

## | STAT243 Lecture 5 Programming

### | 1 Text manipulation, string processing and regular expressions (regex)

#### 🔗 Logic ▾

- Python 中的文本处理与 UNIX, R, Perl 等语言有许多相似之处, 许多概念和工具源自 UNIX 系统.
- 本文中提到的 "string" 指的是由 character 组成的序列, 可包含数字, 空白符 (包括换行符) 和特殊字符, 通常存储为 `str` 类对象.
- 我们将重点介绍 Python 中处理字符串的功能, 特别是使用 `re` 包进行正则表达式操作.

#### ⚠ Remark ▾

`re` 包提供的是 Perl 风格的正则表达式, 但不支持 named character classes (如 `[digit:]`), 应该使用 `\d` 或 `[0-9]` 等字符类.

### | 1.1 Finding Patterns 模式查找

#### | 1.1.1 查找单个匹配项

在 Python 中, 可以使用 matching function `re.search()` 来查询结果以获取匹配内容和位置信息.



#### Python

```
1 import re
2 text = "Here's my number: 919-543-3300."
3 m = re.search("\\d+", text)    ## 也可以使用 m = re.search(r"\d+", text)
4 print(m)                      ## <re.Match object; span=(18, 21), match='919'>
5 print(m.group())              ## '919'
6 print(m.start())              ## 18
7 print(m.end())                ## 21
8 print(m.span())               ## (18, 21)
```

#### ⚠ Remark ▾

1. 注意: `re.search()` 只返回第一个匹配项
2. [Discussion of special characters](#) 解释了为什么我们使用 `\\d` 而不是 `\d`.  
大致原因是: 因为有 `\n` 这类特殊字符的存在, `\d` 并不会被直接传入正则引擎, 而是会先被误编译成特殊字符. 若使用 `\\d`, 则 Python 解释器会先将 `\\d` 编译为 `\d`, 然后再传入正则引擎匹配

#### | 1.1.2 查找所有匹配项

若要获取所有匹配项, 可使用 `re.findall()`:



#### Python

```
1 re.findall("\\d+", text)    ## ['919', '543', '3300']
2                               ## 也可以使用 re.findall(r"\d+", text)
```

等价于先使用 `re.compile()` 进行编译再使用 `re.findall()`:



#### Python

```
1 pattern = re.compile("\\d+")
2 re.findall(pattern, text)
```

#### ⚠ Remark ▾

- 显式编译正则表达式 (`re.compile`) 在复杂模式或重复使用时更高效
- 正则表达式是一种可编译的独立语言。

### 1.1.3 正则表达式的 flags

忽略大小写匹配:

```
Python
1 text = "That cat in the Hat"
2 re.findall("hat", text, re.IGNORECASE) ## ['hat', 'Hat']
```

其他有用的正则表达式标志包括

- `re.VERBOSE` (允许编写带注释的正则表达式)
- `re.MULTILINE` (多行匹配)

### 1.1.4 使用 list comprehension 处理多个字符串

可使用 list comprehension 处理多个字符串:

```
Python
1 def return_group(pattern, txt):
2     m = re.search(pattern, txt)
3     return m.group() if m else None
4
5 texts = [
6     "Here's my number: 919-543-3300.",
7     "hi John, good to meet you",
8     "They bought 731 bananas",
9     "Please call 1.919.554.3800"
10 ]
11 [return_group(r"\d+", s) for s in texts] ## ['919', None, '731', '1']
```

#### ⚠ Remark ▾

需注意处理未匹配的情况, 否则会报错

### 1.1.5 位置匹配与重复匹配

可使用 `^` 和 `$` 匹配字符串的开头或结尾:

```
Python
1 text = "hats are all that are important to a hatter."
2 re.findall(r"^hat\w+", text) ## ['hats']
```

使用重复符号 `{}` 匹配多次出现:

```
Python
```

```
1 text = "Here's my number: 919-543-3300. They bought 731 hats. Please call 1.919.554.3800."
2 re.findall(r"\d{3}[-.]\d{3}[-.]\d{4}", text) ## ['919-543-3300', '919.554.3800']
```

更通用的电话号码匹配 (允许前缀"1-"或"1."):

```
Python
1 re.findall(r"((1[-.]?)\d{3}[-.])\d{1,2}\d{4})", text)
2 ## [('919-543-3300', '-', '543-'), ('1.919.554.3800', '1.', '554.')]

