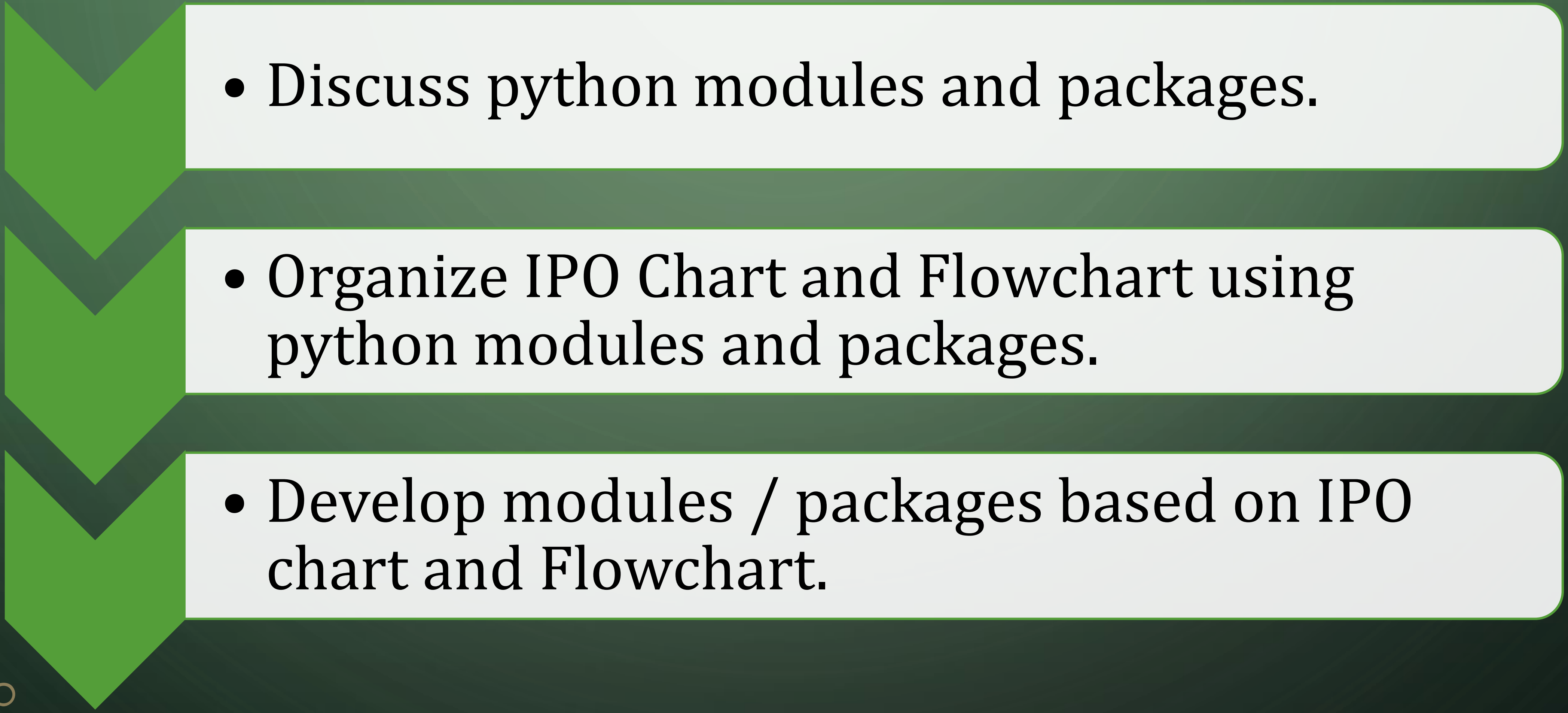
MODULES AND PACKAGES

CU10 : WEEK 13

OBJECTIVES:



- Discuss python modules and packages.
- Organize IPO Chart and Flowchart using python modules and packages.
- Develop modules / packages based on IPO chart and Flowchart.

What is Modular Programming?

MODULAR PROGRAMMING

- process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules. Individual modules can then be cobbled together like building blocks to create a larger application.

Advantages to Modularizing Code

Simplicity

❿Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem.

Maintainability

❿If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program.

Advantages to Modularizing Code

Reusability

❿Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to duplicate code.

Scoping

❿Modules typically define a separate namespace, which helps avoid collisions between identifiers in different areas of a program.

Python Modules

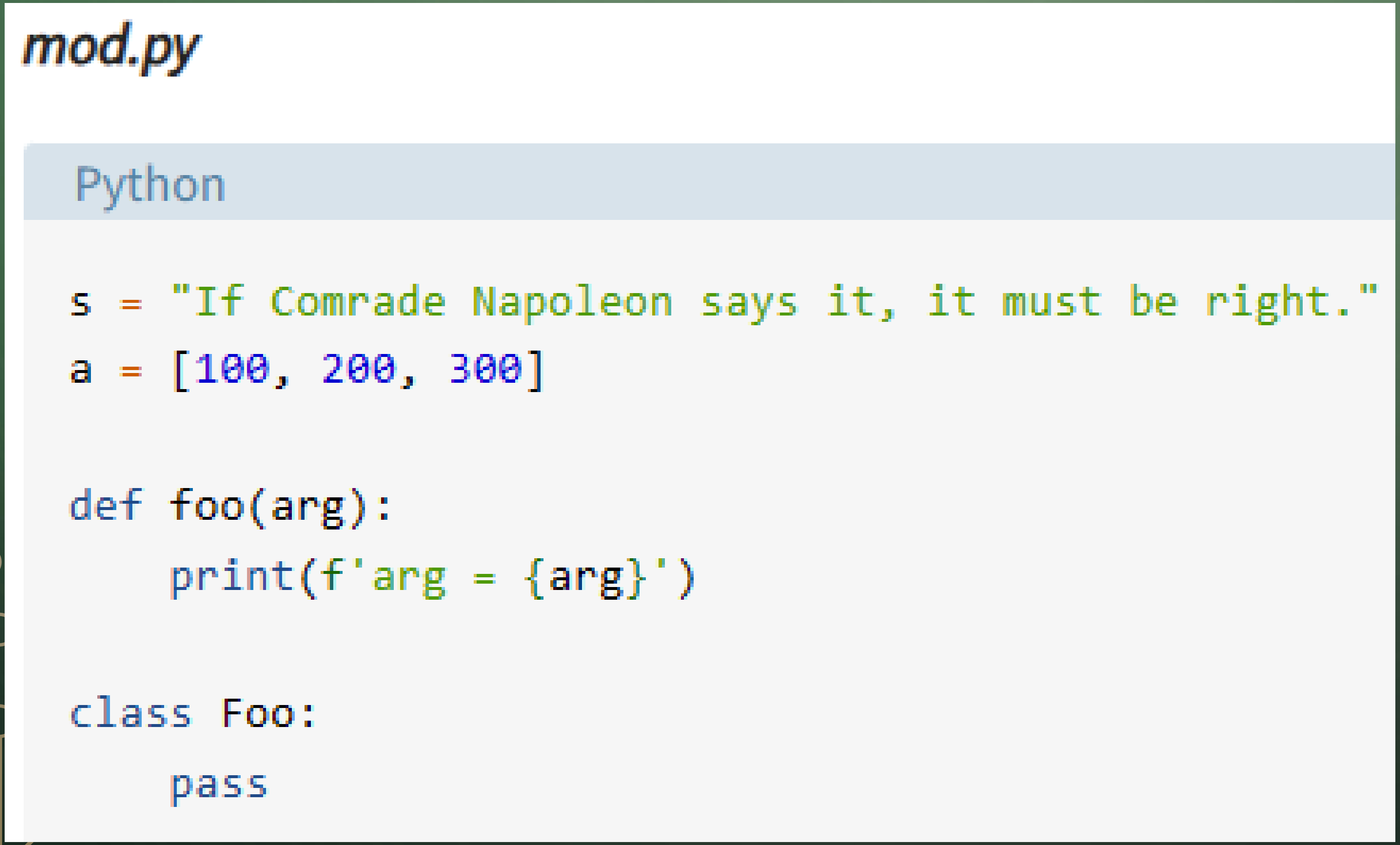
PYTHON MODULES

- a file containing a set of functions you want to include in your application.
- a piece of software that has a specific functionality.
- Modules in Python are just Python files with a .py extension.
- The name of the module is the same as the file name.
- A Python module can have a set of functions, classes, or variables defined and implemented.

PYTHON MODULES

- A module can be written in Python itself.
- A module can be written in C and loaded dynamically at run-time, like the re (regular expression) module.
- A built-in module is intrinsically contained in the interpreter.

