

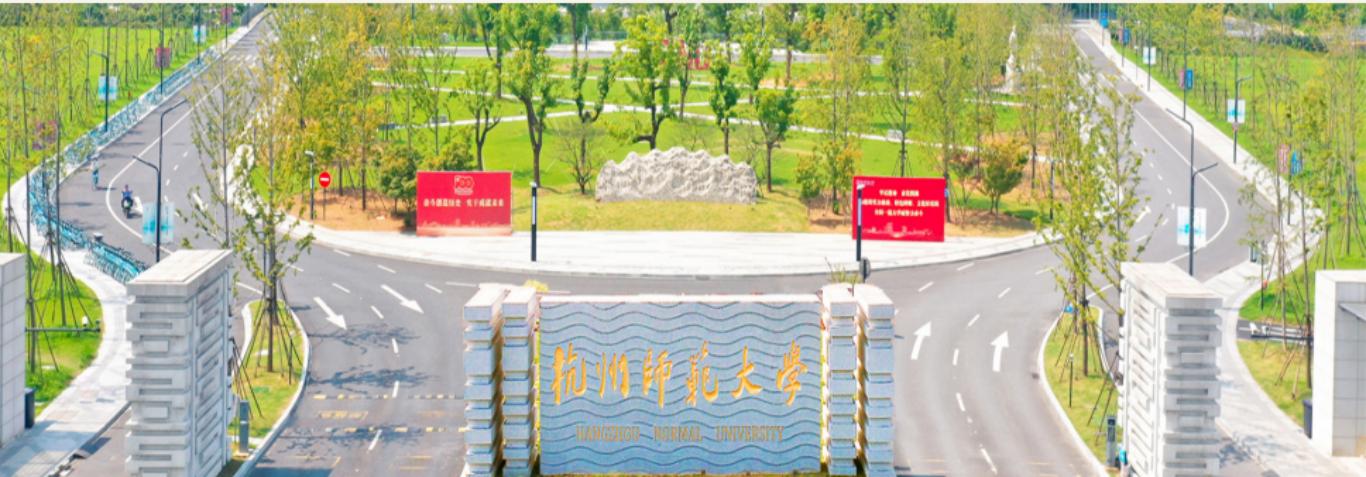
第十二章 - 代码组织 (编程练习)

张建章

阿里巴巴商学院

杭州师范大学

2025-09



1 Python代码组织的基本概念

2 常用的Python标准库模块

3 模块的定义与使用

4 第三方库

代码示例

Python 中，代码组织的基本概念包括模块（Module）、包（Package）和库（Library）。这三个代码组织方式形成了一个层次化的结构：

```
库 (Library)
├── 包1 (Package)
│   ├── __init__.py
│   ├── 模块1.py
│   └── 模块2.py
└── 包2 (Package)
    ├── __init__.py
    ├── 模块3.py
    └── 模块4.py
```

模块 (Module): 模块是一个包含 Python 定义和语句的文件，通常以 `.py` 作为扩展名。模块将相关的代码组织在一起，便于重用和维护。例如，创建一个名为 `math_operations.py` 的模块，内容如下：

```
# math_operations.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

在其他脚本中，可以通过 `import` 语句导入该模块并使用其中的函数：

```
import math_operations

result = math_operations.add(5, 3)
print(result) # 输出: 8
```

Python 提供了大量的内置模块，可以通过 `import` 语句直接导入。例如，使用 `math` 模块计算平方根：

```
import math

result = math.sqrt(16)
print(result) # 输出: 4.0
```

使用 `copy` 模块进行深复制：

```
import copy

l = [1, 2, [3,4]]
new_l = copy.deepcopy(l)
```

包 (Package): 包是一个包含多个模块的目录，用于组织相关的模块。在 Python 3.3 之前，包目录中必须包含一个 `__init__.py` 文件，以表示该目录是一个包。在 Python 3.3 及之后的版本中，`__init__.py` 文件变为可选，但通常仍会包含该文件以明确表示包的存在。例如，创建一个名为 `mypackage` 的包，结构如下：

```
mypackage/  
    __init__.py  
    module1.py  
    module2.py
```

在其他脚本中，可以导入包中的模块：

