What does if __name__ == "__main__": do?

Asked 12 years, 7 months ago Active 20 days ago Viewed 3.6m times

6976

2989

Given the following code, what does the `if __name__ == "__main__":` do?

```
# Threading example
import time, thread

def myfunction(string, sleeptime, lock, *args):
    while True:
        lock.acquire()
        time.sleep(sleeptime)
        lock.release()
        time.sleep(sleeptime)

if __name__ == "__main__":
    lock = thread.allocate_lock()
    thread.start_new_thread(myfunction, ("Thread #: 1", 2, lock))
    thread.start_new_thread(myfunction, ("Thread #: 2", 2, lock))
```

python namespaces main python-module idioms

Share Improve this question Follow

edited Feb 26 '20 at 21:04
iliketocode
6,722 5 42 57

asked Jan 7 '09 at 4:11
Devoted
92k 41 85 110

3 Just for the record - what is "**main**": docs.python.org/3/reference/… and what is "**name**": docs.python.org/3/reference/… – bruziuz Apr 2 at 9:17

38 Answers

Active Oldest Votes

1 2 Next

7696

Short Answer

It's boilerplate code that protects users from accidentally invoking the script when they didn't intend to. Here are some common problems when the guard is omitted from a script:

- If you import the guardless script in another script (e.g. `import my_script_without_a_name_eq_main_guard`), then the second script will trigger the first to run *at import time* and *using the second script's command line arguments*. This is almost always a mistake.
- If you have a custom class in the guardless script and save it to a pickle file, then unpickling it in another script will trigger an import of the guardless script, with the same problems outlined in the previous bullet.

Long Answer

To better understand why and how this matters, we need to take a step back to understand how Python initializes scripts and how this interacts with its module import mechanism.

Whenever the Python interpreter reads a source file, it does two things:

- it sets a few special variables like `__name__`, and then
- it executes all of the code found in the file.

Let's see how this works and how it relates to your question about the `__name__` checks we always see in Python scripts.

Code Sample

Let's use a slightly different code sample to explore how imports and scripts work. Suppose the following is in a file called `foo.py`.

```
# Suppose this is foo.py.

print("before import")
import math

print("before functionA")
def functionA():
```

Special Variables

When the Python interpreter reads a source file, it first defines a few special variables. In this case, we care about the `__name__` variable.

When Your Module Is the Main Program

If you are running your module (the source file) as the main program, e.g.

```
python foo.py
```

the interpreter will assign the hard-coded string `"__main__"` to the `__name__` variable, i.e.

```
# It's as if the interpreter inserts this at the top
# of your module when run as the main program.
__name__ = "__main__"
```

When Your Module Is Imported By Another

On the other hand, suppose some other module is the main program and it imports your module. This means there's a statement like this in the main program, or in some other module the main program imports:

```
# Suppose this is in some other main program.
import foo
```

The interpreter will search for your `foo.py` file (along with searching for a few other variants), and prior to executing that module, it will assign the name `"foo"` from the import statement to the `__name__` variable, i.e.

```
# It's as if the interpreter inserts this at the top
# of your module when it's imported from another module.
__name__ = "foo"
```

Executing the Module's Code

After the special variables are set up, the interpreter executes all the code in the module, one statement at a time. You may want to open another window on the side with the code

```
math = __import__("math")
```

3. It prints the string `"before functionA"`.
4. It executes the `def` block, creating a function object, then assigning that function object to a variable called `functionA`.
5. It prints the string `"before functionB"`.
6. It executes the second `def` block, creating another function object, then assigning it to a variable called `functionB`.
7. It prints the string `"before __name__ guard"`.

Only When Your Module Is the Main Program

8. If your module is the main program, then it will see that `__name__` was indeed set to `"__main__"` and it calls the two functions, printing the strings `"Function A"` and `"Function B 10.0"`.

Only When Your Module Is Imported by Another

8. (**instead**) If your module is not the main program but was imported by another one, then `__name__` will be `"foo"`, not `"__main__"`, and it'll skip the body of the `if` statement.

Always

9. It will print the string `"after __name__ guard"` in both situations.

Summary

In summary, here's what'd be printed in the two cases:

```
# What gets printed if foo is the main program
before import
before functionA
before functionB
before __name__ guard
```

- Your module is a library, but you want to have a script mode where it runs some unit tests or a demo.
- Your module is only used as a main program, but it has some unit tests, and the testing framework works by importing `.py` files like your script and running special test functions. You don't want it to try running the script just because it's importing the module.
- Your module is mostly used as a main program, but it also provides a programmer-friendly API for advanced users.

Beyond those examples, it's elegant that running a script in Python is just setting up a few magic variables and importing the script. "Running" the script is a side effect of importing the script's module.