```

#### ⚠ Remark ▾

- 上述正则表达式可能匹配无效电话号码, 需谨慎设计
- 上述正则表达式使用了 capturing group `()`, 因此返回了一个 list of tuple, capturing group 会在下文中详细解释

## 1.1.6 使用 `finditer` 进行迭代 (惰性) 匹配

对于大文本, 建议使用 `finditer` 进行惰性匹配:

```
Python
1 it = re.finditer(r"(http|ftp)://", text)
2 ## 或 it = re.finditer("(http|ftp):\\|\\|/", text)
3
4 for match in it:
5     print(match.span())

```

这种方法类似于 `pandas.read_csv(chunksize=n)`, 会返回一个 `iterator`, 逐次返回匹配结果

## 1.2 Replacing Patterns 模式替换

### 1.2.1 替换匹配项

使用 `re.sub()` 替换匹配的子字符串:

```
Python
1 text = "Here's my number: 919-543-3300."
2 re.sub(r"\d", "Z", text) ## "Here's my number: ZZZ-ZZZ-ZZZZ."

```

### 1.2.2 分组与捕获

使用 `()` 进行分组, 可控制返回的匹配内容:

```
Python
1 text = "At the site http://www.ibm.com. Some other text. ftp://ibm.com"
2 re.search(r"(http|ftp)://", text).group() ## 'http://'
3 re.search(r"(http|ftp)://", text).group(0) ## 'http://'
4 re.search(r"(http|ftp)://", text).group(1) ## 'http'

```

使用 `findall` 时, 若有 grouping operator, 则只返回 "captured groups", 我们可以通过在最外层添加一个额外的 grouping operator 来捕获完整的 pattern:

```
Python
1 re.findall(r"(http|ftp)://", text) ## ['http', 'ftp']
2 re.findall(r"((http|ftp)://)", text) ## [('http://', 'http'), ('ftp://', 'ftp')]

```

使用 `(?:...)` (non-capturing) 来匹配完整的 pattern 而不是 inner group:

```
Python
1 re.findall(r"((?:http|ftp)://)", text) ## ['http://', 'ftp://']
```

### 1.2.3 对 captured group 进行替换

可在替换字符串中引用捕获组 (如 `\1`):

#### Example ▾

在数字前后添加下划线:

```
Python
1 text = "Here's my number: 919-543-3300. They bought 731 bananas. Please call 919.554.3800."
2 re.sub(r"([0-9]+)", "_\\1_", text)
3 ## "Here's my number: _919_-_543_-_3300_. They bought _731_ bananas. Please call _919_-_554_-_3800_."
```

#### Example ▾

移除非字段分隔符的逗号:

```
Python
1 text = '"H4NY07011","ACKERMAN, GARY L.,"H","$13,242",,,'
2 re.sub(r'([^\,])',', r'\1', text)
3 ## '"H4NY07011","ACKERMAN GARY L.,"H","$13242",,,'
```

#### ⚠ Remark: 代码说明 ▾

- `[^\,]` 匹配非引号, 非逗号的字符, `\1` 引用该字符, 替换时去掉逗号.
- 当 `^` 出现在 `[]` 内部时, 表示取反; 否则表示句首

可以使用 `\2`, `\3` 等来指代多个 captured groups

```
Python
1 text = "Here's my number: 919-543-3300. They bought 731 bananas. Please call 919.554.3800."
2 re.sub(r"([0-9]{3})[-\.]([0-9]{3})[-\.]([0-9]{4})", "area code \\1, number \\2-\\3", text)
3 ## "Here's my number: area code 919, number 543-3300. They bought 731 bananas. Please call area code 919, number 554-3800."
```

#### ⚠ Remark ▾

以下功能未详细展开, 但值得了解:

- **命名分组**: 可为分组命名, 便于引用.
- **回调函数替换**: `re.sub` 可接受一个函数作为替换逻辑.
- **模式内引用**: 使用 `\1` 等语法在模式中引用之前捕获的组.

## 1.3 Greedy Matching 贪婪匹配

### 1.3.1 贪婪匹配

默认情况下, 正则表达式会匹配尽可能多的字符, 称为"贪婪匹配":

```
Python
1 text = "See the 998 balloons."
2 re.findall(r"\d+", text)  ## ['998'] 而不是仅返回第一个 9
```

### 1.3.2 非贪婪匹配

在某些情况下, 贪婪匹配不符合预期, 例如去除 HTML 标签:

```
Python
1 text = "Do an internship <b> in place </b> of <b> one </b> course."
2 re.sub(r"<.*>", "", text)  ## 'Do an internship course.' (过度匹配)
```

可以通过在 **repetition specifier** 之后使用 `?` 进行非贪婪匹配:

```
Python
1 re.sub(r"<.*?>", "", text)  ## 'Do an internship in place of one course.'
```

#### ⚠ Remark ▾

- `?` 在此处表示"尽可能少地匹配", 而非"0 或 1 次"
- 也可改用更精确的表达式, 如:

```
Python
1 re.sub(r"<[>]*>", "", text)
```

## 1.4 Python 中的特殊字符

#### 🔗 Logic ▾

在正则表达式中, 特殊字符需转义才能作为字面量使用. Python 中反斜杠 `\` 也用于表示特殊字符 (如 `\n`, `\t`), 因此需特别注意.