PYTHON MODULES



mod.py

Python

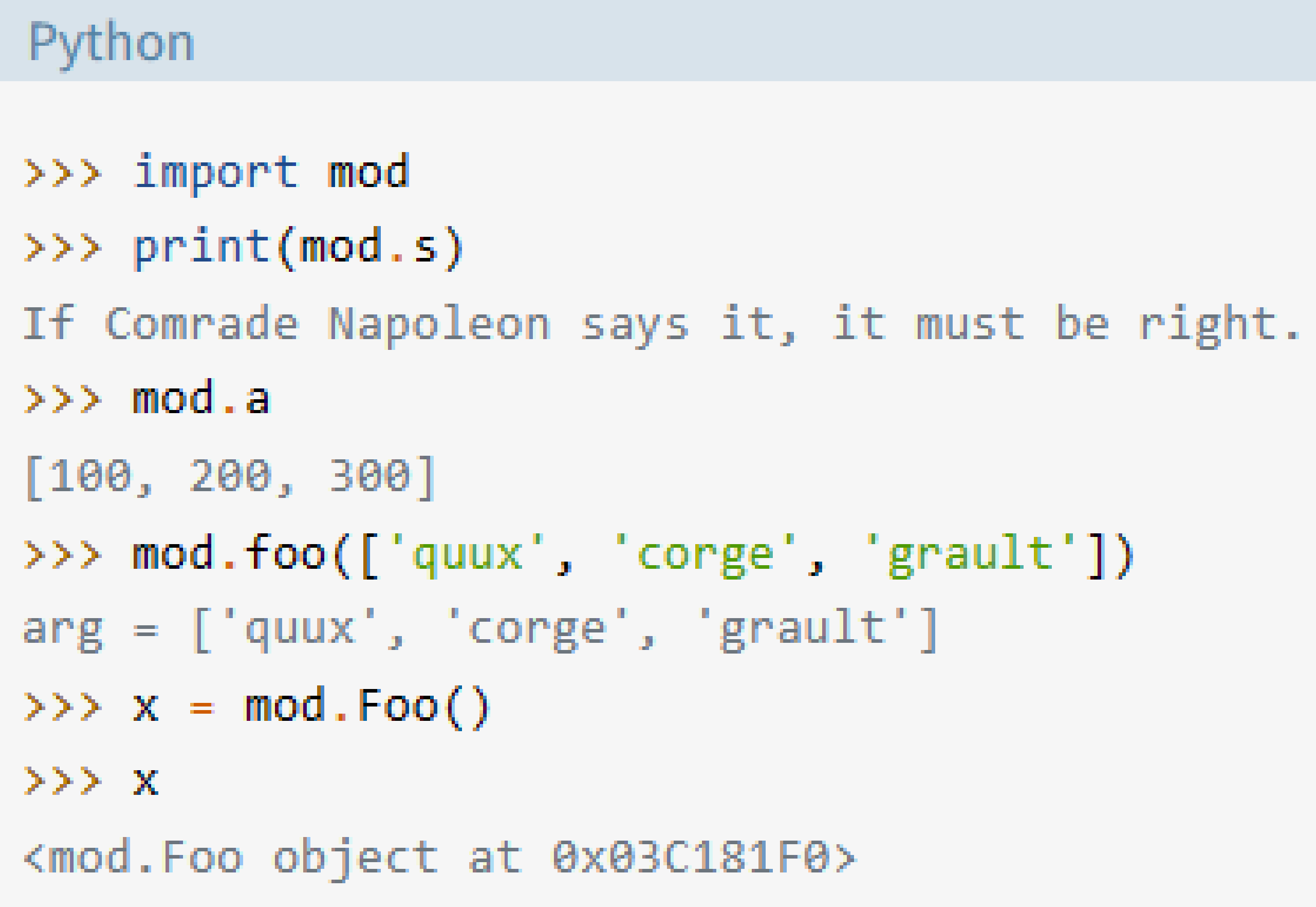
```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

Several objects are defined in mod.py:

- s (a string)
- a (a list)
- foo() (a function)
- Foo (a class)

PYTHON MODULES

these objects can be accessed by importing the module as follows:

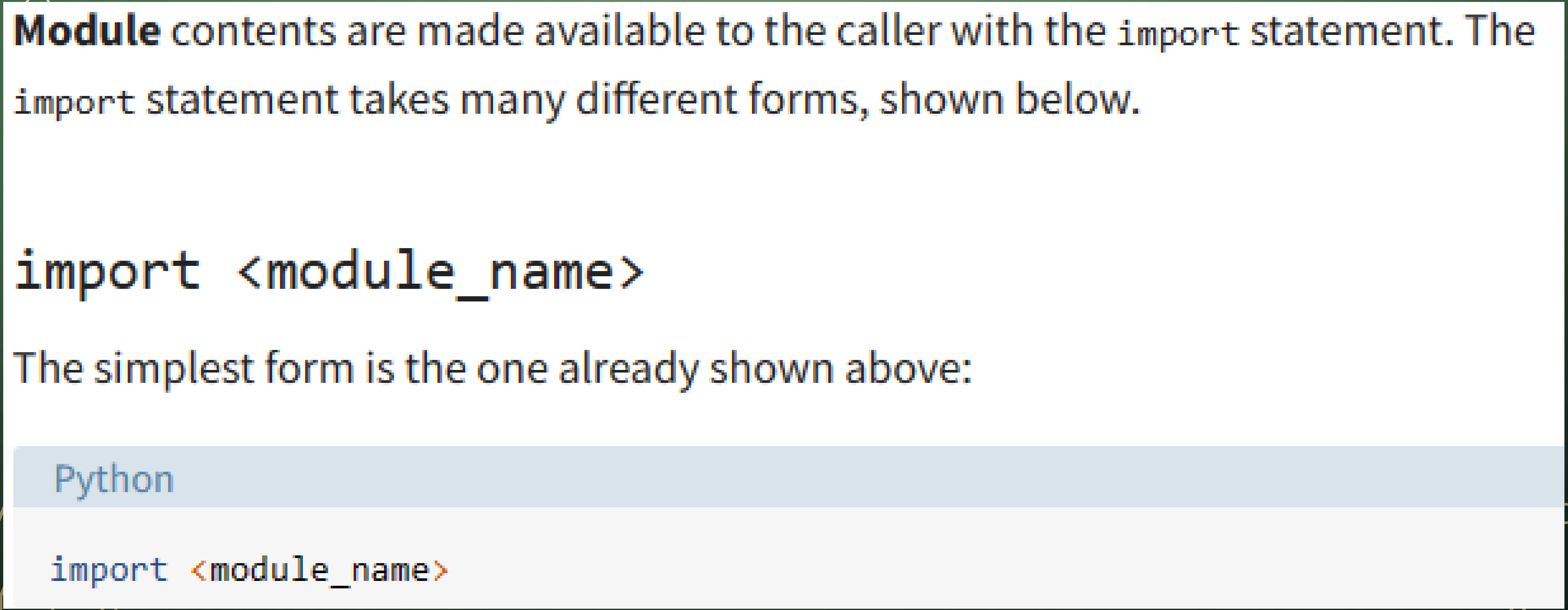


Python

```
>>> import mod
>>> print(mod.s)
If Comrade Napoleon says it, it must be right.
>>> mod.a
[100, 200, 300]
>>> mod.foo(['quux', 'corge', 'grault'])
arg = ['quux', 'corge', 'grault']
>>> x = mod.Foo()
>>> x
<mod.Foo object at 0x03C181F0>
```

THE import STATEMENT

THE import STATEMENT



Module contents are made available to the caller with the `import` statement. The `import` statement takes many different forms, shown below.

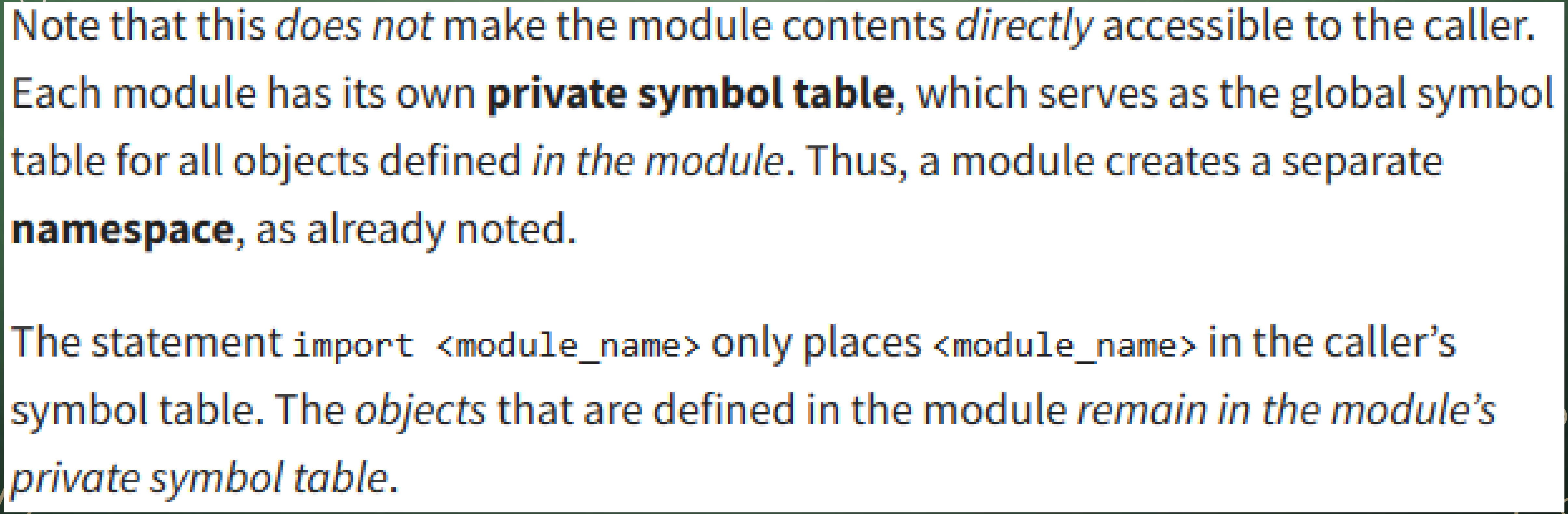
import <module_name>

The simplest form is the one already shown above:

Python

```python
import <module_name>
```

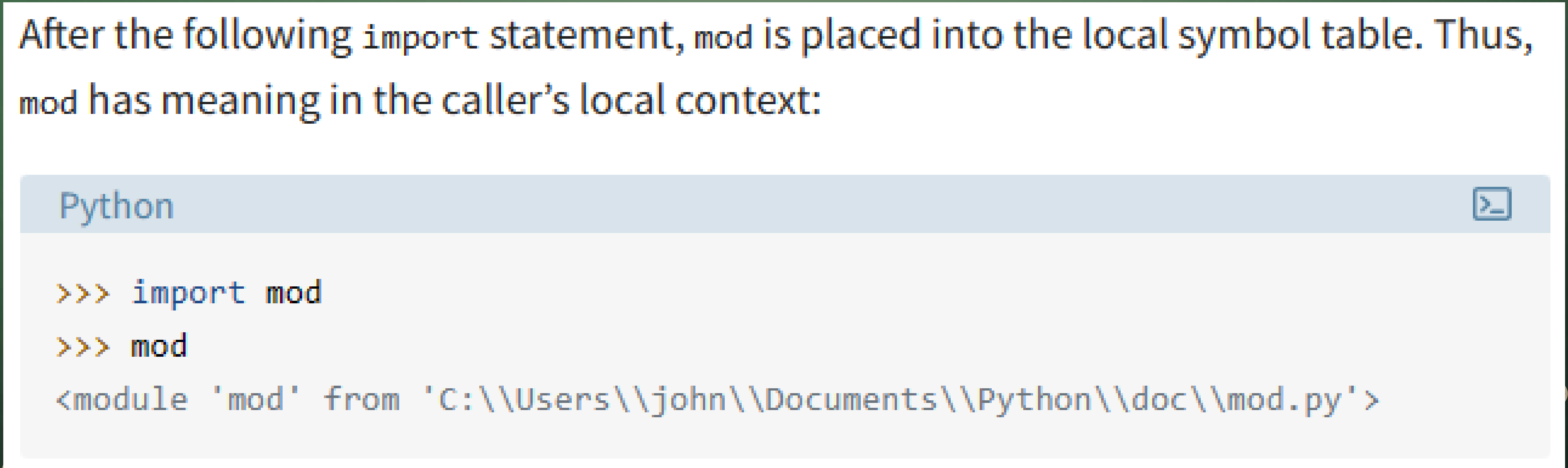
THE import STATEMENT



Note that this *does not* make the module contents *directly* accessible to the caller. Each module has its own **private symbol table**, which serves as the global symbol table for all objects defined *in the module*. Thus, a module creates a separate **namespace**, as already noted.

The statement `import <module_name>` only places `<module_name>` in the caller's symbol table. The *objects* that are defined in the module *remain in the module's private symbol table*.

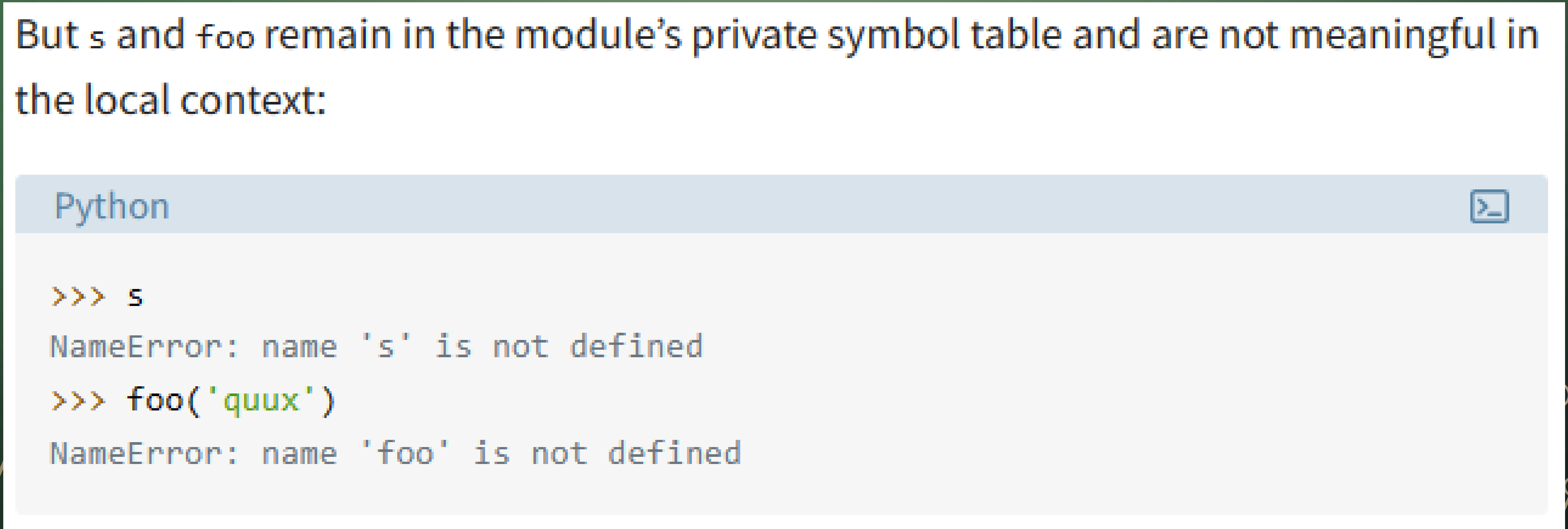
THE import STATEMENT



After the following `import` statement, `mod` is placed into the local symbol table. Thus, `mod` has meaning in the caller’s local context:

