

# Packages

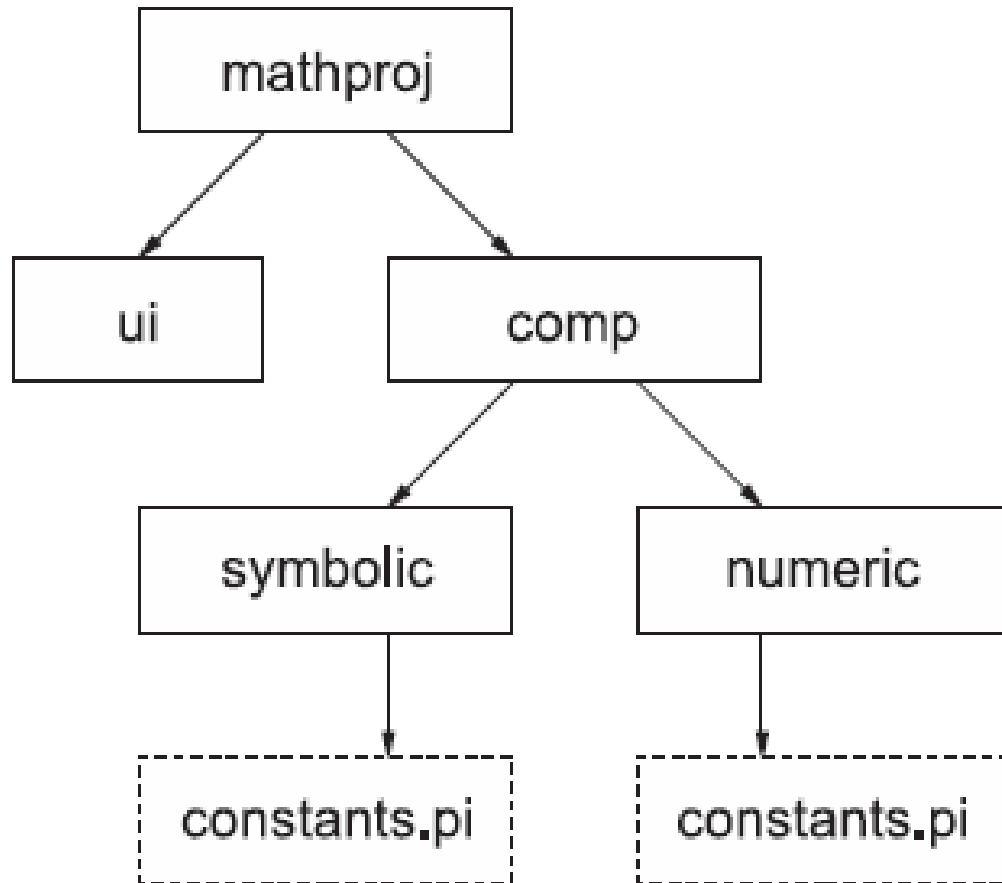
## Chapter 18

# This chapter covers

- Defining a package
- Creating a simple package
- Exploring a concrete example
- Using the `__all__` attribute
- Using packages properly

# What is a package?

- A module is a file containing code.
- A module defines a group of usually related Python functions or other objects.
- The name of the module is derived from the name of the file.
- A package is a directory containing code and possibly further subdirectories.
- A package contains a group of usually related code files (modules).
- The name of the package is derived from the name of the main package directory.
- Packages are a natural extension of the module concept and are designed to handle very large projects.
- Just as modules group related functions, classes, and variables, packages group related modules.



# A first example

# Proper use of packages

- Packages shouldn't use deeply nested directory structures. Except for absolutely huge collections of code, there should be no need to do so. For most packages, a single top-level directory is all that's needed. A two-level hierarchy should be able to effectively handle all but a few of the rest. As written in The Zen of Python, by Tim Peters (see appendix A), "Flat is better than nested."
- Although you can use the `__all__` attribute to hide names from `from ... import *` by not listing those names, doing so probably is not a good idea, because it's inconsistent. If you want to hide names, make them private by prefacing them with an underscore.

# Lab

## Create a Package

# Summary

- Packages let you create libraries of code that span multiple files and directories.
- Using packages allows better organization of large collections of code than single modules would allow.
- You should be wary of nesting directories in your packages more than one or two levels deep unless you have a very large and complex library.