### 1.4.1 转义示例

```
Python
1 ## 匹配数字
2 re.search("\\d+", "a93b")  ## <re.Match object; span=(1, 3), match='93'>
3 re.search(r"\d+", "a93b")  ## <re.Match object; span=(1, 3), match='93'>
4
5 ## 使用双反斜杠表示字面反斜杠
6 tmp = "something \\ other\n"
7 re.search("\\\\", tmp)  ## <re.Match object; span=(10, 11), match='\\'>
```

### 1.4.2 使用原始字符串 (raw string)

建议使用 `r"..."` 避免转义混乱:

```
Python
1 print(r"My Windows path is: C:\Users\nadal.") ## 原样输出
2 re.search(r"\\", tmp) ## 匹配一个反斜杠
```

#### ⚠ Remark ▾

原始字符串不处理 Python 转义, 但正则表达式引擎仍会解析其中的转义符.

## | 2 Interacting with the operating system and external code and configuring Python

### | 2.1 与操作系统交互

#### 🔗 Logic ▾

Python 提供了多种与 operating system 交互的方式, 包括调用 shell 命令, 操作文件系统等

#### | 2.1.1 文件系统操作

常用函数:

- `os.getcwd()`: 获取当前工作目录
- `os.chdir()`: 切换目录
- `import`: 导入模块
- `pickle.dump()` / `pickle.load()`: 序列化与反序列化

#### | 2.1.2 调用 shell 命令

使用 `subprocess.run()` 执行 UNIX 命令并捕获输出:

```
Python
1 import subprocess, io
2
3 result = subprocess.run(['ls', '-al'], capture_output=True)
4 print(result.stdout.decode())
```

使用 `io.BytesIO` 处理输出流:

```
Python
1 with io.BytesIO(result.stdout) as stream:
2     content = stream.readlines()
3     print(content[:2])
```

#### | 2.1.3 文件系统查询

检查文件或目录是否存在:

```
Python
1 os.path.exists("unit2-dataTech.qmd") ## True
```

列出目录内容:



## Python

```
1 os.listdir("../data")
2 ## ['hivSequ.csv', 'cpds.csv', 'precipData.txt', 'stackoverflow-2021.db', 'coop.txt.gz',
   'airline.cube']
```

### 2.1.4 跨平台路径处理

使用 `os.path.join` 处理路径差异:



## Python

```
1 os.listdir(os.path.join("../", "data"))
2 ## ['hivSequ.csv', 'cpds.csv', 'precipData.txt', 'stackoverflow-2021.db', 'coop.txt.gz',
   'airline.cube']
```

建议编写与操作系统无关的代码, 使用 `os.path.join` 等函数确保代码在不同平台上都能正常工作

### 2.1.5 系统信息获取

获取操作系统信息:



## Python

```
1 import platform
2 platform.system() ## 'Linux'
3
4 os.uname()
5 ## posix.uname_result(sysname='Linux', nodename='smeagol', release='6.8.0-64-generic',
   version='##67%20')
```

获取 Python 版本信息:



## Python

```
1 platform.python_version() ## '3.13.2'
2
3 sys.version
4 ## '3.13.2 | packaged by conda-forge | (main, Feb 17 2025, 14:40:20) [GCC 13.3.0]'
```

### 2.1.6 环境变量

获取环境变量:



## Python

```
1 os.environ['PATH']
2 ## '/system/linux/miniforge-3.13/bin:/system/linux/miniforge-
   3.13/condabin:/system/linux/miniforge-3.13/bin/'
```

### 2.1.7 Python 脚本作为 shell 脚本

将 Python 脚本转换为可执行的 shell 脚本:

1. 在文本文件中编写 Python 代码, 例如 `example.py`
2. 在文件第一行添加 `#!/usr/bin/python` 或更具可移植性的 `#!/usr/bin/env python`
3. 使用 `chmod` 使文件可执行: `chmod ugo+x example.py`
4. 从命令行运行脚本: `./example.py`

### 2.1.8 命令行参数解析

使用 `argparse` 包处理命令行参数:

```
Python
1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument('-y', '--year', default=2002, help='year to download')
4 parser.add_argument('-m', '--month', default=None, help='month to download')
5 args = parser.parse_args()
6 year = int(args.year)
```

运行方式:

```
Bash
1 ./example.py 2004 January
```

## 2.1.9 执行控制

使用 `Ctrl-C` 中断执行, 这会优雅地退出, 返回到命令未启动的状态 注意当 Python 超过可用内存时可能会有较长的延迟

## 2.2 与外部代码交互

像 R, Python, Julia 这样的脚本语言允许调用 "外部代码", 通常指 C 或 C++ (也包括 Fortran, Java 等其他语言)