Food for Thought

- Question: Can I have multiple `__name__` checking blocks? Answer: it's strange to do so, but the language won't stop you.
- Suppose the following is in `foo2.py`. What happens if you say `python foo2.py` on the command-line? Why?

```
# Suppose this is foo2.py.
import os, sys; sys.path.insert(0, os.path.dirname(__file__)) # needed for some
interpreters

def functionA():
    print("a1")
    from foo2 import functionB
    print("a2")
```

```
def functionA():
    print("b")

print("t1")
print("m1")
functionA()
print("m2")
print("t2")
```

- What will this do when used as a script? When imported as a module?

```
# Suppose this is in foo4.py
__name__ = "__main__"

def bar():
    print("bar")

print("before __name__ guard")
if __name__ == "__main__":
    bar()
print("after __name__ guard")
```

Share Improve this answer Follow

edited Apr 5 at 21:28

answered Jan 7 '09 at 4:26

Mr Fooz

97.5k 5 65 95

1981

When your script is run by passing it as a command to the Python interpreter,

```
python myscript.py
```

all of the code that is at indentation level 0 gets executed. Functions and classes that are defined are, well, defined, but none of their code gets run. Unlike other languages, there's no `main()` function that gets run automatically - the `main()` function is implicitly all the code at the top level.

In this case, the top-level code is an `if` block. `__name__` is a built-in variable which evaluates to the name of the current module. However, if a module is being run directly (as in `myscript.py` above), then `__name__` instead is set to the string `"__main__"`. Thus, you can test whether your script is being run directly or being imported by something else by testing

```
if __name__ == "__main__":
    ...
```

If your script is being imported into another module, its various function and class definitions will be imported and its top-level code will be executed, but the code in the then-body of the `if` clause above won't get run as the condition is not met. As a basic example, consider the following two scripts:

```
# file one.py
def func():
    print("func() in one.py")

print("top-level in one.py")

if __name__ == "__main__":
    print("one.py is being run directly")
else:
    print("one.py is being imported into another module")



# file two.py
import one

print("top-level in two.py")
one.func()

if __name__ == "__main__":
    print("two.py is being run directly")
else:
    print("two.py is being imported into another module")
```

If you run `two.py` instead:

```
python two.py
```

You get

```
top-level in one.py
one.py is being imported into another module
top-level in two.py
func() in one.py
two.py is being run directly
```

Thus, when module `one` gets loaded, its `__name__` equals `"one"` instead of `"__main__"`.

Share Improve this answer Follow

edited Jan 31 '18 at 13:28
Tonechas
11.7k 15 37 68

answered Jan 7 '09 at 4:28
Adam Rosenfield
364k 94 489 575

So, `if __name__ == "__main__":` basically checks if you are running your python script itself, and not importing it or something? – Merp Feb 18 at 18:20

796

The simplest explanation for the `__name__` variable (imho) is the following:

Create the following files.

```
# a.py
import b
```

and

```
# b.py
print "Hello World from %s!" % __name__

if __name__ == '__main__':
    print "Hello World again from %s!" % __name__
```

Running them will get you this output:

```
$ python a.py
Hello World from b!
```

As you can see, when a module is imported, Python sets `globals()['__name__']` in this module to the module's name. Also, upon import all the code in the module is being run. As the `if` statement evaluates to `False` this part is not executed.

```
$ python b.py
Hello World from __main__!
Hello World again from __main__!
```

As you can see, when a file is executed, Python sets `globals()['__name__']` in this file to `"__main__"`. This time, the `if` statement evaluates to `True` and is being run.

Share Improve this answer Follow edited Dec 18 '18 at 9:25 answered Jan 7 '09 at 11:35

pi.
19.4k 7 36 59

554

What does the `if __name__ == "__main__":` do?

To outline the basics:

- The global variable, `__name__`, in the module that is the entry point to your program, is `'__main__'`. Otherwise, it's the name you import the module by.
- So, code under the `if` block will only run if the module is the entry point to your program.
- It allows the code in the module to be importable by other modules, without executing the code block beneath on import.

Why do we need this?

Developing and Testing Your Code

Say you're writing a Python script designed to be used as a module:

```
def do_important():
    """This function does something very important"""
```

You *could* test the module by adding this call of the function to the bottom:

```
do_important()
```

and running it (on a command prompt) with something like:

```
~$ python important.py
```

The Problem

However, if you want to import the module to another script:

```
import important
```

On import, the `do_important` function would be called, so you'd probably comment out your

A Better Way

The `__name__` variable points to the namespace wherever the Python interpreter happens to be at the moment.

Inside an imported module, it's the name of that module.

But inside the primary module (or an interactive Python session, i.e. the interpreter's Read, Eval, Print Loop, or REPL) you are running everything from its `"__main__"`.

So if you check before executing:

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

With the above, your code will only execute when you're running it as the primary module (or intentionally call it from another script).

An Even Better Way

There's a Pythonic way to improve on this, though.

What if we want to run this business process from outside the module?