```
>>> import mod
>>> mod
<module 'mod' from 'C:\\Users\\john\\Documents\\Python\\doc\\mod.py'>
```

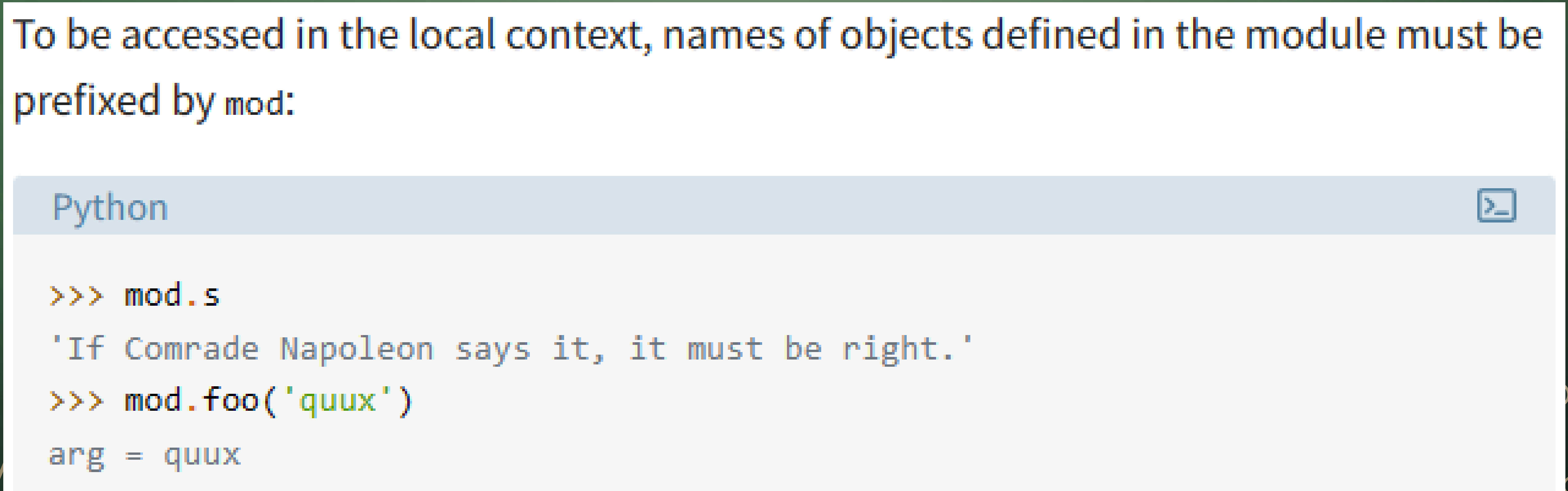
THE import STATEMENT



But s and foo remain in the module’s private symbol table and are not meaningful in the local context:

```
>>> s
NameError: name 's' is not defined
>>> foo('quux')
NameError: name 'foo' is not defined
```

THE import STATEMENT

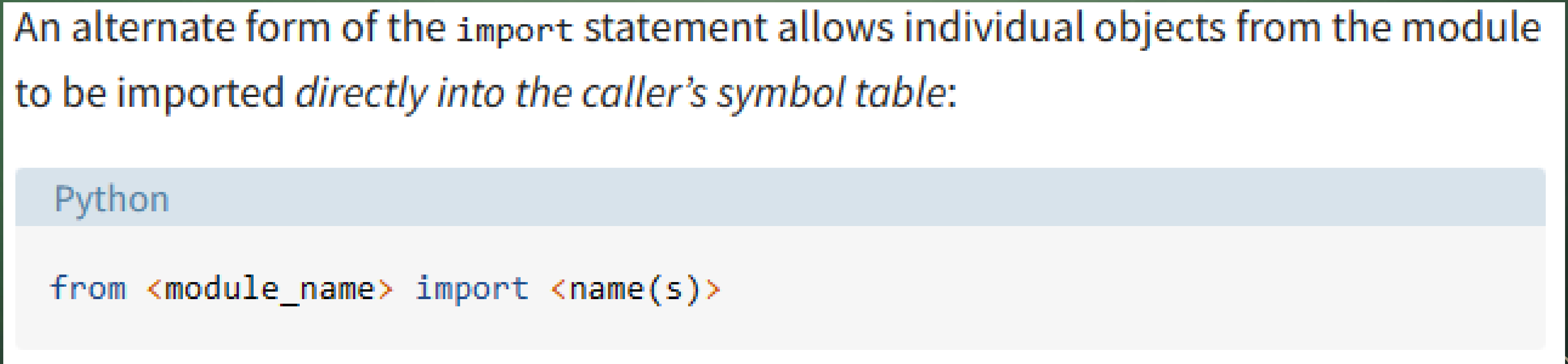


To be accessed in the local context, names of objects defined in the module must be prefixed by `mod`:

```
>>> mod.s
'If Comrade Napoleon says it, it must be right.'
>>> mod.foo('quux')
arg = quux
```

from
<module_name>
import <name(s)>

from <module_name> import <name(s)>

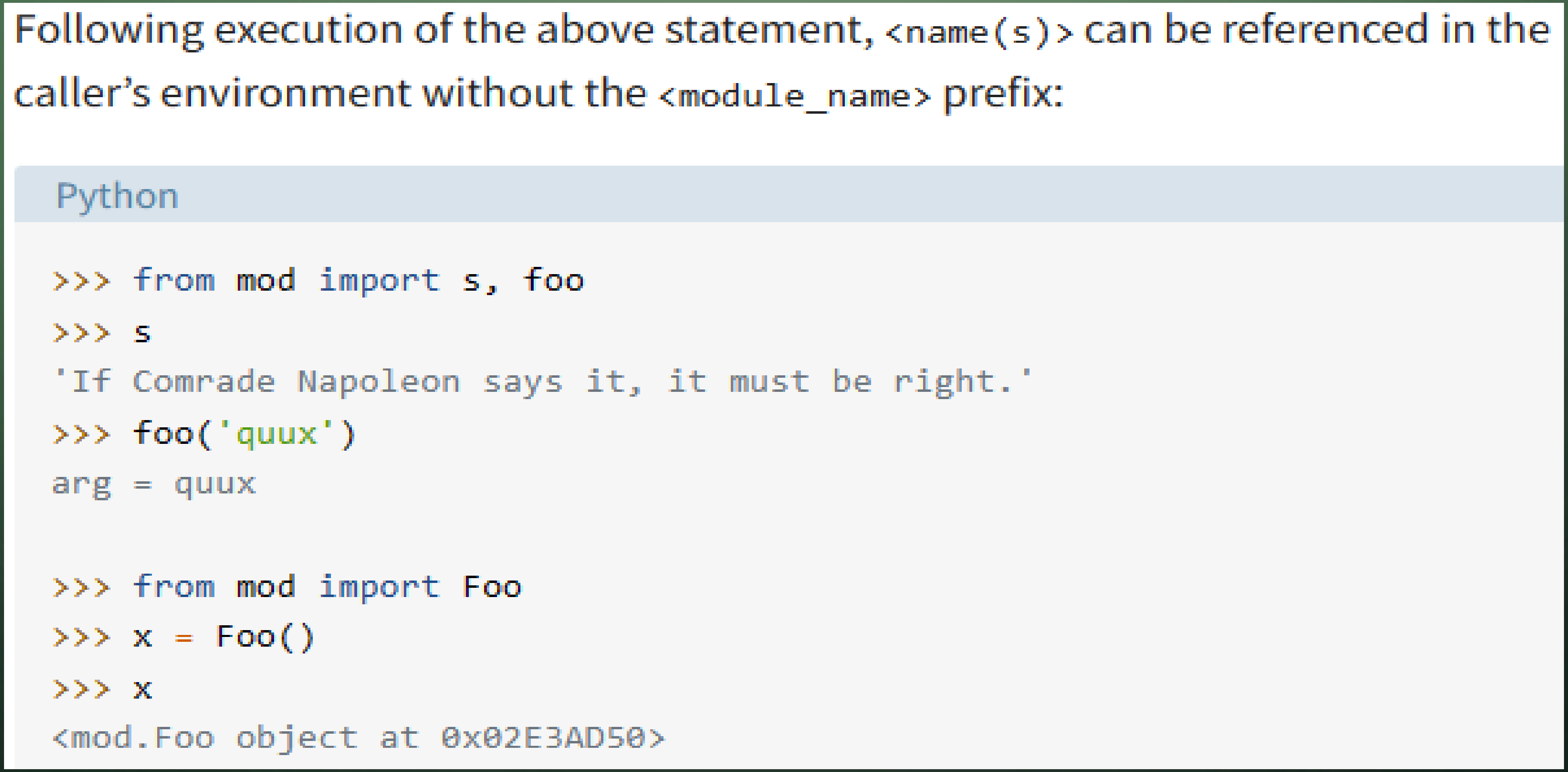


An alternate form of the `import` statement allows individual objects from the module to be imported *directly into the caller's symbol table*:

Python