在 R 和 Python 等通常比编译代码慢得多的语言中, 调用外部代码特别重要, 而在像 Julia 这样的快速语言中则不那么重要 (Julia 使用即时编译)

### 2.2.1 Python 中的外部代码调用

在 Python 中, 可以直接调用 C 或 C++ 代码, 或使用 `Cython` 与 C 交互

使用 Cython 可以:

- 在提供变量类型定义的情况下, 让 Cython 自动将 Python 代码转换为 C
- 定义可以从 Python 代码调用的 C 函数

### 2.2.2 R 中的外部代码调用

在 R 中, 可以使用 `.Call` 直接调用 C 或 C++ 代码, 或使用 `Rcpp` 包 `Rcpp` 专门设计用于编写感觉有点像 R 代码的 C++ 代码, 并且可以非常容易地在 R 和 C++ 之间传递数据

## 3 Modules and Packages

Logic ▾

- 流行的脚本语言通常都有大量在线可用的 add-on packages, Python 的流行很大程度上归功于 PyPI 和 Conda 上丰富的附加包集合, 这些包提供了 Python 的大部分功能
- 要使用一个 package, 需要先在系统上安装 (使用 `pip install` 或 `conda install`), 然后在每次启动新会话时加载 (使用 `import` 语句)
- 有些 modules 默认随 Python 安装 (如 `os` 和 `re`), 但需要用户在特定的 Python 会话中加载

## 3.1 Modules 模块

### 3.1.1 Module 是什么?

- module 是相关代码的集合, 存储在扩展名为 `.py` 的文件中
- 代码可以包含函数, 类, 变量以及可运行代码
- 要访问 module 中的对象, 需要 `import module`

从 Shell 创建示例模块:

```
Bash
1  cat << EOF > mymod.py
2  x = 7
3  range = 3
4  def myfun(x):
5      print("The arg is: ", str(x), ".", sep = '')
6  EOF
```

使用模块:

```
Python
1  import mymod
2  print(mymod.x)  ## 7
3  mymod.myfun(99)  ## The arg is: 99.
```

### 3.1.2 import 语句

- `import` 语句允许我们访问 module 中的代码
- 它将 module 中的 name of object 与 import scope 中的一个 name 关联起来
- name (references) 到 object 的 mapping 称为 **namespace 命名空间** (会在之后详细讲解)

```
Python
1  del mymod
2  ## Check if `mymod` is in scope.
3  try:
4      mymod.x
5  except Exception as error:
6      print(error)  ## name 'mymod' is not defined
```

理解命名空间:

```
Python
1  y = 3
2  import mymod
3
4  mymod  ## 这是在 current (global) scope 中的模块对象
5  ## <module 'mymod' from '/accounts/vis/paciorek/teaching/243fall25/fall-2025/units/mymod.py'>
6
7  x
8  ## NameError: name 'x' is not defined (不在全局命名空间)
9
10 range  ## 这是 builtin 函数, 不是来自模块
11 ## <class 'range'>
12
13 mymod.x
14 ## 7
15
16 mymod.range
17 ## 3
18
19 dir(mymod)  ## 查看模块中的所有对象
20 ## ['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
    '__package__', '__spec__', 'myfun', 'range', 'x']
```

`y` 和 `mymod` 在 global namespace (全局命名空间) 中, 而 `range` 和 `x` 在 `mymod` 的 module namespace (模块命名空间中)

### 3.1.3 从 module 导入对象

可以使用 `from ... import ...` 语句使 module 中定义的对象直接在 current scope 中访问:

```
Python
1 from mymod import x
2 x  ## 现在是全局命名空间的一部分: 7
3
4 dir()  ## 查看当前命名空间中的所有对象
5 ## ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
   __spec__', 'content', 'files', 'io', 'it', 'm', 'math', 'mymod', 'os', 'pattern',
   'platform', 'r', 're', 'return_group', 'st', 'stream', 'subprocess', 'sys', 'text', 'time',
   'tmp', 'x', 'y']
```

此时它与模块中的对象是不同的:

```
Python
1 mymod.x = 5
2 x = 3
3
4 mymod.x  ## 5
5 x        ## 3
```

#### ⚠ Remark ▾

通常不建议使用 `from` 以这种方式导入对象, 因为这可能引入名称冲突并降低模块性

### 3.1.4 使用别名简化模块名

可以使用 `import ... as ...` 来简化模块名

```
Python
1 import mymod as m
2 m.x  ## 5
```

## 3.2 Packages 包

### 3.2.1 `__init__.py` 文件

Packages 是一个包含一个或多个模块的 directory, 其中有一个名为 `__init__.py` 的文件, 该文件在导入包时被调用, 用于初始化包

创建基本包:

```
Bash
1 mkdir mypkg
2
3 cat << EOF > mypkg/__init__.py
4 ## 使 mymod.py 中的对象可作为 mypkg.foo 使用
5 from .mymod import *
6 print("Welcome to my package.")
7 EOF
```

```

8
9 cat << EOF > mypkg/mymod.py
10 x = 7
11 def myfun(val):
12     print(f"Converting {val} to integer: {int(val)}.")
13 EOF

```

调用包:

```

Python
1 import mypkg ## Welcome to my package.
2 mypkg.x ## 7
3 mypkg.myfun(7.3) ## Converting 7.3 to integer: 7.