```
from mypackage import module1
```

Python 自带了一些包，例如 `os` 包，其包含与操作系统交互的多个模块。可以直接导入并使用其功能，例如：

```
import os

# 列出当前目录下的文件
files = os.listdir('.')
print(files)

# 返回当前目录的绝对路径
os.path.abspath('..')
```

1. Python代码组织的基本概念

库 (Library): 库是一个包含多个包和模块的集合，提供特定的功能或解决方案。例如，`NumPy` 是一个用于科学计算的库，包含多个包和模块，提供对大型多维数组和矩阵的支持，以及大量的数学函数。以下是一个典型 Python 库的目录结构示例：

```
numpy/
    __init__.py
    core/
        __init__.py
        multiarray.py
        umath.py
    linalg/
        __init__.py
        lapack_lite.py
    fft/
        __init__.py
        fftpack.py
```

该结构展示了**库的分层设计**，每个包提供了特定功能，例如，`core` 提供核心数组操作，`linalg` 提供线性代数功能，而 `fft` 处理快速傅里叶变换。

Python 标准库是随 Python 解释器一同发布的包和模块的集合，提供了丰富的功能，涵盖文件操作、系统交互、网络通信、数据处理等多个方面。利用标准库，开发者无需编写大量基础代码，即可实现复杂的功能，从而提高开发效率和代码质量。

代码示例

讲义表12.1列出了常用的 Python 标准库中的包和模块及其基本功能和应用场景。

random 模块： `random` 模块用于生成随机数，支持各种分布的随机数生成，以及从序列中随机选择元素等功能。

(1) 生成 0 到 1 之间的随机浮点数：

```
import random

# 生成随机浮点数
random_number = random.random()
print(random_number)
```

(2) 生成指定范围内的随机整数：

```
import random

# 生成 1 到 10 之间的随机整数
random_integer = random.randint(1, 10)
print(random_integer)
```

(3) 从列表中随机选择元素：

```
import random

# 定义列表
items = ['apple', 'banana', 'cherry']

# 随机选择一个元素
random_item = random.choice(items)
print(random_item)
```

1. 编写自定义模块：

自定义模块本质上是一个包含 Python 代码的文件，以 `.py` 作为扩展名。在该文件中，可以定义函数、类和变量等。例如，创建一个名为 `string_utils.py` 的文件，内容如下：

```
# string_utils.py

def to_uppercase(s):
    """将字符串转换为大写"""
    return s.upper()

def count_vowels(s):
    """统计字符串中元音字母的数量"""
    vowels = 'aeiouAEIOU'
    return sum(1 for char in s if char in vowels)
```

在其他 Python 脚本中，可以通过 `import` 语句导入该模块并使用其中的函数：

```
import string_utils

text = "Hello World"
uppercase_text = string_utils.to_uppercase(text)
vowel_count = string_utils.count_vowels(text)

print(uppercase_text)  # 输出: HELLO WORLD
print(vowel_count)   # 输出: 3
```

2. 模块文件的命名和存储位置对模块导入的影响：

命名规范：根据 PEP 8 的建议，模块名称应简短且全部使用小写字母，必要时可使用下划线以提高可读性。

存储位置： Python 解释器通过 `sys.path` 列表中的目录来搜索模块。因此，**模块文件应存放在这些目录中，或者与导入它的脚本位于同一目录。**

3. 模块功能探索

在 Python 编程中，了解模块的内容和功能对于高效开发至关重要。可以使用内置的 `dir()` 和 `help()` 函数，以及 `__doc__` 属性，快速获取模块、函数或类的相关信息。

- 使用 `dir()` 函数获取模块的属性和方法列表；
- 使用 `help()` 函数获取模块、函数或类的详细信息；
- 使用 `__doc__` 属性查看文档字符串；

4. 导入包与模块中的内容

导入包和模块中的内容是组织代码、提高可读性和重用性的关键。

(1) 导入整个模块： 使用 `import` 语句导入模块后，需通过 `模块名.对象名` 的方式访问其中的函数或变量。例如：

```
import math
print(math.pi)  # 输出: 3.141592653589793
```

此方式将整个 `math` 模块导入，访问其内容时需使用 `math` 作为前缀。

(2) 从模块中导入特定函数或变量： 使用 `from...import...` 语句直接导入模块中的特定对象，可直接使用其名称，无需模块前缀。例如：

