

Modules

# The Big Picture

# This chapter covers

- Why Use Modules?
- Python Program Architecture
- How Import Work
- The Module Search Path

# Introduction

- This module begins the in-depth look at the Python module – the highest-level program organization unit, which packages program code and data for reuse, and provides self-contained namespaces that minimize variable name clashes across your programs.
- Each file is a module, and modules import other modules to use the names they define.
- Modules might also correspond to extensions coded in external languages such as C, Java, or C#, and even to directories in package imports.

# What is a module?

- A module is a file containing code.
- It defines a group of Python functions or other objects, and the name of the module is derived from the name of the file.

# Why Use Modules?

- In short, modules provide an easy way to organize components into a system by serving as self-contained packages of variables known as *namespaces*.
- All the names defined at the top level of a module file become attributes of the imported module object.

# Modules have three roles

- Code Reuse
- System Namespace Partitioning
- Implementing Shared Services or Data

# Code Reuse

- Modules let you save code in files permanently.
- Unlike code you type at the Python interactive prompt, which goes away when you exit Python, code in module files is persistent - it can be reloaded and rerun as many times as needed.
- Just as importantly, modules are a place to define names, known as attributes, which may be referenced by multiple external clients.
- When used well, this supports a modular program design that groups functionality into reusable units.

# System Namespace Partitioning

- Modules are also the highest-level program organization unit in Python.
- Although they are fundamentally just packages of names, these packages are also self-contained - you can never see a name in another file, unless you explicitly import that file.
- Much like the local scopes of functions, this helps avoid name clashes across your programs.
- In fact, you can't avoid this feature—everything “lives” in a module, both the code you run and the objects you create are always implicitly enclosed in modules.
- Because of that, modules are natural tools for grouping system components.

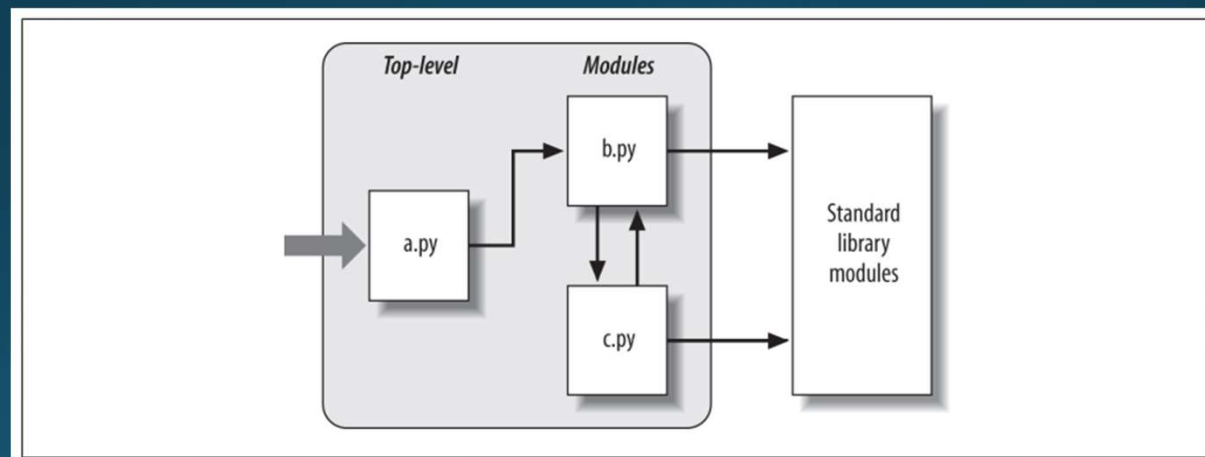


# Implementing Shared Services or Data

- From an operational perspective, modules are also useful for implementing components that are shared across a system and hence require only a single copy.
- For instance, if you need to provide a global object that's used by more than one function or file, you can code it in a module that can then be imported by many clients.

# Python Program Architecture

- At a base level, a Python program consists of text files containing Python *statements*, with one main *top-level* file, and zero or more supplemental files known as *modules*.



# How Imports Work

1. Find the module's file.
2. Compile it to byte code (if needed).
3. Run the module's code to build the objects it defines

# The Module Search Path

1. The home directory of the program.
2. PYTHONPATH directories (if set).
3. Standard library directories.
4. The contents of any .pth files (if present).
5. The site-packages home of third-party extensions.

# A first module

- Create a text file called mymath.py, and in that text file, enter the Python code in listing 10.1.

```
1  """mymath - our example math module"""
2  pi = 3.14159
3  def area(r):
4      """area(r): return the area of a circle with radius r."""
5      global pi
6      return(pi * r * r)
```

# To use the module

- Import the module
- Start using the module

```
1  import mymath
2
3  print(mymath.pi)
4
5  from mymath import pi
6
7  print(pi)
```

# Modules

- A module is a file defining Python objects.
- If the name of the module file is `modulename.py`, the Python name of the module is `modulename`.
- You can bring a module named `modulename` into use with the `import modulename` statement. After this statement is executed, objects defined in the module can be accessed as `modulename.objectname`.
- Specific names from a module can be brought directly into your program by using the `from modulename import objectname` statement. This statement makes `objectname` accessible to your program without your needing to prepend it with `modulename`, and it's useful for bringing in names that are often used.

# Where to place your own modules

- Place your modules in one of the directories that Python normally searches for modules.
- Place all the modules used by a Python program in the same directory as the program.
- Create a directory (or directories) to hold your modules, and modify the `sys.path` variable so that it includes this new directory (or directories).



# Private names in modules

- The exception is that identifiers in the module beginning with an **underscore** can't be imported with `from module import *`.
- By starting all internal names (that is, names that shouldn't be accessed outside the module) with an underscore, you can ensure that `from module import *` brings in only those names that the user will want to access.

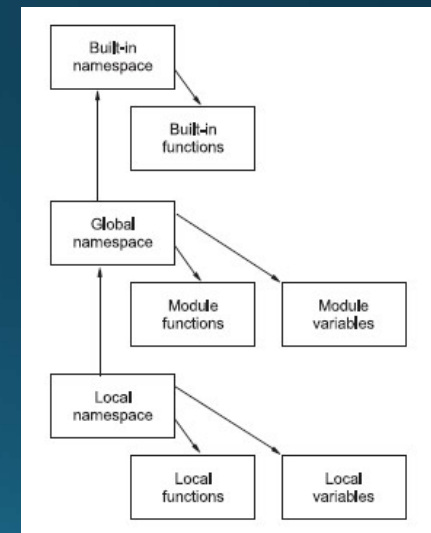
```
1  """modtest: our test module"""
2  def f(x):
3      |   return x
4
5  def _g(x):
6      |   return x
7
8  a = 4
9  _b = 2
```

# Library and third-party modules

- After you've installed Python, all the functionality in these library modules is available to you.
- All that's needed is to import the appropriate modules, functions, classes, and so forth explicitly, before you use them.
- Available third-party modules and links to them are identified in the Python Package Index (pyPI), which will be discussed in chapter 19.
- You need to download these modules and install them in a directory in your module search path to make them available for import into your programs.

# Python scoping rules and namespaces

- A namespace in Python is a mapping from identifiers to objects—that is, how Python keeps track of what variables and identifiers are active and what they point to.



Lab

# Create a module

# Summary

- Python modules allow you to put related code and objects into a file.
- Using modules also helps prevent conflicting variable names, because imported
- objects are normally named in association with their module.