If we put the code we want to exercise as we develop and test in a function like this and then do our check for `'__main__'` immediately after:

```
def main():
    """business logic for when running this module as the primary one!"""
    setup()
    foo = do_important()
    bar = do_even_more_important(foo)
    for baz in bar:
        do_super_important(baz)
    teardown()

# Here's our payoff idiom!
if __name__ == '__main__':
    main()
```

We now have a final function for the end of our module that will run if we run the module as the primary module.

This module represents the (otherwise anonymous) scope in which the interpreter's main program executes — commands read either from standard input, from a script file, or from an interactive prompt. It is this environment in which the idiomatic "conditional script" stanza causes a script to run:

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

Share Improve this answer Follow edited Mar 27 '18 at 2:27 answered Nov 23 '13 at 4:38

Aaron Hall ♦
302k 75 374 314

145

`if __name__ == "__main__"` is the part that runs when the script is run from (say) the command line using a command like `python myscript.py`.

Share Improve this answer Follow edited Jul 10 '15 at 15:49 answered Jan 7 '09 at 4:14

Mark Amery
115k 61 361 414

Harley Holcombe
158k 15 68 62

3 Why does a file `helloworld.py` with just `print("hello world")` in it can run with command `python helloworld.py` even when there is no `if __name__ == "__main__"` ? – hi15 Aug 22 '19 at 16:39

1 When you run `python helloworld.py` it will run the whole script file (whether you specify `if __name__ == "__main__"` or not) . There is only a difference in execution when you are importing `helloworld.py` from a different script. In that case the `if __name__ == "__main__"` codeblock does not execute at all. – Nihal Sangeeth Dec 29 '20 at 14:09

94

What does `if __name__ == "__main__":` do?

`__name__` is a global variable (in Python, global actually means on the module level) that exists in all namespaces. It is typically the module's name (as a `str` type).

As the only special case, however, in whatever Python process you run, as in mycode.py:

```
python mycode.py
```

the otherwise anonymous global namespace is assigned the value of `'__main__'` to its `__name__`.

Thus, including the final lines

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

- at the end of your mycode.py script,
- when it is the primary, entry-point module that is run by a Python process,

will cause your script's uniquely defined `main` function to run.

Another benefit of using this construct: you can also import your code as a module in another script and then run the main function if and when your program decides:

```
import mycode
# ... any amount of other code
mycode.main()
```

Share Improve this answer Follow

edited Jan 10 '17 at 17:35
coffee-grinder
25.2k 19 53 81

answered Oct 14 '14 at 20:22
Aaron Hall ♦
302k 75 374 314

86

There are lots of different takes here on the mechanics of the code in question, the "How", but for me none of it made sense until I understood the "Why". This should be especially helpful for new programmers.

Take file "ab.py":

```
def a():
    print('A function in ab file');
a()
```

And a second file "xy.py":

```
import ab
def main():
    print('main function: this is where the action is')
def x():
    print ('peripheral task: might be useful in other projects')
x()
if __name__ == "__main__":
    main()
```

What is this code actually doing?

When you execute `xy.py`, you `import ab`. The import statement runs the module immediately on import, so `ab`'s operations get executed before the remainder of `xy`'s. Once finished with `ab`, it continues with `xy`.

The interpreter keeps track of which scripts are running with `__name__`. When you run a script - no matter what you've named it - the interpreter calls it `"__main__"`, making it the master or 'home' script that gets returned to after running an external script.

Any other script that's called from this `"__main__"` script is assigned its filename as its `__name__` (e.g., `__name__ == "ab.py"`). Hence, the line `if __name__ == "__main__":` is the interpreter's test to determine if it's interpreting/parsing the 'home' script that was initially executed, or if it's temporarily peeking into another (external) script. This gives the programmer flexibility to have the script behave differently if it's executed directly vs. called externally.

Let's step through the above code to understand what's happening, focusing first on the unindented lines and the order they appear in the scripts. Remember that function - or `def` -

- Oh, a function. I'll remember that.
- Another one.
- Function `x()` ; ok, printing '*peripheral task: might be useful in other projects*'.
- What's this? An `if` statement. Well, the condition has been met (the variable `__name__` has been set to `"__main__"`), so I'll enter the `main()` function and print '*main function: this is where the action is*'.

The bottom two lines mean: "If this is the `"__main__"` or 'home' script, execute the function called `main()` ". That's why you'll see a `def main():` block up top, which contains the main flow of the script's functionality.

Why implement this?

Remember what I said earlier about import statements? When you import a module it doesn't just 'recognize' it and wait for further instructions - it actually runs all the executable operations contained within the script. So, putting the meat of your script into the `main()` function effectively quarantines it, putting it in isolation so that it won't immediately run when imported by another script.