```

由于 `__init__.py` 中设置了 import 模块的方式, 我们在调用 package 之后无需知道特定的 module 的位置即可使用它们

#### ⚠ Remark ▾

可以在 `__init__.py` 中设置 `__all__` 来定义 `from ... import *` 所导入的内容

### 3.2.2 internal/private objects 内部/私有对象

可以添加用户不直接使用的 internal module (仅供 main module 使用), 按照约定以下划线 `_` 开头来表示私有/内部函数:

```

Bash
1 cat << EOF > mypkg/auxil.py
2 def _helper(val):
3     return val + 10
4 EOF
5
6 cat << EOF >> mypkg/mymod.py
7 from .auxil import _helper
8 def myfun10(val):
9     print(f"Converting {val} to integer plus 10: {int(_helper(val))}.")
10 EOF

```

### 3.2.3 子包

包可以在嵌套目录中包含模块, 通过这类 **subpackages** 实现额外的模块性:

#### ⚠ Remark: subpackage 的调用方式 ▾

subpackage 一般通过以下两种方式被调用:

- 通过 package 的 main `__init__.py` 在用户使用 package 时自动调用, 即在 `mypkg/__init__.py` 中添加 `from . import mysubpkg`
- 要求用户手动调用, 如 `import mypkg.mysubpkg`

```

Shell
1 mkdir mypkg/mysubpkg
2
3 cat << EOF > mypkg/mysubpkg/__init__.py
4 from .values import *
5 print("Welcome to my package's subpackage.")

```

```
6 EOF
7
8 cat << EOF > mypkg/mysubpkg/values.py
9 x = 999
10 b = 7
11 _my_internal_var = 9
12 EOF
```

使用子包:



Python

```
1 import mypkg.mysubpkg  ## __init__.py 被调用
2 ## Welcome to my package's subpackage.
3
4 mypkg.mysubpkg.b
5 ## 7
6
7 mypkg.x
8 ## 7
```

#### ⚠ Remark ▾

一般来说，我们不会把 **子包 (mysubpkg)** 里的东西，直接“搬运”到 **主包 (mypkg)** 的命名空间里。也就是说，通常用户要用子包的内容时，需要写 `mypkg.mysubpkg.xxx`。

但是，有些情况下会有例外。举个例子：

- 在 **NumPy** 里，函数 `linspace` 实际上是定义在文件 `numpy/core/function_base.py` 里的。
- 可是用户在用的时候，并不需要写 `numpy.core.linspace`，只需要写 `numpy.linspace` 就行。
- 这是因为 NumPy 在它的 `__init__.py` 文件里，提前帮你把 `core` 子包里的 `linspace` 导入到了 `numpy` 的顶层命名空间。

对比一下：

- 有些函数 NumPy 没有帮你“搬运”到顶层，比如线性代数的函数。
- 它们必须通过子包来访问，例如 `numpy.linalg.<函数名>`，而不是直接 `numpy.<函数名>`。

## 3.3 Installing Packages 安装包

### 🔗 Logic ▾

- 如果包在 PyPI 或通过 Conda 可用但不在系统上，可以轻松安装
- 通常不需要机器上的 root 权限来安装包，但可能需要使用 `pip install --user` 或设置新的 Conda 环境
- 包通常依赖于其他包，pip 或 conda 通常会自动安装依赖项
- Conda 的一个优点是它还可以安装 Python 包依赖的非 Python 包，而使用 pip 时有时需要安装一个 system package 来满足依赖项

### 3.3.1 包管理工具

**mamba:**

- `mamba` 是 conda 的直接替代品，通常在依赖项解析方面做得更好
- 在最近的 conda 版本中，可以通过运行 `conda config --set solver libmamba` 在使用 conda 命令时使用 `mamba` 的依赖项解析器

**conda-forge channel:**

通常建议在使用 Conda 安装包时使用 **conda-forge** 通道, conda-forge 提供了由社区维护的各种最新包

### 3.3.2 使包可安装 (optional)

可以配置包以便通过 pip 构建和安装, 需要以下文件:

- `pyproject.toml`: 打包工具使用的配置文件
- `setup.py`: 使用 `setuptools` 时构建和安装包时运行
- `setup.cfg`: 使用 `setuptools` 时提供包的元数据
- `environment.yml`: 提供使用包的完整环境信息
- `LICENSE`: 指定包的许可证

