Theory: Create module

0 / 0 problems solved

Start practicing

0 users solved this topic.

Current topic:

Create module

§1. Module design

Basically, a module is just a file that has a **.py** extension and contains statements and definitions. What is the point? Well, modules help you organize and reuse code. Once you wrote a module you can load it from the interpreter or another module.

A simple module that is written for direct execution is often called a **script**. The difference between a module and a script in Python is only in the way they are used. Modules are loaded from other modules or scripts and scripts are executed directly.

Let's take a look at the example of a simple script below:

```
# hello.py script

print("Hello, World!")
```

You have already seen this example but now we want to turn it into a script. What you need to do is simple: you just save this code in a file named `hello.py` and then run it with Python. To run a script use `python <script>`, where `<script>` is the path to your Python file.

```
~$ python hello.py
Hello, World!
```

Congratulations! This is your first script in Python.

§2. Module importing

A module can be loaded from another module. That allows you to write a piece of code once and then use it wherever you want. It is really helpful when you work on larger projects and want to separate concerns between different modules. We already saw examples of an imported module from another module in the previous topic.

When working in the interactive mode of the interpreter you can load modules as well. Pay attention, that the module should be placed in the directory from which you run Python. For example, you can load `hello.py` file we discussed in the previous section from the interpreter like this:

```
~$ python
Python 3.6.6 (default, Sep 12 2018, 18:26:19)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> import hello
Hello, World!
```

§3. Common mistakes

Now it is time to cover some common mistakes you can make when defining or importing modules.

If you accidentally import a module from itself, the code of the module will be executed twice and that is generally not something you want to happen.

```
# itself.py

import itself

print("Hello, it's me!")
```

The output looks like this:

```
Hello, it's me!
Hello, it's me!
```

So be careful and avoid situations when you import a module from itself.

Another common mistake is **name shadowing**. For example, you have created a local module that has the same name as some built-in module. In this case, you won't be able to import anything from the original module, because the import system will search names in your custom module.

Imagine, you created a module `socket.py` and then you try to import some function from the standard Python `socket` module within your module.

```
# socket.py

from socket import socket

print("All cool!")
```

You'll see an error message that says that Python cannot import `socket` from `socket` module:

```
...

ImportError: cannot import name 'socket'
```

One way to avoid this is not to name your files the same as the built-in modules you might use. Just suffix `_script` to the name of your scripts and modules and you will be safe from this name shadowing problem.

Whenever the module is imported it is fully executed and then added to your current namespace. Even special forms of import statement such as `from module import something` don't affect that fact. This may become a problem in situations when you want to be able to both import your module and execute it as a script.

Consider the example:

```
# unsafe_module.py

name = "George"

print("Hello,", name)
```

If you define another script and import `name` from `unsafe_module` you'll see *Hello, George* printed.

```
# unsafe_bye.py script

from unsafe_module import name

print("Bye,", name)
```

The output:

```
Hello, George
Bye, George
```

To solve this issue you can simply divide your file into two: one containing only definitions, another containing the code that imports definitions from the first file and uses them. But it's also common to use the **__main__ pattern**.

§4. __main__ pattern

Let's learn another option of how to make your script safe to import. We will change the `unsafe_module.py` file from the previous section.

Table of contents:

↑ Create module
§1. Module design
§2. Module importing
§3. Common mistakes
§4. __main__ pattern
Feedback & Comments

```
# safe_module.py

name = "George"

if __name__ == "__main__":
    print("Hello,", name)
```

The name of the module is always available in the built-in variable `__name__`. When you are executing a script `__name__` has a value `"__main__"`. So here we check the value of `__name__` and print the line only if the module is executed as a script.

```
# safe_bye.py script

from safe_module import name

print("Bye,", name)
```

The output is the following:

```
Bye, George
```

In general, if you have more than just one line to execute in a script it's convenient to move all that code into a function called main and then call it like that:

```
# safe_main_module.py

name = "George"

def main():
    print("Hello,", name)

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

Note, that the naming itself doesn't affect the way a function is executed, it's just a convention to indicate that this function is run only when the file is used like a script.

Report a typo

Start practicing

Comments (12) Hints (1) Useful links (1) Show discussion