Again, there will be exceptions, but common practice is that `main()` doesn't usually get called externally. So you may be wondering one more thing: if we're not calling `main()` , why are we calling the script at all? It's because many people structure their scripts with standalone functions that are built to be run independent of the rest of the code in the file. They're then later called somewhere else in the body of the script. Which brings me to this:

But the code works without it

Yes, that's right. These separate functions **can** be called from an in-line script that's not contained inside a `main()` function. If you're accustomed (as I am, in my early learning stages of programming) to building in-line scripts that do exactly what you need, and you'll try to figure it out again if you ever need that operation again ... well, you're not used to this kind of internal structure to your code, because it's more complicated to build and it's not as intuitive to read.

But that's a script that probably can't have its functions called externally, because if it did it would immediately start calculating and assigning variables. And chances are if you're trying to re-use a function, your new script is related closely enough to the old one that there will be

Share Improve this answer Follow edited May 23 '18 at 22:29 answered Sep 29 '16 at 4:33

joechoj
1,159 7 12

61

When there are certain statements in our module (`M.py`) we want to be executed when it'll be running as main (not imported), we can place those statements (test-cases, print statements) under this `if` block.

As by default (when module running as main, not imported) the `__name__` variable is set to `"__main__"` , and when it'll be imported the `__name__` variable will get a different value, most probably the name of the module (`'M'`). This is helpful in running different variants of a modules together, and separating their specific input & output statements and also if there are any test-cases.

In short, use this ' `if __name__ == "main"` ' block to prevent (certain) code from being run when the module is imported.

Share Improve this answer Follow edited May 23 '18 at 22:07 answered Apr 3 '13 at 14:09

Peter Mortensen
28.7k 21 95 123

Nabeel Ahmed
15k 4 51 54

57

Put simply, `__name__` is a variable defined for each script that defines whether the script is being run as the main module or it is being run as an imported module.

So if we have two scripts;

```
#script1.py
print "Script 1's name: {}".format(__name__)
```

and

```
#script2.py
import script1
print "Script 2's name: {}".format(__name__)
```

The output from executing script1 is

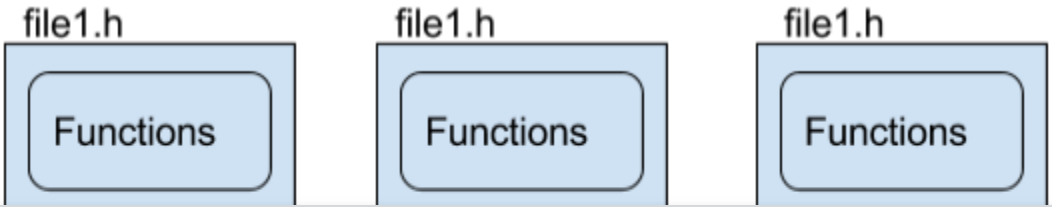
```
Script 1's name: __main__
```

And the output from executing script2 is:

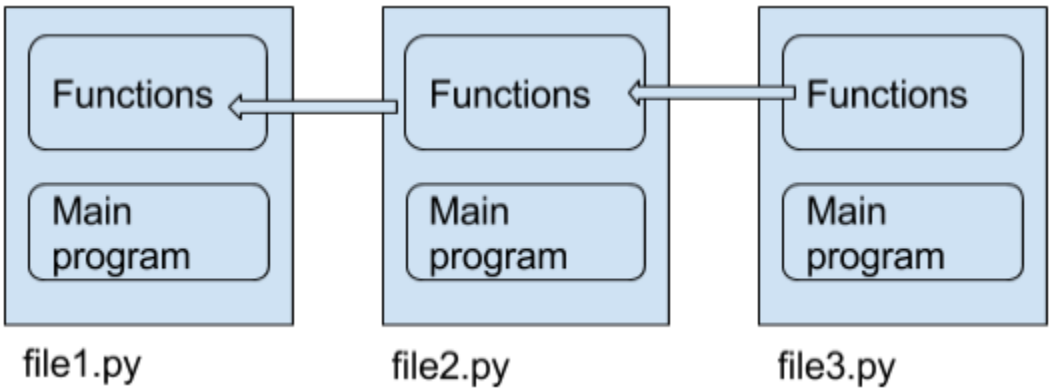
```
Script1's name is script1
Script 2's name: __main__
```

As you can see, `__name__` tells us which code is the 'main' module. This is great, because you can just write code and not have to worry about structural issues like in C/C++, where, if a file does not implement a 'main' function then it cannot be compiled as an executable and if it does, it cannot then be used as a library.

Say you write a Python script that does something great and you implement a boatload of functions that are useful for other purposes. If I want to use them I can just import your script and use them without executing your program (given that your code only executes within the `if __name__ == "__main__":` context). Whereas in C/C++ you would have to portion out those pieces into a separate module that then includes the file. Picture the situation below;



The arrows are import links. For three modules each trying to include the previous modules code there are six files (nine, counting the implementation files) and five links. This makes it difficult to include other code into a C project unless it is compiled specifically as a library. Now picture it for Python:



You write a module, and if someone wants to use your code they just import it and the `__name__` variable can help to separate the executable portion of the program from the library part.