### 3.3.3 可重现性和包管理

#### Logic ▾

为了可重现性, 了解使用的包版本很重要

pip 和 conda 使这变得容易, 可以创建一个 **requirements** 文件来捕获当前使用的包及其版本

使用 **pip**:



Shell

```
1 pip freeze > requirements.txt
2 pip install -r requirements.txt
```

使用 **conda**:



Shell

```
1 conda env export --no-builds > environment.yml
2 conda env create -f environment.yml
```

`--no-builds` flag 确保不包含系统特定的依赖项的信息 (例如那些只在 MacOS 上工作而在 Windows 上不工作的依赖项)

### 3.3.4 Conda 环境

Conda 环境提供了额外的模块化/可重现性层, 允许为计算设置完全可重现的环境:



Shell

```
1 conda create -n myenv python=3.13
2 source activate myenv
3 conda install numpy
```

#### Remark ▾

如果我们使用 `conda activate` 而不是 `source activate`, Conda 会提示我们运行 `conda init`, 这将对我们的 `~/.bashrc` 进行更改, 其中之一是在启动 shell 时自动激活 Conda 基础环境

这可能没什么问题, 但了解这一点很有帮助。

### 3.3.5 Package location

#### Logic ▾

Python 中的包可能安装在文件系统的各个位置, 有时找出包从文件系统的哪个位置加载是很有用的

使用 `__file__` 和 `__version__` 对象查看包在文件系统上的安装位置和版本:

```
Python
1 import numpy as np
2 np.__file__  ## '/system/linux/miniforge-3.13/lib/python3.13/site-packages/numpy/__init__.py'
3 np.__version__  ## '2.1.3'
```

#### ⚠ Remark ▾

- `pip list` 或 `conda list` 也会显示所有包的版本号
- `sys.path` 显示 Python 在系统上查找包的位置

## 3.3.6 源码包与二进制包

### 源码包和二进制包的区别

- 源码包包含原始的 Python 代码 (有时也包含 C/C++ 和 Fortran 代码)
- 二进制包将所有非 Python 代码以二进制格式储存, C/C++ 和 Fortran 代码已经被编译

### 使用源码包和二进制包进行安装的区别

- 如果从源码安装包,
  - C/C++/Fortran 代码将在我们的系统上编译
  - 需要系统上有可用的编译器并且配置正确, 但编译后的代码一般能确保在我们的系统上运行
- 如果从二进制安装包,
  - 不需要在系统上进行编译
  - 某些情况下, 代码可能无法在我们的系统上运行, 因为它是以一种与系统不兼容的方式编译的

#### ⚠ Remark ▾

- Python `wheels` 是 Python 包的二进制包格式
- 某些包的 wheel 会因 operating system 的不同而不同, 以便包能在安装的系统上正确安装

## 4 Types and Data Structures

### 4.1 Types 和 Classes

#### 🔗 Logic ▾

Type 指定了

- 给定信息如何存储
- 可以对信息执行哪些操作

"Primitive" types 是最基本的类型, 通常直接关系到数据在内存或磁盘上的存储方式 (例如 boolean, integer, numeric, character, pointer 等等)

#### 4.1.1 Static vs. Dynamic Typing

##### Static 和 Dynamic Typing 是什么?

- 在像 C 和 C++ 这样的编译语言中, 必须定义每个变量的类型. 因此这些语言是 **statically typed**
- 像 Python 和 R 这样的解释型语言是 **dynamically typed**. 可以在不同时间将不同 type 的信息与给定的变量名关联, 而无需声明变量的 type

```
Python
1 x = 'hello'
2 print(x) ## hello
3 x * 3 ## 'hellohellohello'
4
5 x = 7
6 x * 3 ## 21
7
8 x = {'mykey': 7}
9 x * 3 ## TypeError: unsupported operand type(s) for *: 'dict' and 'int'
```

### Static 和 Dynamic Typing 的优缺点:

- Dynamic typing 有助于快速实现, 可以
  - 避免繁琐的类型定义
  - 避免类型间的微小不一致所导致的问题 (e.g. 整数乘以实数)
- Static typing 有助于软件开发, 可以
  - 防止因 mismatched values 和 unexpected user inputs 所导致的错误
  - 通常更快的执行速度 (因为在运行代码时不需要检查变量类型)