```
from <module_name> import <name(s)>
```

from <module_name> import <name(s)>

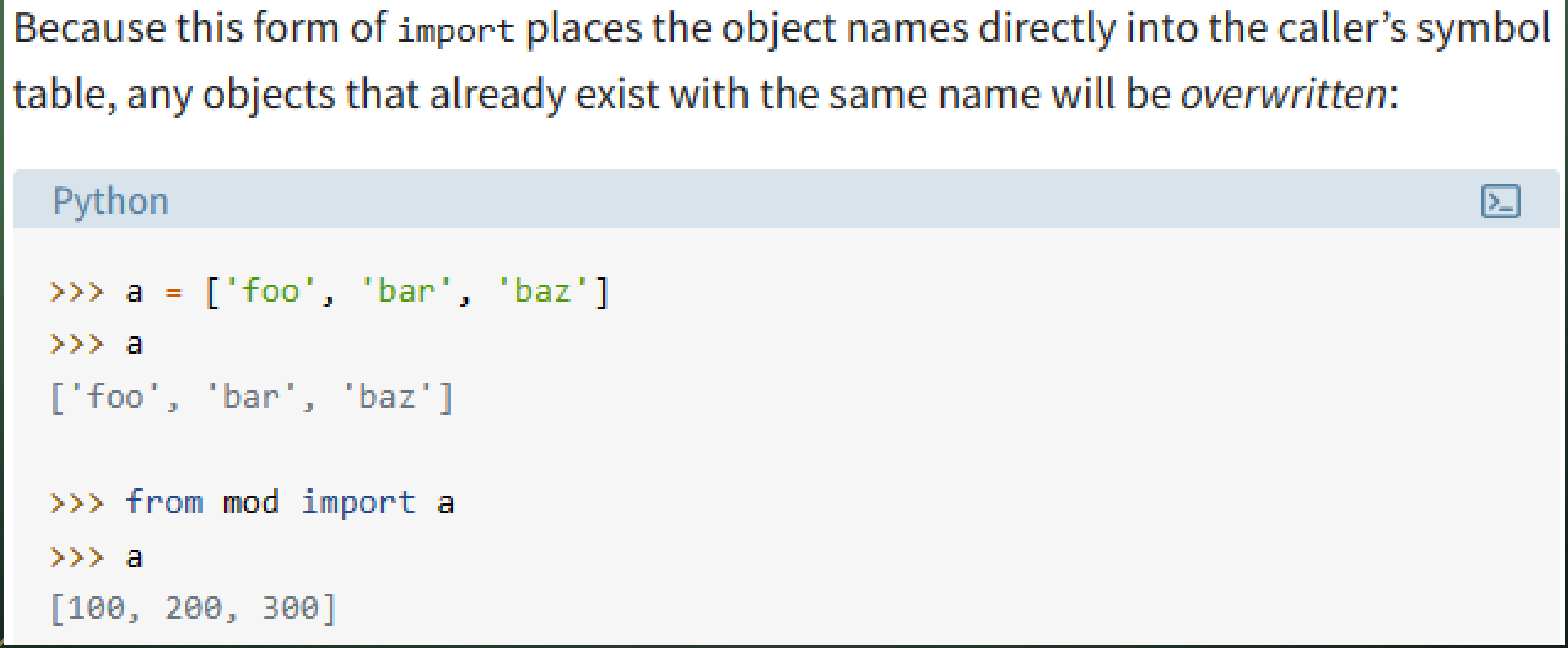


Following execution of the above statement, `<name(s)>` can be referenced in the caller's environment without the `<module_name>` prefix:

Python

```
>>> from mod import s, foo
>>> s
'If Comrade Napoleon says it, it must be right.'
>>> foo('quux')
arg = quux

>>> from mod import Foo
>>> x = Foo()
>>> x
<mod.Foo object at 0x02E3AD50>
```

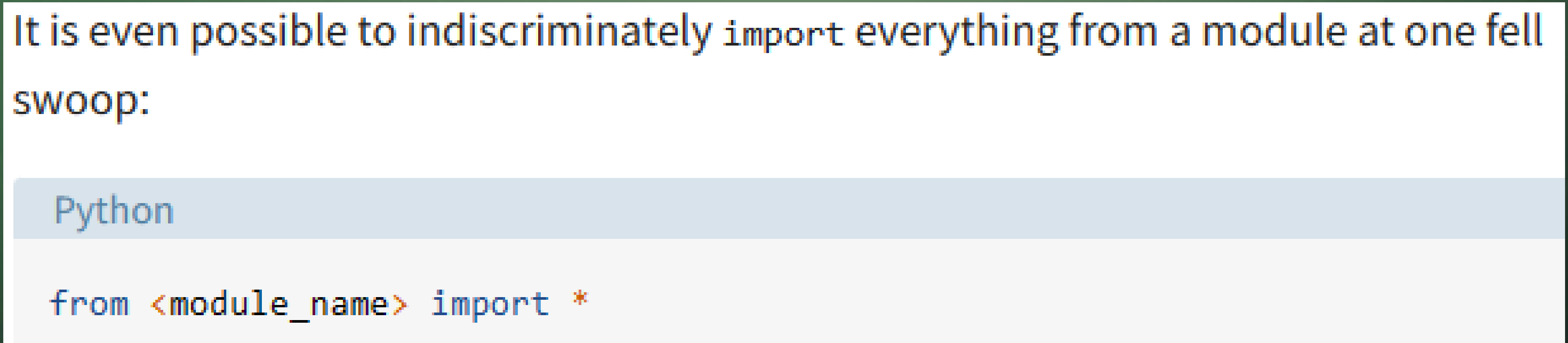
from <module_name> import <name(s)>



Because this form of `import` places the object names directly into the caller’s symbol table, any objects that already exist with the same name will be *overwritten*:

```
>>> a = ['foo', 'bar', 'baz']
>>> a
['foo', 'bar', 'baz']

>>> from mod import a
>>> a
[100, 200, 300]
```

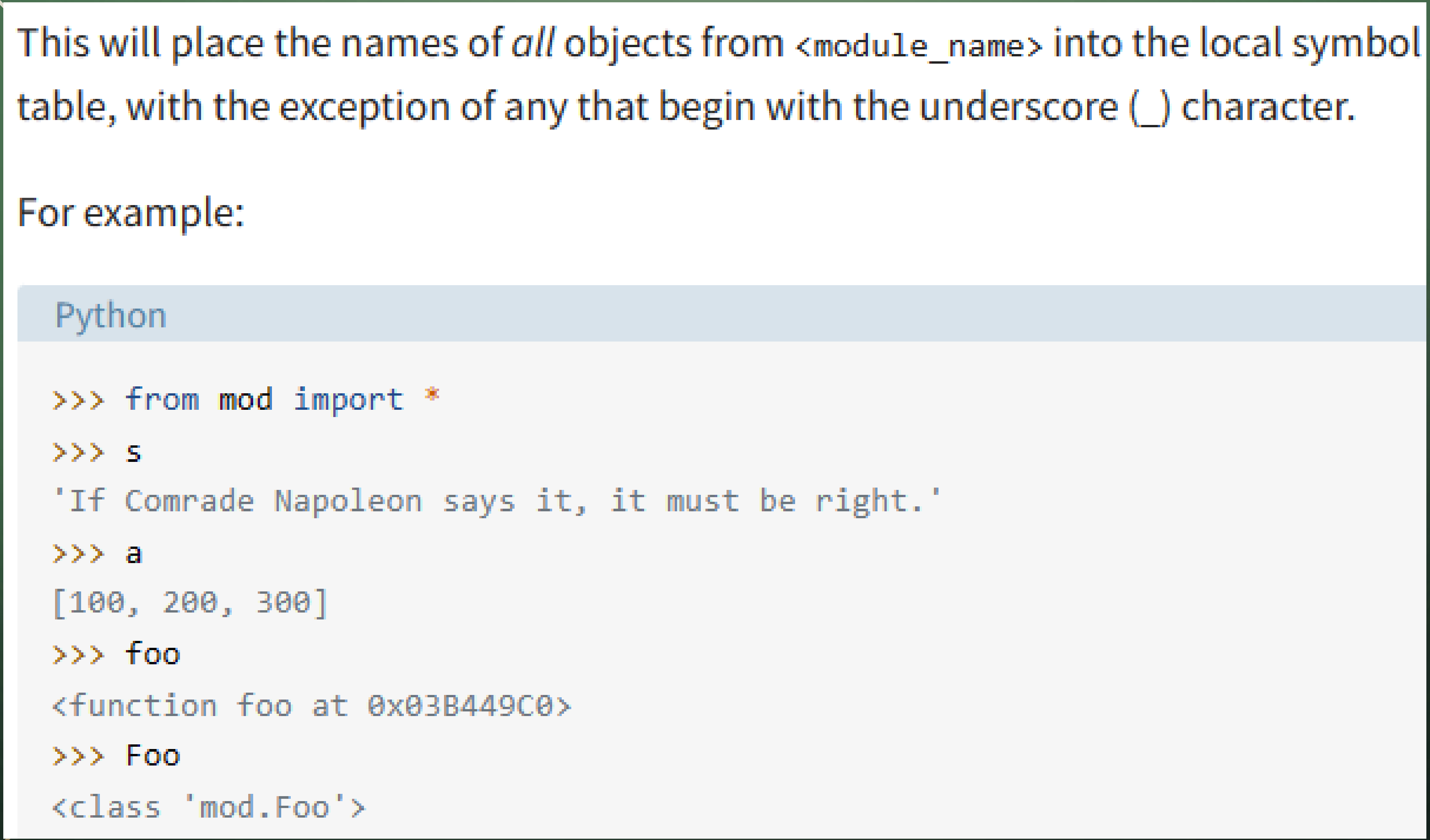
from <module_name> import <name(s)>



It is even possible to indiscriminately `import` everything from a module at one fell swoop:

Python