```
from math import pi
print(pi)  # 输出: 3.141592653589793
```

此方式仅导入 `pi`，而非整个 `math` 模块。

3. 模块的定义与使用

(3) 为导入的包、模块或函数指定别名: 使用 `as` 关键字为导入的包、模块或函数指定别名，**简化代码书写或避免命名冲突**。例如：

```
import numpy as np  
from string import capwords as cp
```

此方式将 `numpy` 包重命名为 `np`，`capwords` 函数重命名为 `cp`，简化代码书写。

(4) 从包的模块中导入特定函数或变量: 使用 `from...import...` 语句从包的特定模块中导入所需对象。例如：

```
from os.path import abspath  
  
# 直接调用导入的函数  
myabs_path = abspath('..')
```

此方式仅导入 `abspath` 函数，调用时无需指定包和模块前缀。

4. 第三方库

在 Python 编程中，第三方库是由外部开发者创建的代码集合，旨在提供特定的功能或解决方案，供其他项目复用。这些库通常被打包并发布在 **Python 包索引 (PyPI)** (<https://pypi.org/>) 上，开发者可以使用包管理工具如 **pip** 进行安装和管理。通过利用第三方库，开发者能够避免重复编写常见或复杂的代码，节省时间和精力。

代码示例



pip 安装和管理

pip 是官方推荐的包管理工具，用于**安装和管理第三方库**。通过 **pip**，可以从 **Python 包索引 (PyPI)** 以及其他索引安装软件包。

1. 安装和升级 pip :

在大多数 Python 发行版中，pip 已经预装。可以通过以下命令检查 pip 是否已安装：

```
pip --version
```

如果未安装 pip，可以使用以下命令进行安装：

```
python -m ensurepip --default-pip
```

为了确保使用最新版本的 pip，建议运行以下命令进行升级：

```
python -m pip install --upgrade pip
```

2. 安装第三方库：

使用 pip 安装第三方库非常简便。例如，安装名为 requests 的库，可以执行命令 pip install requests，此命令会从 PyPI 下载并安装 requests 及其所有依赖项。

3. 升级库：

如果需要升级已安装的库，可以使用以下命令：

```
pip install -U requests
```

此命令会将 `requests` 库升级到最新版本。

4. 卸载库：

如果需要卸载已安装的库，可以使用以下命令：

```
pip uninstall requests
```

此命令会从环境中移除 `requests` 库。

5. 列出已安装的库：

要查看当前环境中已安装的所有库，可以运行：

```
pip list
```

此命令会显示已安装库的列表及其版本号。

6. 使用虚拟环境：

为了避免不同项目之间的依赖冲突，建议为每个项目创建独立的虚拟环境。可以使用 `venv` 模块创建虚拟环境：

```
python -m venv myenv
```

激活虚拟环境后，使用 `pip` 安装的库将仅作用于该环境，确保项目的依赖独立性。

7. 使用 `requirements.txt` 管理依赖：

在项目开发中，通常会使用 `requirements.txt` 文件来记录项目的所有依赖库。可以通过以下命令生成该文件：

```
pip freeze > requirements.txt
```

此命令将当前环境中所有库及其版本信息写入 `requirements.txt`。在新的环境中，可以使用以下命令根据该文件安装所有依赖：

```
pip install -r requirements.txt
```

寻找第三方库的主要途径包括：

- **Python 包索引 (PyPI):** PyPI 是官方的第三方软件仓库，提供了超过 50 万个 Python 包，涵盖从数据分析到网络开发等各个领域。
- **GitHub:** 作为全球最大的代码托管平台，GitHub 上托管了大量的 Python 项目和库。开发者可以搜索特定功能的库，并查看其源代码、文档和更新情况。
- **Awesome Python 列表:** 这是一个由社区维护的精选 Python 库和工具的列表，涵盖了不同的应用领域，帮助开发者快速找到高质量的第三方库。
- **官方文档和社区论坛:** Python 的官方文档和社区论坛（如，Stack Overflow）也是获取第三方库信息的重要来源。在这些平台上，开发者可以找到推荐的库、使用示例以及其他开发者的经验分享。

THE END