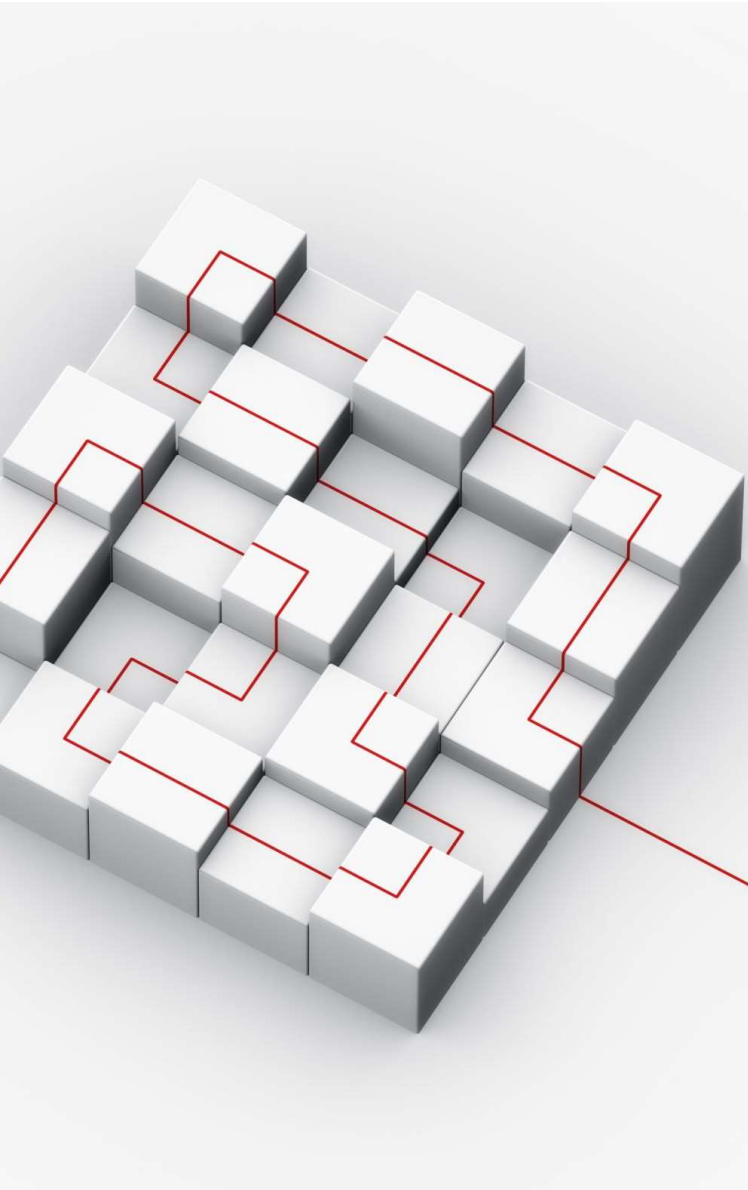


Module

Advanced Topics



This chapter covers

- Module Design Concepts
- Data Hiding in Modules
- Enabling Future Language Features
- Mixed Usage Modes
- Changing the Module Search Path
- The as Extension for import and from

Module Design Concepts

- You're always in a module in Python.
 - There's no way to write code that doesn't live in some module.
 - Even code typed at the interactive prompt really goes in a built-in module called `__main__`; the only unique things about the interactive prompt are that code runs and is discarded immediately, and expression results are printed automatically.

Module Design Concepts

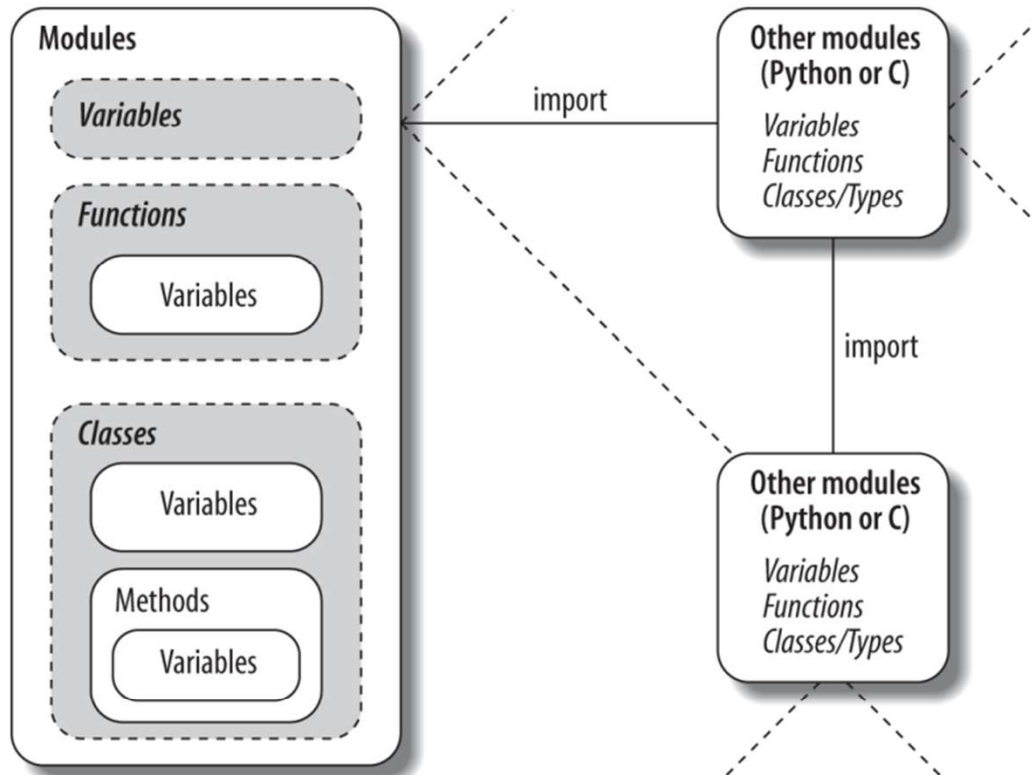
- Minimize module coupling: global variables.
 - Like functions, modules work best if they're written to be closed boxes.
 - As a rule of thumb, they should be as independent of global variables used within other modules as possible, except for functions and classes imported from them.
 - The only things a module should share with the outside world are the tools it uses, and the tools it defines.