Share Improve this answer Follow

edited May 23 '18 at 22:28
Peter Mortensen
28.7k 21 95 123

answered Oct 15 '16 at 9:07
redbandit
1,903 14 11

2 The C/C++ illustration is wrong: 3 times the same unit name (*file1*). – Wolf Jan 11 '18 at 12:59

52

Let's look at the answer in a more abstract way:

Suppose we have this code in `x.py`:

```
...
<Block A>
if __name__ == '__main__':
    <Block B>
...
```

Blocks A and B are run when we are running `x.py`.

But just block A (and not B) is run when we are running another module, `y.py` for example, in which `x.py` is imported and the code is run from there (like when a function in `x.py` is called from `y.py`).

Share Improve this answer Follow

edited May 27 '20 at 10:50
kubuntu
2,515 1 20 24

answered Jan 20 '15 at 17:48
Alisa
2,354 3 26 43

1 I wasn't able to edit the post (minimum 6 characters if change required). Line 14 has 'x.y' rather than 'x.py'. – fpsdkfsdkmsdfsdfm May 6 '20 at 14:49

46

I've been reading so much throughout the answers on this page. I would say, if you know the thing, for sure you will understand those answers, otherwise, you are still confused.

To be short, you need to know several points:

1. `import a` action actually runs all that can be ran in `a.py`, meaning each line in `a.py`
2. Because of point 1, you may not want everything to be run in `a.py` when importing it
3. To solve the problem in point 2, python allows you to put a condition check
4. `__name__` is an implicit variable in all `.py` modules:

- when `a.py` is `import`ed, the value of `__name__` of `a.py` module is set to its file name "`a`"
- when `a.py` is run directly using "`python a.py`", the value of `__name__` is set to a string `__main__`

5. Based on the mechanism how python sets the variable `__name__` for each module, do you know how to achieve point 3? The answer is fairly easy, right? Put a if condition: `if __name__ == "__main__": // do A`

- then `python a.py` will run the part `// do A`
- and `import a` will skip the part `// do A`

6. You can even put if `__name__ == "a"` depending on your functional need, but rarely do

The important thing that python is special at is point 4! The rest is just basic logic.

Share Improve this answer Follow edited Aug 9 '20 at 22:19 answered Jun 24 '18 at 15:48

jack
1,025 7 20

2 Yes, point 1 is vital to understand. From that, the need for this mechanism become clear. – Eureka Mar 24 '19 at 21:16

Nailed it, was totally confused still with the above answers, but now it's crystal clear! – bad_coder Jan 30 at 21:09

2 this should be the accepted answer.. – Sunil Garg Feb 16 at 10:04

1 This is by far the best "understandable" answer. – Enis Arik Jul 24 at 11:00

41

When you run Python interactively the local `__name__` variable is assigned a value of `__main__`. Likewise, when you execute a Python module from the command line, rather than importing it into another module, its `__name__` attribute is assigned a value of `__main__`, rather than the actual name of the module. In this way, modules can look at their own `__name__` value to determine for themselves how they are being used, whether as support for another program or as the main application executed from the command line. Thus, the following idiom is quite common in Python modules:

```
if __name__ == '__main__':
    # Do something appropriate here, like calling a
    # main() function defined elsewhere in this module.
    main()
else:
    # Do nothing. This module has been imported by another
    # module that wants to make use of the functions,
    # classes and other useful bits it has defined.
```

Share Improve this answer Follow

answered Dec 11 '13 at 11:23
Zain
1,146 1 14 26

39

Consider:

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

It checks if the `__name__` attribute of the Python script is `"__main__"`. In other words, if the program itself is executed, the attribute will be `__main__`, so the program will be executed (in this case the `main()` function).

However, if your Python script is used by a module, any code outside of the `if` statement will be executed, so `if \__name__ == "\__main__"` is used just to check if the program is used as a module or not, and therefore decides whether to run the code.

Share Improve this answer Follow

edited May 23 '18 at 22:31
Peter Mortensen
28.7k 21 95 123

answered Aug 22 '17 at 18:53
Larry
1,112 1 11 19

38

The code under `if __name__ == '__main__':` **will be executed only if the module is invoked as a script**.

As an example consider the following module `my_test_module.py`:

```
# my_test_module.py

print('This is going to be printed out, no matter what')

if __name__ == '__main__':
    print('This is going to be printed out, only if user invokes the module as a
script')
```

1st possibility: Import `my_test_module.py` in another module

```
# main.py

import my_test_module

if __name__ == '__main__':
    print('Hello from main.py')
```

Now if you invoke `main.py`:

```
python main.py

>> 'This is going to be printed out, no matter what'
>> 'Hello from main.py'
```

Note that only the top-level `print()` statement in `my_test_module` is executed.

