below.

addition.py

def add(x, y):

z=x+y

print('add() executed under the scope: ', \_\_name\_\_)

return z

if \_\_name\_\_ == '\_\_main\_\_':

x=input('Enter the first number to add: ')

y=input('Enter the secode number to add: ')

result = add(int(x),int(y))

print(x, '+', y,'=', result)

print('Code executed under the scope: ', \_\_name\_\_)

Above, the if condition check that if the scope is \_\_main\_\_ then only execute the code that takes user's inputs and adds them.

Now, let's see what happens when we import the above addition module in the interactive shell.

Python Shell

>>> import addition

>>> addition.add(3,3)

add() executed under the scope: addition

6

You can also use the from import statement, as shown below:

Python Shell

>>> from addition import add

>>> add(3,3)

add() executed under the scope: addition

6

As you can see, because we used an if condition to check the scope, it does not execute user input codes after importing the addition module, because it executes under the module's scope, which is addition scope. It only imports the add() method. The same thing will happen when you import the addition module in other modules.

Now, let's see what happens when you execute it from the command prompt/terminal.

C:\Python37> python addition.py   
Enter the first number to add: 3   
Enter the secode number to add: 3   
add() executed under the scope: \_\_main\_\_   
3 + 3 = 6   
Code executed under the scope: \_\_main\_\_

As you can see, it still executes the same code because of addition.py being executed in the top-level scope \_\_main\_\_.

Thus, value of the name allows the Python interpreter to determine whether a module is intended to be an executable script or not. If its value is main, the statements outside function definitions will be executed. If not, the contents of the module are populated in top-level module (or interpreter namespace) without the executable part.

Note: The Python script file executing from the command prompt/terminal will be executed under the top-level scope \_\_main\_\_ scope. However, importing a module will be executed under the module's own scope. So, the top-level scope will be \_\_main\_\_, and the second scope would be module's scope.

Thus, using the special variable \_\_name\_\_ and the top-level scope \_\_main\_\_ increases the reusability. The Python script file can be executed from the command prompt/termainal as an indipendent script as well as when imported as a module.

**Interpreted vs Compiled Programming Languages: What's the Difference?**

Every program is a set of instructions, whether it’s to add two numbers or send a request over the internet. Compilers and interpreters take human-readable code and convert it to computer-readable machine code.

In a compiled language, the target machine directly translates the program. In an interpreted language, the source code is not directly translated by the target machine. Instead, a different program, aka the interpreter, reads and executes the code.

### **Okay… but what does that** actually **mean?**

Imagine you have a hummus recipe that you want to make, but it's written in ancient Greek. There are two ways you, a non-ancient-Greek speaker, could follow its directions.

The first is if someone had already translated it into English for you. You (and anyone else who can speak English) could read the English version of the recipe and make hummus. Think of this translated recipe as the compiled version.

The second way is if you have a friend who knows ancient Greek. When you're ready to make hummus, your friend sits next to you and translates the recipe into English as you go, line by line. In this case, your friend is the interpreter for the interpreted version of the recipe.

### **Compiled Languages**

Compiled languages are converted directly into machine code that the processor can execute. As a result, they tend to be faster and more efficient to execute than interpreted languages. They also give the developer more control over hardware aspects, like memory management and CPU usage.

Compiled languages need a “build” step – they need to be manually compiled first. You need to “rebuild” the program every time you need to make a change. In our hummus example, the entire translation is written before it gets to you. If the original author decides that he wants to use a different kind of olive oil, the entire recipe would need to be translated again and resent to you.

Examples of pure compiled languages are C, C++, Erlang, Haskell, Rust, and Go.

### **Interpreted Languages**

Interpreters run through a program line by line and execute each command. Here, if the author decides he wants to use a different kind of olive oil, he could scratch the old one out and add the new one. Your translator friend can then convey that change to you as it happens.

Interpreted languages were once significantly slower than compiled languages. But, with the development of [just-in-time compilation](https://guide.freecodecamp.org/computer-science/just-in-time-compilation), that gap is shrinking.

Examples of common interpreted languages are PHP, Ruby, Python, and JavaScript.

### **A Small Caveat**

Most programming languages can have both compiled and interpreted implementations – the language itself is not necessarily compiled or interpreted. However, for simplicity’s sake, they’re typically referred to as such.

Python, for example, can be executed as either a compiled program or as an interpreted language in interactive mode. On the other hand, most command line tools, CLIs, and shells can theoretically be classified as interpreted languages.

## Advantages and disadvantages

### Advantages of compiled languages

Programs that are compiled into native machine code tend to be faster than interpreted code. This is because the process of translating code at run time adds to the overhead, and can cause the program to be slower overall.

### Disadvantages of compiled languages

The most notable disadvantages are:

* Additional time needed to complete the entire compilation step before testing
* Platform dependence of the generated binary code

### Advantages of interpreted languages

Interpreted languages tend to be more flexible, and often offer features like dynamic typing and smaller program size. Also, because interpreters execute the source program code themselves, the code itself is platform independent.

### Disadvantages of interpreted languages

The most notable disadvantage is typical execution speed compared to compiled languages.