Module Design Concepts

- Maximize module cohesion: unified purpose.
 - You can minimize a module's couplings by maximizing its cohesion; if all the components of a module share a general purpose, you're less likely to depend on external names.

Module Design Concepts

- Modules should rarely change other modules' variables.
 - It's perfectly OK to use global variables defined in another module (that's how clients import services, after all), but changing global variables in another module is often a symptom of a design problem.
 - There are exceptions, of course, but you should try to communicate results through devices such as function arguments and return values, not cross-module changes.
 - Otherwise, your global variables' values become dependent on the order of arbitrarily remote assignments in other files, and your modules become harder to understand and reuse.



Data Hiding in Modules

- In Python, data hiding in modules is a convention, not a syntactical constraint.
- Some purists object to this liberal attitude toward data hiding, claiming that it means Python can't implement encapsulation.
- However, encapsulation in Python is more about packaging than about restricting.

Minimizing from * Damage: `_X` and `__all__`

- As a special case, you can prefix names with a single underscore (e.g., `_X`) to prevent them from being copied out when a client imports a module's names with a `from *` statement.
- This really is intended only to minimize namespace pollution; because `from *` copies out all names, the importer may get more than it's bargained for (including names that overwrite names in the importer).
- Underscores aren't "private" declarations: you can still see and change such names with other import forms, such as the `import` statement.

Minimizing from * Damage: `_X` and `__all__`

- Alternatively, you can achieve a hiding effect similar to the `_X` naming convention by assigning a list of variable name strings to the variable `__all__` at the top level of the module.
- When this feature is used, the `from *` statement will copy out only those names listed in the `__all__` list.
- In effect, this is the converse of the `_X` convention: `__all__` identifies names to be copied, while `_X` identifies names not to be copied.
- Python looks for an `__all__` list in the module first and copies its names irrespective of any underscores; if `__all__` is not defined, `from *` copies all names without a single leading underscore.

Minimizing from * Damage: `_X` and `__all__`

- Like the `_X` convention, the `__all__` list has meaning only to the `from *` statement form and does not amount to a privacy declaration: other import statements can still access all names, as the last two tests show.
- Still, module writers can use either technique to implement modules that are well behaved when used with `from *`.

Enabling Future Language Features: `__future__`

- Changes to the language that may potentially break existing code are usually introduced gradually in Python.
- They often initially appear as optional extensions, which are disabled by default.
- To turn on such extensions, use a special import statement of this form

```
from __future__ import featurename
```

Enabling Future Language Features: `__future__`

- When used in a script, this statement must appear as the first executable statement in the file (possibly following a docstring or comment), because it enables special compilation of code on a per-module basis.
- It's also possible to submit this statement at the interactive prompt to experiment with upcoming language changes; the feature will then be available for the remainder of the interactive session.

Enabling Future Language Features: `__future__`

- For a list of futurisms, you may import and turn on this way, run a `dir` call on the `__future__` module after importing it, or see its library manual entry.
- Per its documentation, none of its feature names will ever be removed, so it's safe to leave in a `__future__` import even in code run by a version of Python where the feature is present normally.

Mixed Usage Modes: `__name__` and `__main__`

- The next module-related trick lets you both import a file as a module and run it as a standalone program and is widely used in Python files.
- It's so simple that some miss the point at first: each module has a built-in attribute called `__name__`, which Python creates and assigns automatically as follows:
 - If the file is being run as a top-level program file, `__name__` is set to the string `"__main__"` when it starts.
 - If the file is being imported instead, `__name__` is set to the module's name as known by its clients.