2nd possibility: Invoke `my_test_module.py` as a script

Now if you run `my_test_module.py` as a Python script, both `print()` statements will be exectued:

```
python my_test_module.py

>>> 'This is going to be printed out, no matter what'
>>> 'This is going to be printed out, only if user invokes the module as a
script'
```

36

Before explaining anything about `if __name__ == '__main__'` it is important to understand what `__name__` is and what it does.

What is `__name__`?

`__name__` is a DunderAlias - can be thought of as a global variable (accessible from modules) and works in a similar way to `global`.

It is a string (global as mentioned above) as indicated by `type(__name__)` (yielding `<class 'str'>`), and is an inbuilt standard for both Python 3 and Python 2 versions.

Where:

It can not only be used in scripts but can also be found in both the interpreter and modules/packages.

Interpreter:

```
>>> print(__name__)
__main__
>>>
```

Script:

test_file.py:

```
print(__name__)
```

Resulting in `__main__`

Module or package:

somefile.py:

```
def somefunction():
    print(__name__)
```

test_file.py:

You should see that, where `__name__`, where it is the main file (or program) will *always* return `__main__`, and if it is a module/package, or anything that is running off some other Python script, will return the name of the file where it has originated from.

Practice:

Being a variable means that it's value *can* be overwritten ("can" does not mean "should"), overwriting the value of `__name__` will result in a lack of readability. So do not do it, for any reason. If you need a variable define a new variable.

It is always assumed that the value of `__name__` to be `__main__` or the name of the file. Once again changing this default value will cause more confusion that it will do good, causing problems further down the line.

example:

```
>>> __name__ = 'Horrify' # Change default from __main__
>>> if __name__ == 'Horrify': print(__name__)
...
>>> else: print('Not Horrify')
...
Horrify
>>>
```

It is considered good practice in general to include the `if __name__ == '__main__'` in scripts.

Now to answer `if __name__ == '__main__':`

Now we know the behaviour of `__name__` things become clearer:

An `if` is a flow control statement that contains the block of code will execute if the value given is true. We have seen that `__name__` can take either `__main__` or the file name it has been imported from.

This means that if `__name__` is equal to `__main__` then the file must be the main file and must actually be running (or it is the interpreter), not a module or package imported into the script.

If indeed `__name__` does take the value of `__main__` then whatever is in that block of code will execute.

It is also possible to do other, less common but useful things with `__name__`, some I will show here:

Executing only if the file is a module or package:

```
if __name__ != '__main__':
    # Do some useful things
```

Running one condition if the file is the main one and another if it is not:

```
if __name__ == '__main__':
    # Execute something
else:
    # Do some useful things
```

You can also use it to provide runnable help functions/utilities on packages and modules without the elaborate use of libraries.

It also allows modules to be run from the command line as main scripts, which can be also very useful.

Share Improve this answer Follow edited May 23 '18 at 22:39 answered Apr 3 '18 at 19:32

Simon
9,382 8 53 78

33

I think it's best to break the answer in depth and in simple words:

`__name__` : Every module in Python has a special attribute called `__name__` . It is a built-in variable that returns the name of the module.

`__main__` : Like other programming languages, Python too has an execution entry point, i.e., main. `'__main__'` *is the name of the scope in which top-level code executes*. Basically you have two ways of using a Python module: Run it directly as a script, or import it. When a module is run as a script, its `__name__` is set to `__main__` .

Thus, the value of the `__name__` attribute is set to `__main__` when the module is run as the main program. Otherwise the value of `__name__` is set to contain the name of the module.

Share Improve this answer Follow

edited May 23 '18 at 22:30
Peter Mortensen
28.7k 21 95 123

answered Nov 30 '16 at 6:47
Inconnu
5,028 2 32 41

31

It is a special for when a Python file is called from the command line. This is typically used to call a "main()" function or execute other appropriate startup code, like commandline arguments handling for instance.

It could be written in several ways. Another is:

```
def some_function_for_instance_main():
    dosomething()

__name__ == '__main__' and some_function_for_instance_main()
```

I am not saying you should use this in production code, but it serves to illustrate that there is nothing "magical" about `if __name__ == '__main__'` .

It just a convention for invoking a main function in Python files.