#### ⚠ Remark ▾

Python (和 R) 中更复杂的类型通常使用 references (pointers). 我们将在讨论 memory 时详细研究

## 4.1.2 Python 中的类型

#### 🔗 Logic ▾

Python 中重要的内置数据类型包括列表, 元组和字典, 以及基本标量类型如整数, 浮点数和字符串

查看 Python 和 numpy 中各种内置数据结构的类型.

```
Python
1 x = 3
2 type(x) ## <class 'int'>
3
4 x = 3.0
5 type(x) ## <class 'float'>
6
7 x = 'abc'
8 type(x) ## <class 'str'>
9
10 x = False
11 type(x) ## <class 'bool'>
12
13 x = [3, 3.0, 'abc']
14 type(x) ## <class 'list'>
15
16 import numpy as np
17 x = np.array([3, 5, 7]) ## 整数数组
18 type(x) ## <class 'numpy.ndarray'>
```

```

19 type(x[0])  ## <class 'numpy.int64'>
20
21 x = np.random.normal(size=3)  ## 浮点数数组
22 type(x[0])  ## <class 'numpy.float64'>
23
24 x = np.random.normal(size=(3, 4))  ## 多维数组
25 type(x)  ## <class 'numpy.ndarray'>

```

有时 numpy 可能会修改 data types, 这通常效果很好, 但有时可能需要我们自己控制 types.



#### Python

```

1 x = np.array([3, 5, 7.3])
2 x  ## array([3., 5., 7.3])
3 type(x[0])  ## <class 'numpy.float64'>
4
5 x = np.array([3.0, 5.0, 7.0])  ## 强制使用浮点数
6 type(x[0])  ## <class 'numpy.float64'>
7
8 x = np.array([3, 5, 7], dtype='float64')
9 type(x[0])  ## <class 'numpy.float64'>

```

#### ⚠ Remark ▾

在使用 GPU 时会出现这种情况, 默认通常使用 32 位数字而不是 64 位数字.

### 4.1.3 Composite objects

许多对象可以是 **composite** 的 (例如字典列表或包含列表, 元组和字符串的字典).



#### Python

```

1 mydict = {'a': 3, 'b': 7}
2 mylist = [3, 5, 7]
3 mylist[1] = mydict
4 mylist  ## [3, {'a': 3, 'b': 7}, 7]
5 mydict['a'] = mylist

```

### 4.1.4 Mutable objects 可变对象

Python 中的大多数对象可以 **就地** 修改 (即仅修改对象的某些部分), 但 tuple, strings 和 sets 是 **immutable** 的.



#### Python

```

1 x = (3, 5, 7)
2 try:
3     x[1] = 4
4 except Exception as error:
5     print(error)
6 ## 'tuple' object does not support item assignment
7
8 s = 'abc'
9 s[1]
10 ## 'b'
11
12 try:
13     s[1] = 'y'
14 except Exception as error:
15     print(error)

```

## 4.1.5 Converting between types

我们可以在不同的 basic types 之间进行 **cast (coerce)**

### ⚠ Remark ▾

在 compiled languages 中通常需要显式进行强制转换, 而在像 Python 这样的 interpreted languages 中则较少需要



#### Python

```
1 y = str(x[0])
2 y ## '3'
3
4 y = int(x[0])
5 type(y) ## <class 'int'>
```

一些常见的转换包括:

- 将被解释为字符串的数字转换为实际数字
- 在布尔值和数值之间进行转换

在某些情况下, Python 会以智能的方式 (或偶尔不那么智能的方式) 在后台自动进行转换. 考虑这些隐式强制转换的示例.



#### Python

```
1 x = np.array([False, True, True])
2 x.sum() ## 2
3
4 x = np.random.normal(size=5)
5 try:
6     x[3] = 'hat' ## ValueError
7 except Exception as error:
8     print(error) ## could not convert string to float: 'hat'
9
10 myList = [1, 3, 5, 9, 4, 7]
11
12 myList[2.0]
13 ## TypeError: list indices must be integers or slices, not float
14 myList[2.73]
15 ## TypeError: list indices must be integers or slices, not float
```

R 的严格性较低, 会在某些 Python 不会进行转换的情况下进行转换.



#### R

```
1 x <- rnorm(5)
2 x[2.0]
3 ## [1] -0.4702201
4
5 x[2.73]
6 ## [1] -0.4702201
```

## 4.1.6 Python Object Protocols

有多种 **broad categories** of kinds of objects:

- mapping

- number
- sequence
- iterator

这些称为 **object protocols**, 属于给定类别的所有对象共享关键特征

#### Example ▾

- sequence 对象具有长度和通过索引访问 (有序) 元素的概念
- 迭代器对象具有 "下一个" 和 "停止" 的概念.

#### Remark ▾

Object protocols 基于 Python object model 的 **dunder (双下划线)** 方法实现

## 5 Object-Oriented Programming

### 5.1 OOP Principles

#### 5.1.1 OOP 是什么?

Object-oriented programming (OOP) 在组织代码时需要考虑:

- 储存信息的 objects
- 以特定方式操作这些 objects 的 methods

其中 objects 属于 class, class 的组成部分包括:

- 存储信息的 fields (data)
- 操作 fields 的 methods (functions)

#### 5.1.2 OOP principles

OOP 中的一些标准概念包括 encapsulation (**封装**), inheritance (**继承**), polymorphism (**多态**), 和 abstraction (**抽象**).