Mixed Usage Modes: `__name__` and `__main__`

- The upshot is that a module can test its own `__name__` to determine whether it's being run or imported.

```
def tester():  
    print("It's Christmas in Heaven...")  
  
if __name__ == '__main__':  
    tester()
```

Only when run
Not when imported

Mixed Usage Modes: `__name__` and `__main__`

- In effect, a module's `__name__` variable serves as a usage mode flag, allowing its code to be leveraged as both an importable library and a top-level script.
- Though simple, you'll see this hook used in most of the Python program files you are likely to encounter in the wild - both for testing and dual usage.
- For instance, perhaps the most common way you'll see the `__name__` test applied is for self-test code. In short, you can package code that tests a module's exports in the module itself by wrapping it in a `__name__` test at the bottom of the file. This way, you can use the file in clients by importing it, but also test its logic by running it from the system shell or via another launching scheme.

Changing the Module Search Path

- Previously, we learned that the module search path is a list of directories that can be customized via the environment variable `PYTHONPATH`, and possibly via `.pth` files.
- `sys.path` is initialized on startup, but thereafter you can delete, append, and reset its components however you like.

[illegible]

Changing the Module Search Path

- Thus, you can use this technique to dynamically configure a search path inside a Python program.
- Be careful, though: if you delete a critical directory from the path, you may lose access to critical utilities.
- Also, remember that such `sys.path` settings endure for only if the Python session or program (technically, process) that made them runs; they are not retained after Python exits.
- By contrast, `PYTHONPATH` and `.pth` file path configurations live in the operating system instead of a running Python program, and so are more global: they are picked up by every program on your machine and live on after a program completes.

The as Extension for import and from

- Consider the following import statement;
`import modulename as name`
- This extension is commonly used to provide short synonyms for longer names, and to avoid name clashes when you are already using a name in your script that would otherwise be overwritten by a normal import statement.

```
import reallylongmodulename as name  
name.func()
```

Module Gotchas

Module Name Clashes: Package and Package-Relative Imports

- If you have two modules of the same name, you may only be able to import one of them —by default, the one whose directory is leftmost in the `sys.path` module search path will always be chosen.
- This isn't an issue if the module you prefer is in your top-level script's directory; since that is always first in the module path, its contents will be located first automatically.
- For cross-directory imports, however, the linear nature of the module search path means that same-named files can clash.

Solutions

- Avoid same-named files or use the package imports feature.
- If you need to get to both same-named files, structure your source files in sub-directories, such that package import directory names make the module references unique.
- If the enclosing package directory names are unique, you'll be able to access either or both same-named modules.

Statement Order Matters in Top-Level Code

- Code at the top level of a module file (not nested in a function) runs as soon as Python reaches it during an import; because of that, it cannot reference names assigned lower in the file.
- Code inside a function body doesn't run until the function is called; because names in a function aren't resolved until the function runs, they can usually reference names anywhere in the file.


```
func1() # Error: "func1" not yet assigned
```

```
def func1():  
    print(func2())
```

OK: "func2" looked up later

```
func1() # Error: "func2" not yet assigned
```

```
def func2():  
    return "Hello"
```

```
func1() # OK: "func1" and "func2" assigned
```

from Copies Names but Doesn't Link

- The from statement is the source of a variety of potential gotchas in Python.
- As we've learned, the from statement is really an assignment to names in the importer's scope - a name-copy operation, not a name aliasing.

from * Can Obscure the Meaning of Variables

- Because you don't list the variables you want when using the from module import * statement form, it can accidentally overwrite names you're already using in your scope.
- Worse, it can make it difficult to determine where a variable comes from.
- This is especially true if the from * form is used on more than one imported file.

```
>>> from module1 import *           # Bad: may overwrite my names silently
>>> from module2 import *           # Worse: no way to tell what we get!
>>> from module3 import *
>>> . . .

>>> func()                          # Huh???
```

Solutions

- Try to explicitly list the attributes you want in your from statements and restrict the from * form to at most one imported module per file.
- That way, any undefined names must by deduction be in the module named in the single from *.
- You can avoid the issue altogether if you always use import instead of from, but that advice is too harsh; like much else in programming, from is a convenient tool if used wisely.

reload May Not Impact from Imports

- Names imported with from simply become references to objects, which happen to have been referenced by the same names in the importee when the from ran.

```
>>> import nested1
>>> nested1.X
99
>>> nested1.X = 100
>>> nested1.X
100
>>> from importlib import reload
>>> reload(nested1)
<module 'nested1' from 'C:\\pythoncodes\\modules\\nested1.py'>
>>> nested1.X
99
```

Recursive from Imports May Not Work

- Because imports execute a file's statements from top to bottom, you need to be careful when using modules that import each other.
- This is often called recursive imports, but the recursion doesn't really occur.

Exercises

The End