Share Improve this answer Follow

edited Jun 15 at 15:46

answered Jan 24 '13 at 13:48
Prof. Falken
22.5k 18 95 166

7 I would consider this bad form as you're 1) relying on side effects and 2) abusing `and`. `and` is used for checking if two boolean statements are both true. Since you're not interested in the result of the `and`, an `if` statement more clearly communicates your intentions. – jpmc26 Dec 26 '13 at 18:07

9 Leaving aside the question of whether exploiting the short-circuit behaviour of boolean operators as a flow control mechanism is bad style or not, the bigger problem is that this *doesn't answer the question at all*. – Mark Amery Jul 10 '15 at 15:33

1 @jpmc26 Anyone with a background in Perl or Javascript is totally comfortable with this idiom, using `and` as a control statement. I don't have any issue with it. Another similar idiom is using `or` to set default values. For example, `x = input("what is your name? ") or "Nameless Person"`. – John Henckel Jul 9 at 16:04

@JohnHenckel This is not Perl or JavaScript. This is not a Python idiom. It is considered bad form to use a function with side effects in the middle of a Boolean statement in Python. Particularly in this case, there is absolutely no benefit to using `and` here; the function doesn't even return a value. It just makes the code less obvious. – jpmc26 Jul 9 at 18:51

1 @jpmc26 I'm trying to find an authoritative source that agrees with you. Is this mentioned somewhere? For example in PEP8 does it say that we should avoid using `and` for control purposes, or using `or` to assign a default value? I tried to google it, but I could not find anything. – John Henckel Jul 20 at 18:23

24

There are a number of variables that the system (Python interpreter) provides for source files (modules). You can get their values anytime you want, so, let us focus on the **__name__** variable/attribute:

When Python loads a source code file, it executes all of the code found in it. (Note that it doesn't call all of the methods and functions defined in the file, but it does define them.)

Before the interpreter executes the source code file though, it defines a few special variables for that file; **__name__** is one of those special variables that Python automatically defines for each source code file.

If Python is loading this source code file as the main program (i.e. the file you run), then it sets the special **__name__** variable for this file to have a value **"__main__"**.

If this is being imported from another module, **__name__** will be set to that module's name.

So, in your example in part:

```
if __name__ == "__main__":
   lock = thread.allocate_lock()
   thread.start_new_thread(myfunction, ("Thread #: 1", 2, lock))
   thread.start_new_thread(myfunction, ("Thread #: 2", 2, lock))
```

means that the code block:

```
lock = thread.allocate_lock()
thread.start_new_thread(myfunction, ("Thread #: 1", 2, lock))
thread.start_new_thread(myfunction, ("Thread #: 2", 2, lock))
```

will be executed only when you run the module directly; the code block will not execute if another module is calling/importing it because the value of **__name__** will not equal to "**main**" in that particular instance.

Hope this helps out.

Share Improve this answer Follow edited Jul 20 '16 at 9:30 answered Nov 25 '15 at 12:26

codewizard
326 2 8

22

`if __name__ == "__main__":` is basically the top-level script environment, and it specifies the interpreter that ('I have the highest priority to be executed first').

`'__main__'` is the name of the scope in which top-level code executes. A module's `__name__` is set equal to `'__main__'` when read from standard input, a script, or from an interactive prompt.

```
if __name__ == "__main__":
    # Execute only if run as a script
    main()
```

Share Improve this answer Follow

edited May 23 '18 at 22:14
Peter Mortensen
28.7k 21 95 123

answered Apr 24 '16 at 8:23
The Gr8 Adakron
1,062 1 11 14

21

Consider:

```
print __name__
```

The output for the above is `__main__`.

```
if __name__ == "__main__":
  print "direct method"
```

The above statement is true and prints *"direct method"*. Suppose if they imported this class in another class it doesn't print *"direct method"* because, while importing, it will set `__name__` `equal to "first model name"`.

Share Improve this answer Follow

edited Feb 6 '19 at 23:16
simhumileco
22.9k 14 112 95

answered Jun 22 '16 at 10:47
Janarthanan Ramu
1,095 10 16

19

You can make the file usable as a **script** as well as an **importable module**.

fibo.py (a module named `fibo`)

```
# Other modules can IMPORT this MODULE to use the function fib
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

# This allows the file to be used as a SCRIPT
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

Reference: https://docs.python.org/3.5/tutorial/modules.html

Share Improve this answer Follow

answered Mar 13 '17 at 21:44
kgf3JfUtW
10.2k 7 41 62

18

The reason for

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

is primarily to avoid the import lock problems that would arise from having code directly imported. You want `main()` to run if your file was directly invoked (that's the `__name__ == "__main__"` case), but if your code was imported then the importer has to enter your code from the true main module to avoid import lock problems.

A side-effect is that you automatically sign on to a methodology that supports multiple entry points. You can run your program using `main()` as the entry point, *but you don't have to*. While `setup.py` expects `main()`, other tools use alternate entry points. For example, to run your file as a `gunicorn` process, you define an `app()` function instead of a `main()`. Just as with `setup.py`, `gunicorn` imports your code so you don't want it do do anything while it's being imported (because of the import lock issue).

3 Good to learn about *import lock*. Could you please explain *sign on to a methodology that [...]* part a little bit more? – Wolf Jan 11 '18 at 13:06

1 @Wolf: Sure. I've added a few sentences about the multiple entry points methodology. – personal_cloud Apr 14 '18 at 0:26

16

Every module in python has a attribute called `__name__`. The value of `__name__` attribute is `__main__` when the module is run directly, like `python my_module.py`. Otherwise (like when you say `import my_module`) the value of `__name__` is the name of the module.

Small example to explain in short.