```python
from <module_name> import *
```



This will place the names of *all* objects from `<module_name>` into the local symbol table, with the exception of any that begin with the underscore (_) character.

For example:

Python

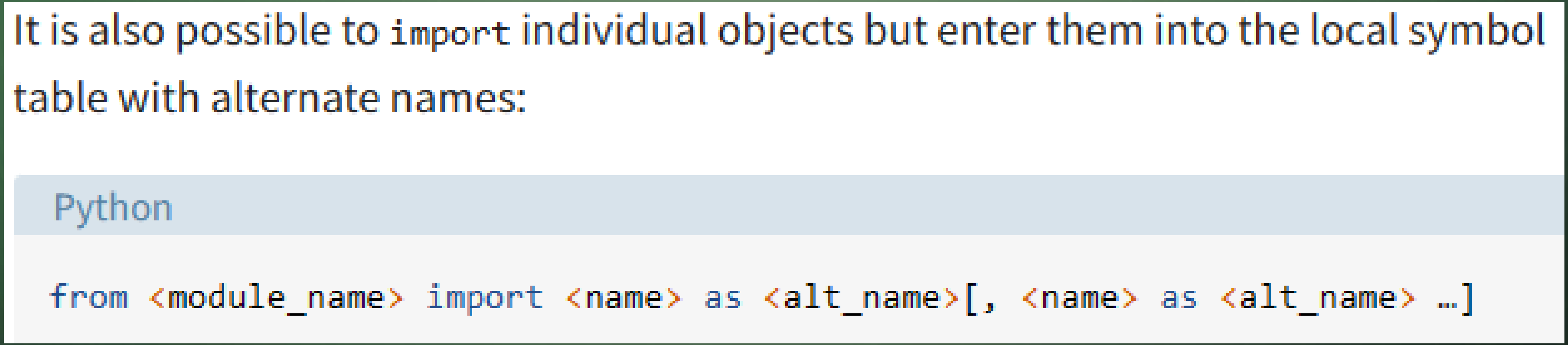
```
>>> from mod import *
>>> s
'If Comrade Napoleon says it, it must be right.'
>>> a
[100, 200, 300]
>>> foo
<function foo at 0x03B449C0>
>>> Foo
<class 'mod.Foo'>
```

from <module_name> import <name(s)>

- This isn’t necessarily recommended in large-scale production code. It’s a bit dangerous because you are entering names into the local symbol table **en masse**. Unless you know them all well and can be confident there won’t be a conflict, you have a decent chance of overwriting an existing name inadvertently. However, this syntax is quite handy when you are just mucking around with the interactive interpreter, for testing or discovery purposes, because it quickly gives you access to everything a module has to offer without a lot of typing.

from <module_name> import <name> as <alt_name>

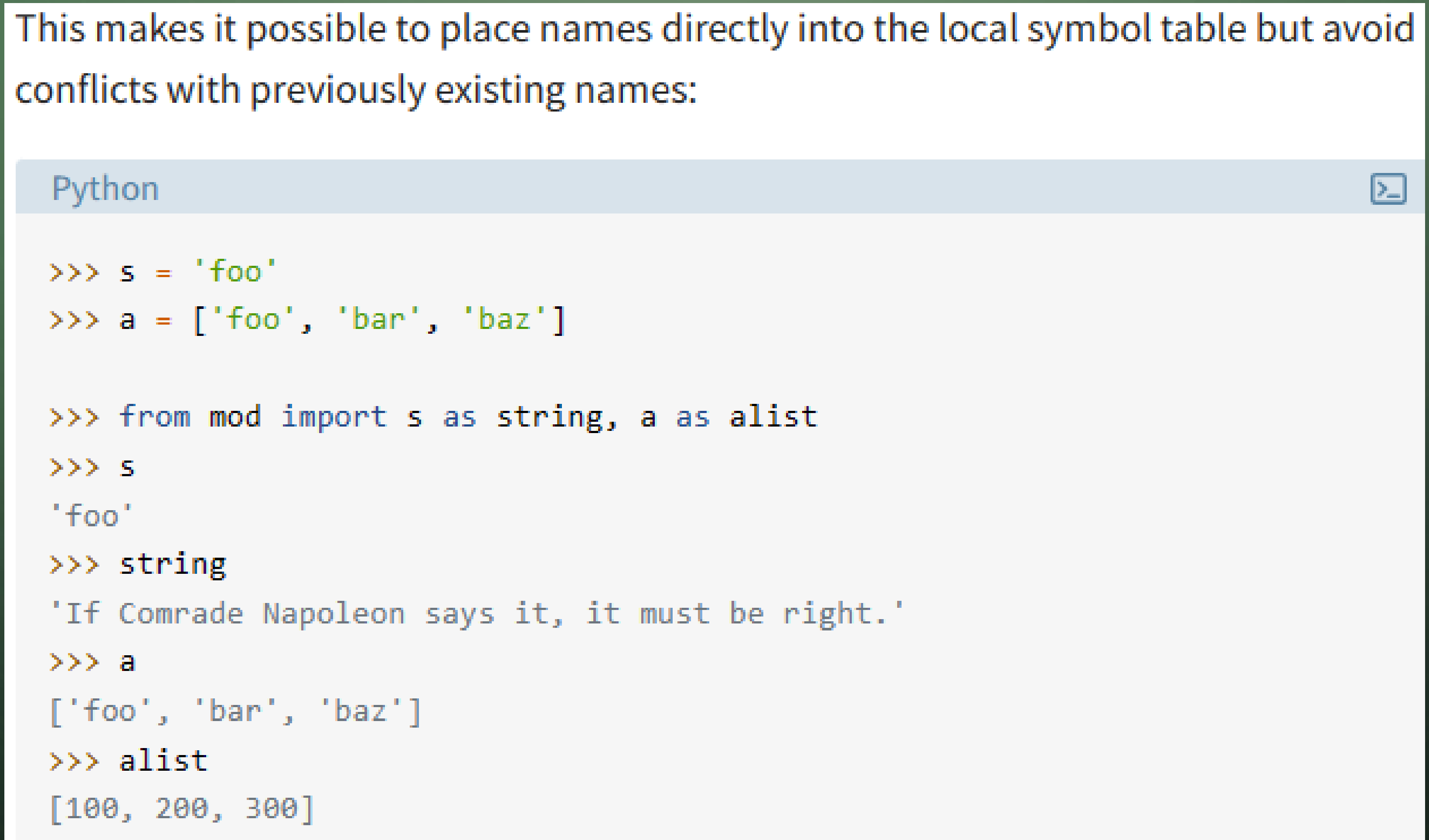
from <module_name> import <name> as <alt_name>



It is also possible to `import` individual objects but enter them into the local symbol table with alternate names:

```
from <module_name> import <name> as <alt_name>[, <name> as <alt_name> …]
```

from <module_name> import <name> as <alt_name>

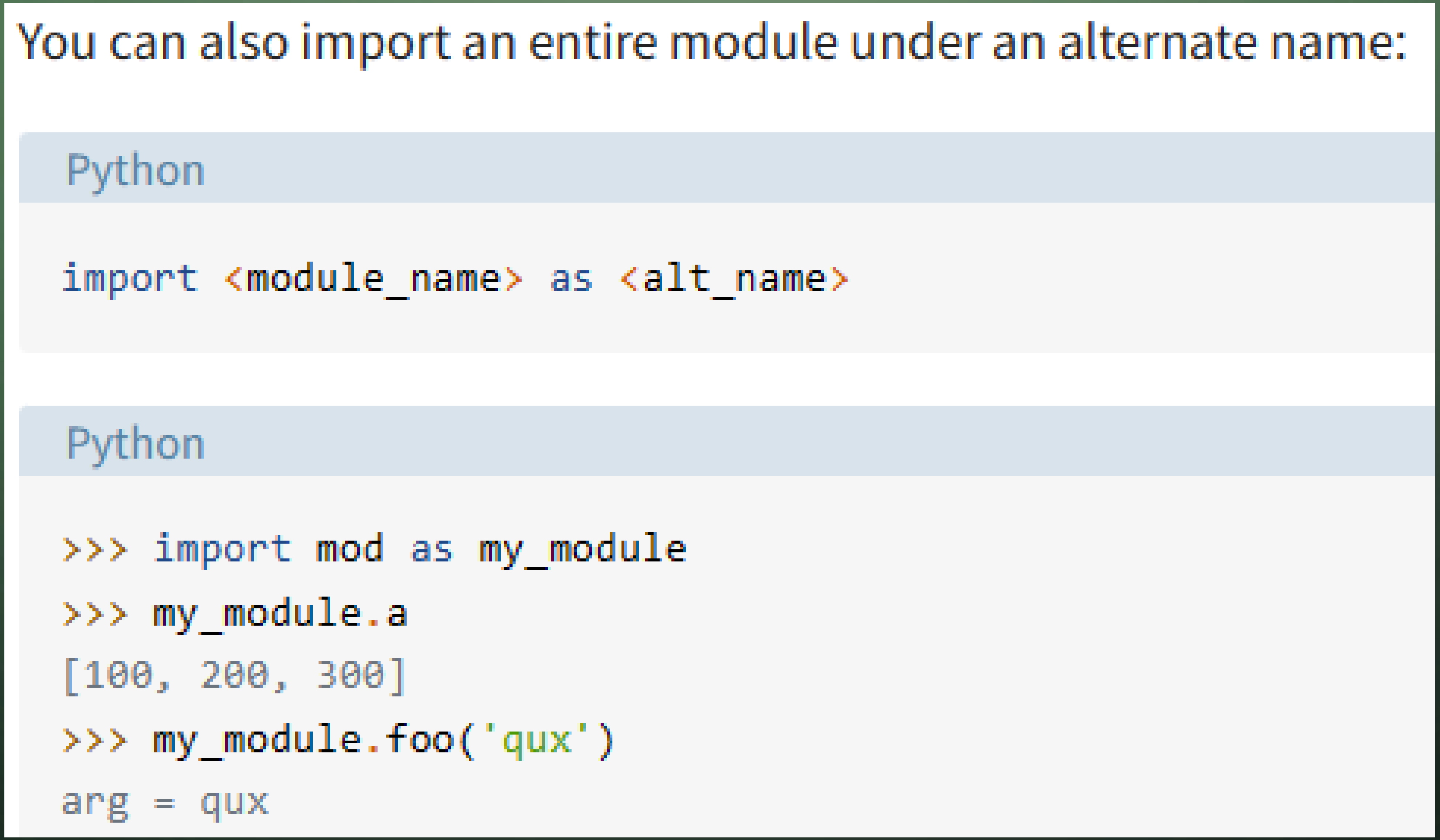


This makes it possible to place names directly into the local symbol table but avoid conflicts with previously existing names:

```
>>> s = 'foo'
>>> a = ['foo', 'bar', 'baz']

>>> from mod import s as string, a as alist
>>> s
'foo'
>>> string
'If Comrade Napoleon says it, it must be right.'
>>> a
['foo', 'bar', 'baz']
>>> alist
[100, 200, 300]
```

```
import <module_name> as <alt_name>
```

import <module_name> as <alt_name>

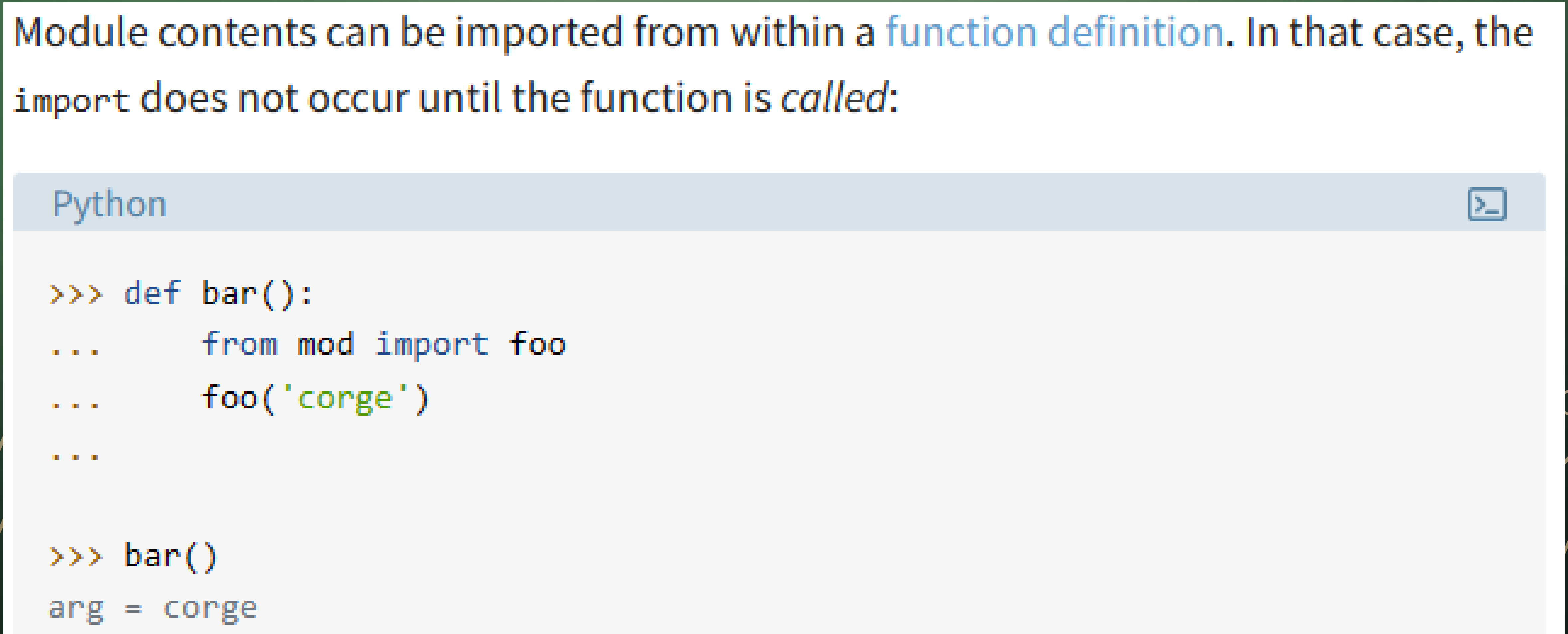


You can also import an entire module under an alternate name:

```
import <module_name> as <alt_name>
```

```
>>> import mod as my_module
>>> my_module.a
[100, 200, 300]
>>> my_module.foo('qux')
arg = qux
```

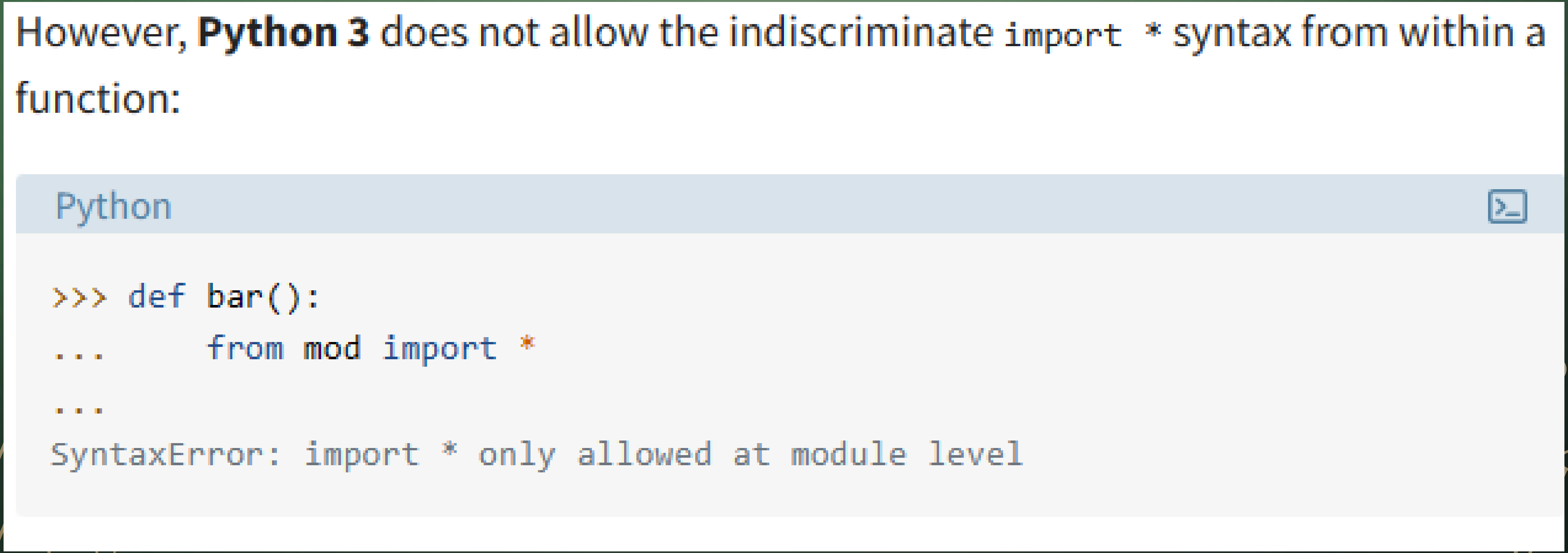
import <module_name> as <alt_name>



Module contents can be imported from within a function definition. In that case, the `import` does not occur until the function is *called*:

```
>>> def bar():
...     from mod import foo
...     foo('corge')
...

>>> bar()
arg = corge
```

import <module_name> as <alt_name>

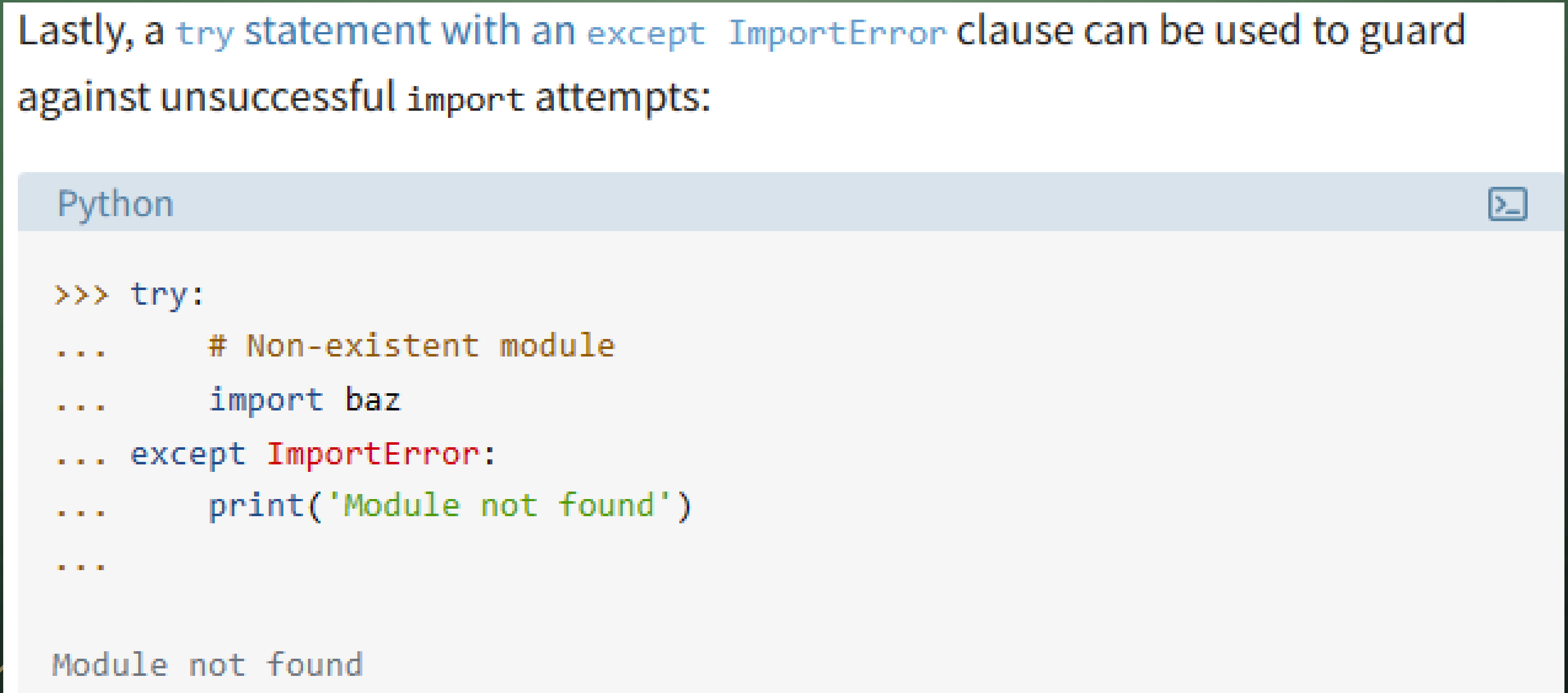


However, **Python 3** does not allow the indiscriminate `import *` syntax from within a function:

Python

```
>>> def bar():
...     from mod import *
...
SyntaxError: import * only allowed at module level
```

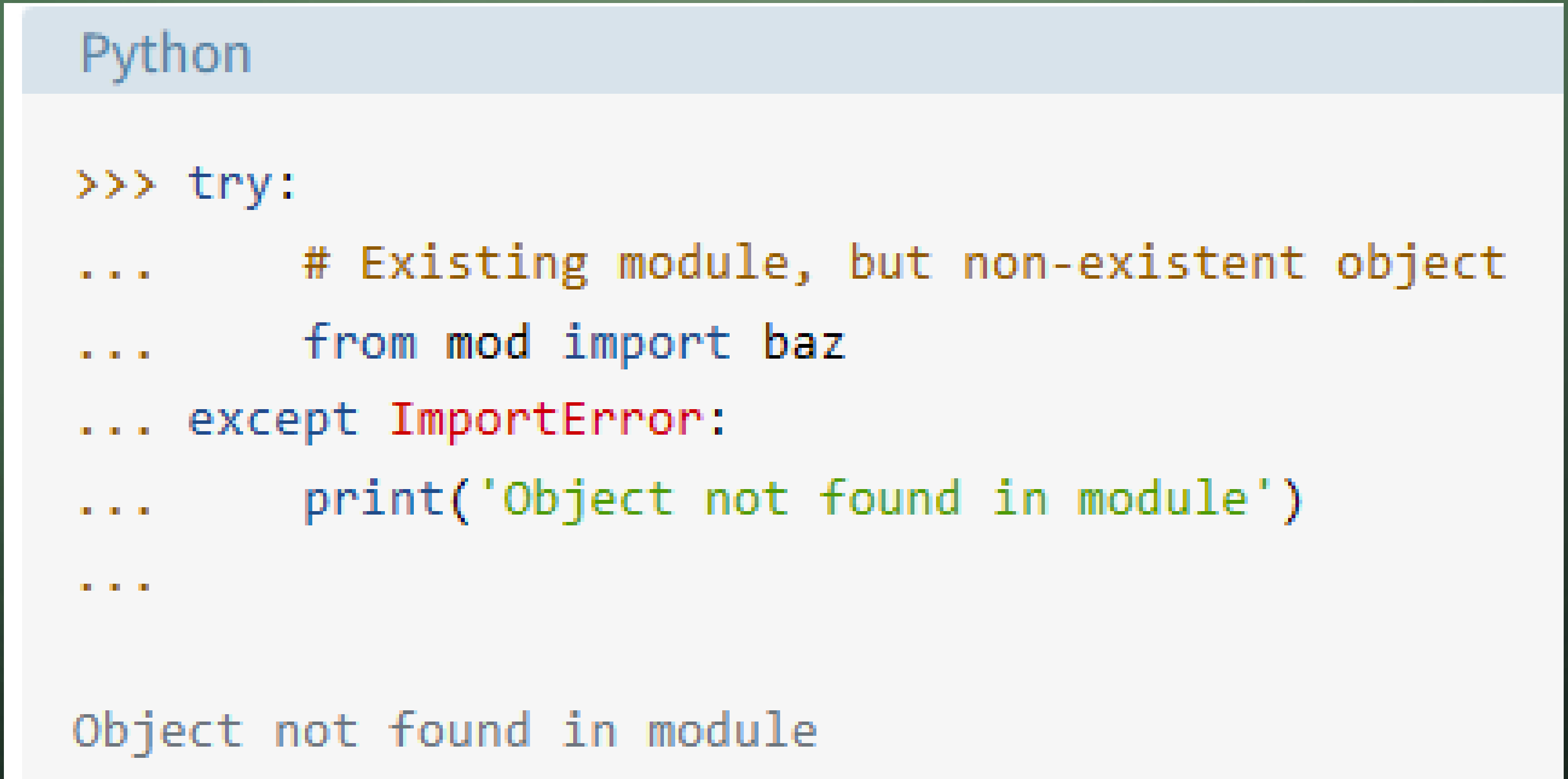
import <module_name> as <alt_name>



Lastly, a try statement with an except ImportError clause can be used to guard against unsuccessful import attempts:

```
>>> try:
...     # Non-existent module
...     import baz
... except ImportError:
...     print('Module not found')
...