```
#Script test.py

apple = 42

def hello_world():
    print("I am inside hello_world")

if __name__ == "__main__":
    print("Value of __name__ is: ", __name__)
    print("Going to call hello_world")
    hello_world()
```

We can execute this directly as

```
python test.py
```

Output

```
Value of __name__ is: __main__
Going to call hello_world
I am inside hello_world
```

Now suppose we call above script from other script

```
#script external_calling.py

import test
print(test.apple)
test.hello_world()

print(test.__name__)
```

When you execute this

```
python external_calling.py
```

edited Dec 28 '19 at 23:40
charlesreid1
3,318 2 26 43

answered Jun 12 '19 at 9:23
Rishi Bansal
2,719 2 19 36

14

This answer is for Java programmers learning Python. Every Java file typically contains one public class. You can use that class in two ways:

1. Call the class from other files. You just have to import it in the calling program.
2. Run the class stand alone, for testing purposes.

For the latter case, the class should contain a public static void main() method. In Python this purpose is served by the globally defined label `'__main__'`.

Share Improve this answer Follow

edited Oct 18 '18 at 3:09
eyllanesc
202k 15 96 156

answered Oct 7 '18 at 4:52
Raja
874 10 11

10

If this .py file are imported by other .py files, the code under "the if statement" will not be executed.

If this .py are run by `python this_py.py` under shell, or double clicked in Windows. the code under "the if statement" will be executed.

It is usually written for testing.

Share Improve this answer Follow

answered Jun 19 '18 at 11:44
pah8J
717 7 15

10

If the python interpreter is running a particular module then `__name__` global variable will have value `"__main__"`

```
def a():
    print("a")
def b():
    print("b")

if __name__ == "__main__":

        print ("you can see me" )
        a()
else:

        print ("You can't see me")
        b()
```

When you run this script prints **you can see me**

a

If you import this file say A to file B and execute the file B then `if __name__ == "__main__"` in file A becomes false, so it prints **You can't see me**

b

Share Improve this answer Follow

answered Jul 30 '19 at 16:22

Nikil Munireddy
168 1 5

if name == 'main':

9

We see if `__name__ == '__main__':` quite often.

It checks if a module is being imported or not.

In other words, the code within the `if` block will be executed only when the code runs directly. Here `directly` means `not imported`.

Let's see what it does using a simple code that prints the name of the module:

```
# test.py
def test():
   print('test module name=%s' %(__name__))

if __name__ == '__main__':
   print('call test()')
   test()
```

If we run the code directly via `python test.py`, the module name is `__main__`:

```
call test()
test module name=__main__
```

Share Improve this answer Follow

edited May 23 '18 at 22:36
Peter Mortensen
28.7k 21 95 123

answered Apr 4 '18 at 14:32
Ali Hallaji
1,848 1 18 28

9

In simple words:

The code you see under `if __name__ == "__main__":` will only get called upon when your python file is executed as "python example1.py".

However, if you wish to import your python file 'example1.py' as a module to work with another python file say 'example2.py', the code under `if __name__ == "__main__":` will not run or take any effect.

Share Improve this answer Follow

answered Oct 22 '20 at 18:01
Mustapha Babatunde
393 4 7

7

All the answers have pretty much explained the functionality. But I will provide one example of its usage which might help clearing out the concept further.

Assume that you have two Python files, a.py and b.py. Now, a.py imports b.py. We run the a.py file, where the "import b.py" code is executed first. Before the rest of the a.py code runs, the code in the file b.py must run completely.

In the b.py code there is some code that is exclusive to that file b.py and we don't want any other file (other than b.py file), that has imported the b.py file, to run it.

So that is what this line of code checks. If it is the main file (i.e., b.py) running the code, which in this case it is not (a.py is the main file running), then only the code gets executed.

Share Improve this answer Follow

edited May 23 '18 at 22:38
Peter Mortensen
28.7k 21 95 123

answered May 4 '18 at 8:25
preetika mondal
171 1 8

5

Create a file, **a.py**:

```
print(__name__) # It will print out __main__
```

`__name__` is always equal to `__main__` whenever that file is **run directly** showing that this is the main file.

Create another file, **b.py**, in the same directory:

```
import a  # Prints a
```

Run it. It will print **a**, i.e., the name of the file which **is imported**.

So, to show **two different behavior of the same file**, this is a commonly used trick:

```
# Code to be run when imported into another python file

if __name__ == '__main__':
    # Code to be run only when run directly
```

Share Improve this answer Follow

edited May 23 '18 at 22:34

answered Jan 8 '18 at 15:24