Module not found
```

import <module_name> as <alt_name>



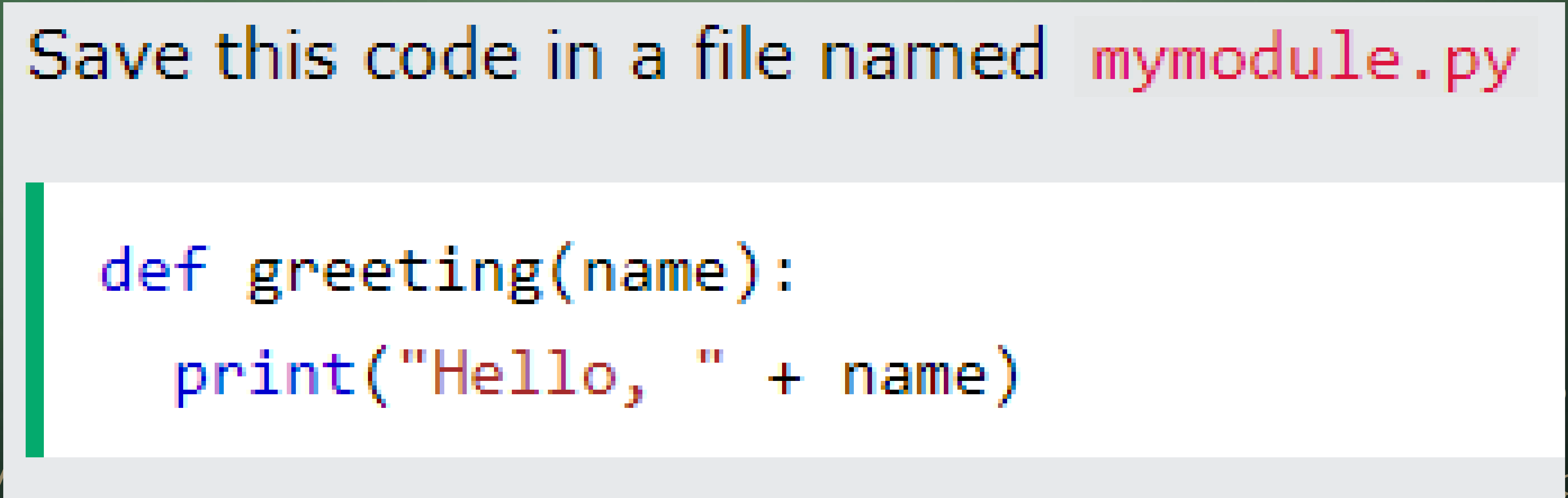
Python

```
>>> try:
...     # Existing module, but non-existent object
...     from mod import baz
... except ImportError:
...     print('Object not found in module')
...

Object not found in module
```

Any Questions?

Create a Module

- To create a module just save the code you want in a file with the file extension .py:

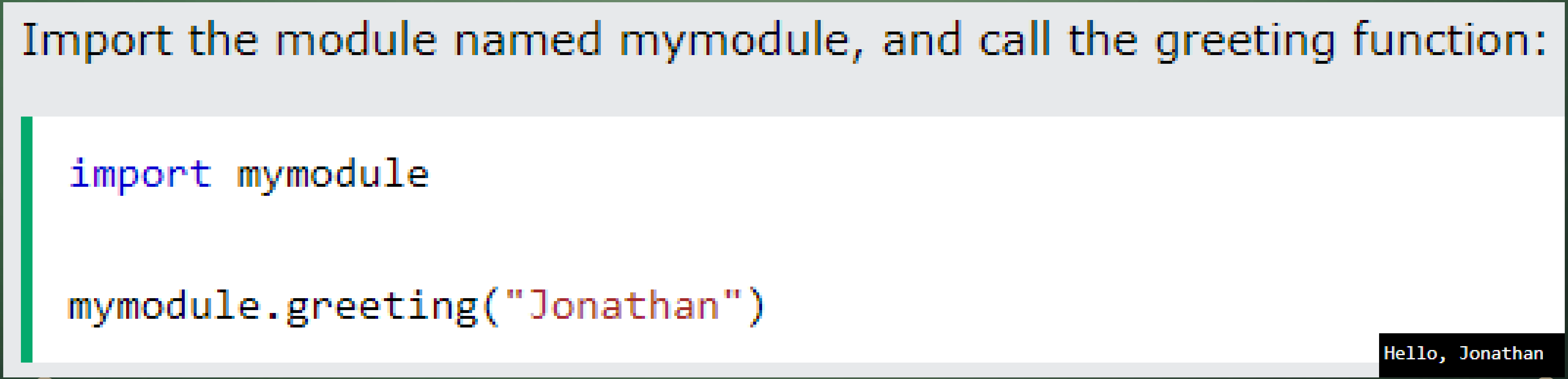


Save this code in a file named `mymodule.py`

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module

- Now we can use the module we just created, by using the import statement:



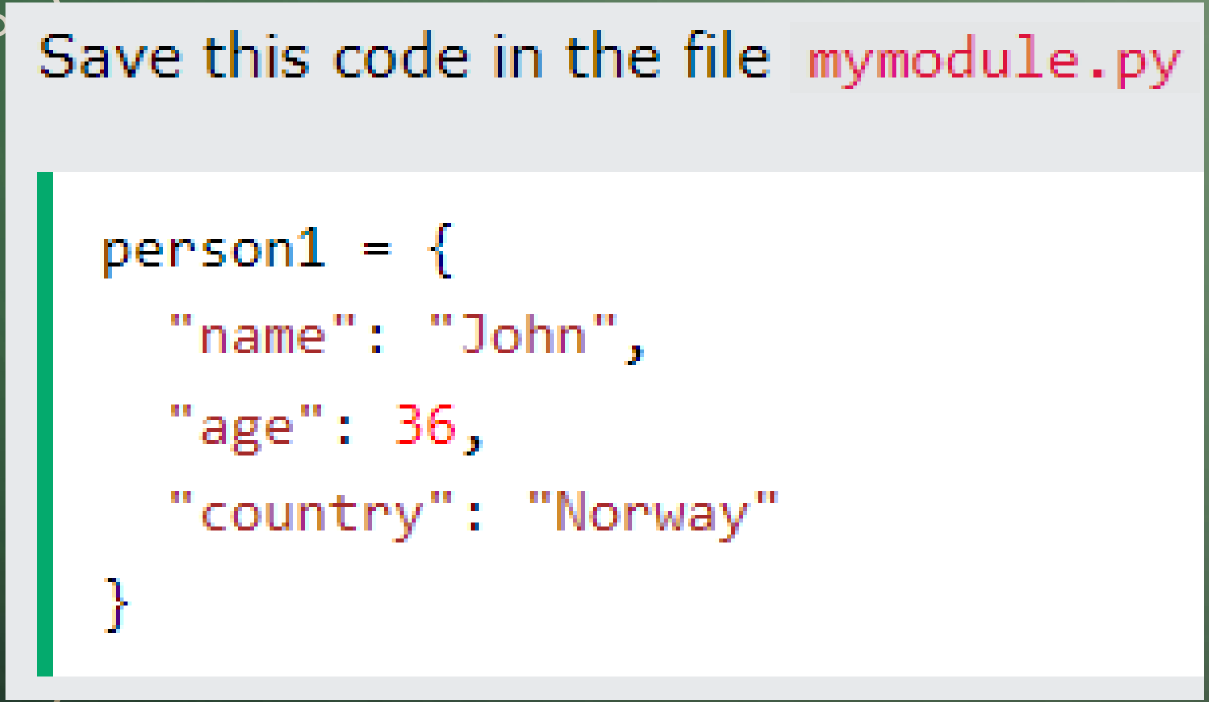
Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("Jonathan")
```

```
Hello, Jonathan
```

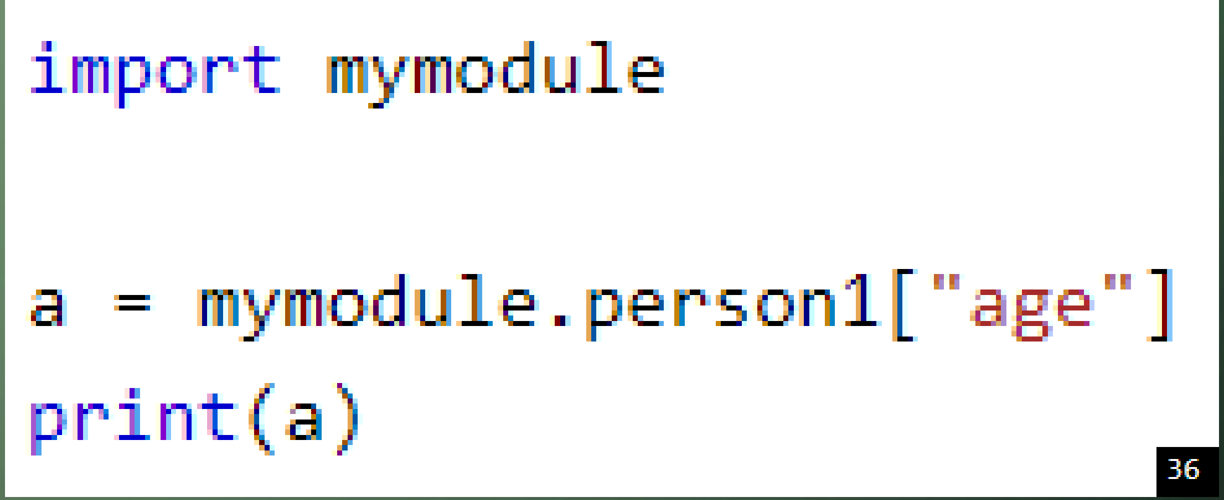
Note: When using a function from a module, use the syntax: *module_name.function_name*.

Variables in Module



Save this code in the file `mymodule.py`

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

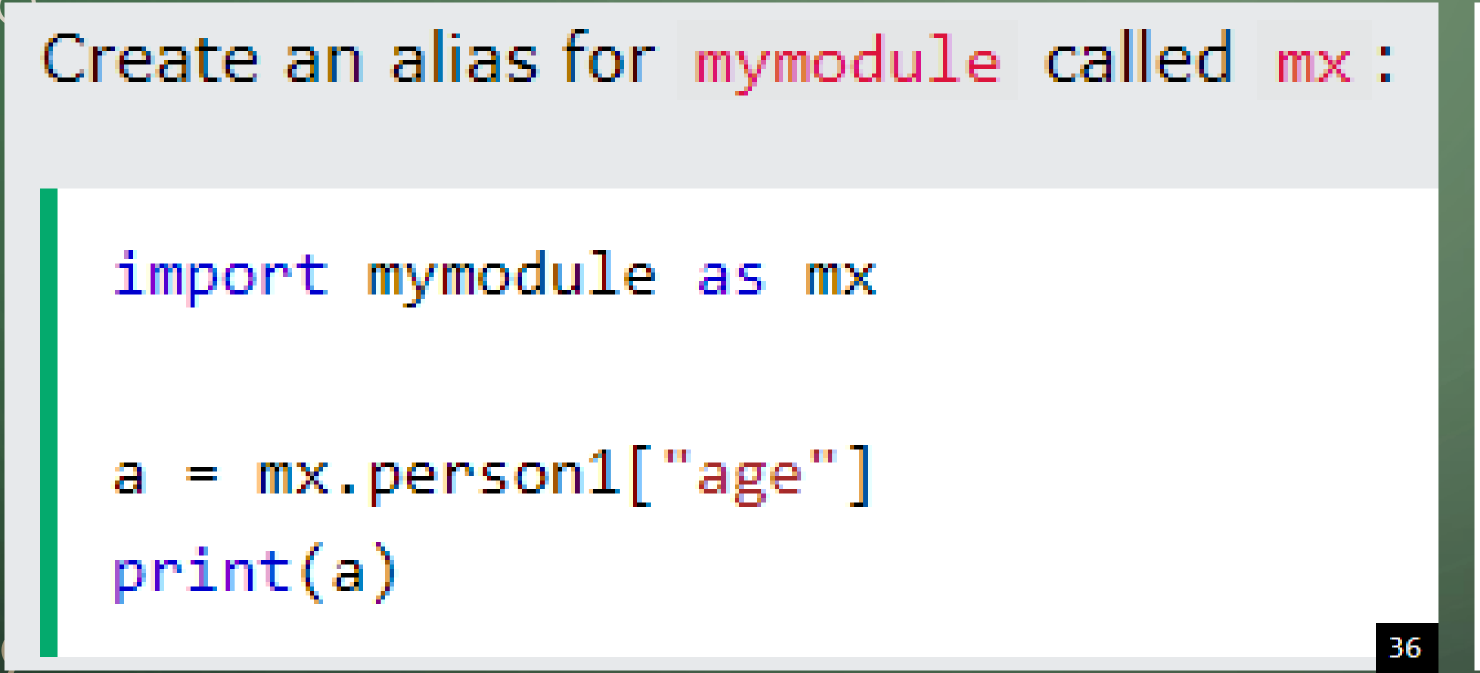


```
import mymodule

a = mymodule.person1["age"]
print(a)
```

```
36
```

- The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Re-naming a Module

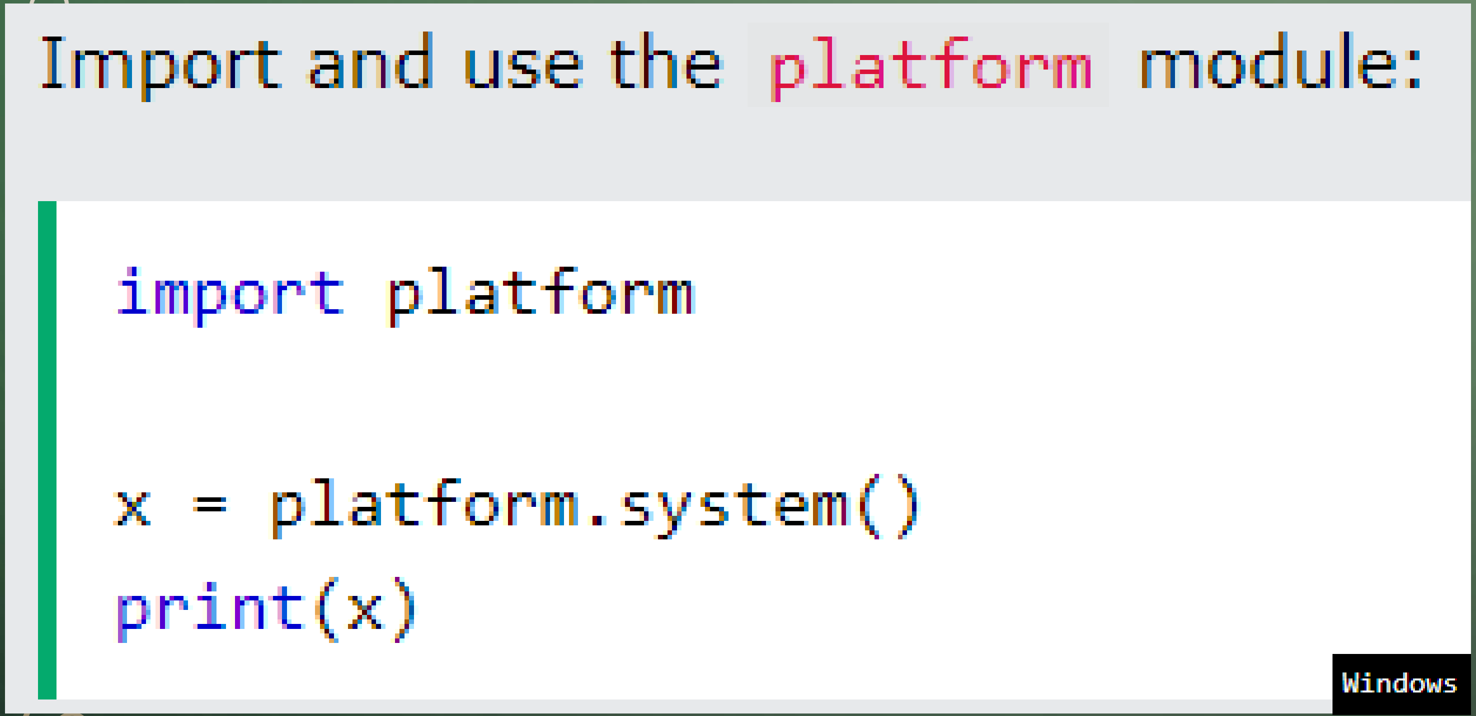


Create an alias for `mymodule` called `mx`:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

- You can name the module file whatever you like, but it must have the file extension .py
- You can create an alias when you import a module, by using the as keyword:

Built-in Modules



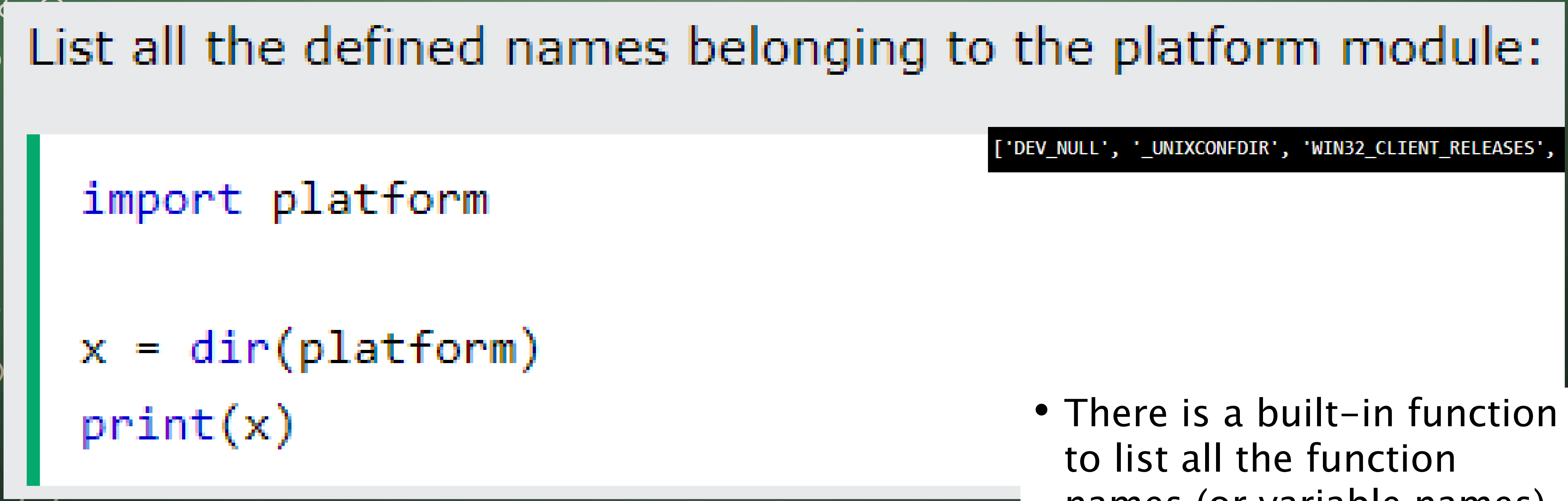
Import and use the `platform` module:

```
import platform

x = platform.system()
print(x)
```

Windows

- There are several built-in modules in Python, which you can import whenever you like.

Using the dir() Function



List all the defined names belonging to the platform module:

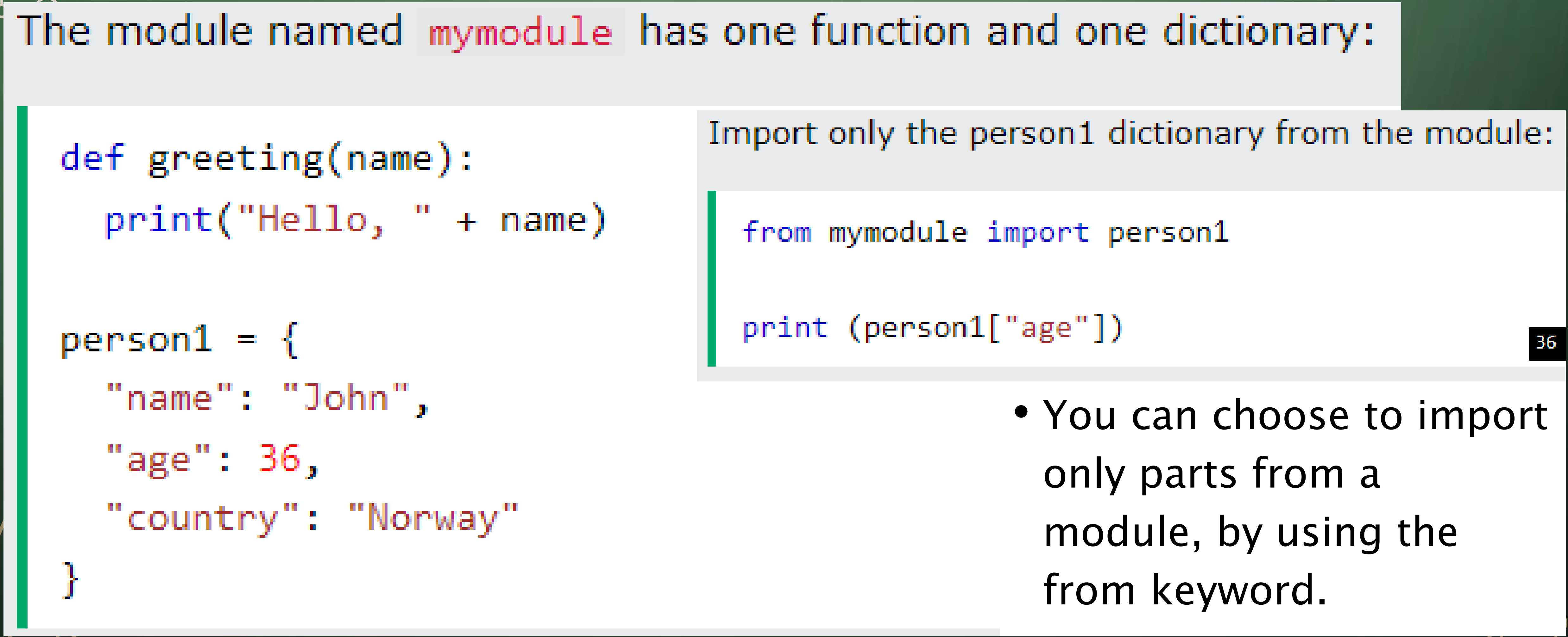
```
import platform

x = dir(platform)
print(x)
```

```
['DEV_NULL', '_UNIXCONFDIR', 'WIN32_CLIENT_RELEASES',
```

- There is a built-in function to list all the function names (or variable names) in a module. The dir() function:

Note: The dir() function can be used on all modules, also the ones you create yourself.

Import From Module



The module named mymodule has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

- You can choose to import only parts from a module, by using the from keyword.

REFERENCES:

- Learn Python Programming. (2023). https://www.tutorialsteacher.com/python
- Modules and Packages. (n.d.). https://www.learnpython.org/en/Modules_and_Packages
- Python Tutorial. (2022). https://www.w3resource.com/python/python-tutorial.php
- Python Tutorial. (n.d.). https://www.tutorialspoint.com/python/index.htm
- Python Tutorial. (n.d.). https://www.w3schools.com/python/default.asp
- Sturtz, John. (n.d.). Python Modules and Packages – An Introduction. https://realpython.com/python-modules-packages/