##### 1. 封装:

- 防止用户从 object 外部直接访问 object 内的内部数据
- 不过 class 的设计允许用户通过程序员建立的接口访问数据 ('getter' 和 'setter' methods)
- 然而, Python 实际上并不真正强制执行 internal/private information 的概念

##### 2. 继承:

- 允许一个类基于另一个类, 并添加 more specialized features

##### 3. 多态:

- 允许对象或函数根据 context 有不同的行为
- 多态函数根据 input types 表现不同
- 一个例子是有一个名为 "algorithm" 的基类或超类, 以及各种特定的机器学习算法继承自该类, 所有这些类都可能有一个 "predict" method

##### 4. 抽象:

- 涉及隐藏某些操作的细节, 为用户提供输入和获取输出的接口

#### Remark ▾

Class 通常用于有初始化类对象的 **constructors** 和移除对象的 **destructors**

#### 5.1.3 Python 中的 Classes

Python 提供了相当标准的方法来编写面向对象的代码

我们的示例是创建一个用于处理随机时间序列的 class:

- Class 的每个 object 都有控制时间序列随机行为的特定参数值
- 使用特定的 object, 我们可以模拟一个或多个时间序列

```
Python
1 import numpy as np
2
3 class tsSimClass:
4     """
5     时间序列模拟器的 class definition
6     """
7
8     def __init__(self, times, mean=0, cor_param=1, seed=1):
9         """constructor, 在调用 `tsSimClass(...)` 创建类实例时调用"""
10        self.n = len(times)
11        self.mean = mean
12        self.cor_param = cor_param
13
14        ## 私有属性 (封装)
15        self._times = times
16        self._current_U = False
17
18        ## 一些设置步骤
19        self._calc_mats()
20        np.random.seed(seed)
21
22    def __str__(self):
23        """print method"""
24        return f"An object of class `tsSimClass` with {self.n} time points."
25
26    def __len__(self):
27        return self.n
28
29    ## 公共方法: getter 和 setter (封装)
30    def set_times(self, new_times):
31        self._times = new_times
32        self._current_U = False
33        self._calc_mats()
34
35    def get_times(self):
36        return self._times
37
38    ## 主要公共方法
39    def simulate(self):
40        if not self._current_U:
41            self._calc_mats()
42            return self.mean + np.dot(self.U.T, np.random.normal(size=self.n))
43
44    ## 私有方法
45    def _calc_mats(self):
46        """计算相关矩阵和 Cholesky 因子 (缓存)"""
47        lag_mat = np.abs(self._times[:, np.newaxis] - self._times)
48        cor_mat = np.exp(-lag_mat**2 / self.cor_param**2)
49        self.U = np.linalg.cholesky(cor_mat)
```

```
50         print("Done updating correlation matrix and Cholesky factor.")
51         self._current_U = True
```

使用类的示例:



Python

```
1  myts = tsSimClass(np.arange(1, 101), 2, 1)
2  ## Done updating correlation matrix and Cholesky factor.
3
4  print(myts)
5  ## An object of class 'tsSimClass' with 100 time points.
6
7  ## 模拟时间序列
8  y1 = myts.simulate()
```

### 5.1.4 Copies 和 References

在以下示例中,

- `myts_ref` 是 `myts` 的 (浅) 拷贝, 两个 names 指向相同的底层对象, 但是在对 `myts_ref` 进行 assignment 时没有拷贝任何数据
- `myts_full_copy` 是对不同 object 的 reference, `myts` 的所有数据都必须拷贝到 `myts_full_copy`. 这需要额外的内存和时间, 但也更安全



Python

```
1  myts_ref = myts  ## 'myts_ref' 和 'myts' 是同一底层对象的名称
2
3  import copy
4  myts_full_copy = copy.deepcopy(myts)  ## 完整的深拷贝
5
6  ### Now let's change the values of a field.
7  myts.set_times(np.arange(1,1001,10))
8  ## Done updating correlation matrix and Cholesky factor.
9
10 myts.get_times()[0:4]
11 ## array([ 1, 11, 21, 31])
12
13 myts_ref.get_times()[0:4] ## the same as `myts`
14 ## array([ 1, 11, 21, 31])
15
16 myts_full_copy.get_times()[0:4] ## different from `myts`
17 ## array([1, 2, 3, 4])
```

### 5.1.5 Encapsulation 封装

- Private fields 的使用保护它们不被用户修改
- Python 并不真正提供此功能, 但按照约定, 名称以下划线开头的属性被视为私有
- 在模块中, 以下划线开头的对象是私有属性的弱形式. 用户可以访问它们, 但 `from foo import *` 不会导入它们.

### 5.1.6 Inheritance 继承

继承可以是减少代码重复并以逻辑方式组织代码的强大方法



Python

```
1  class Bear:
2      def __init__(self, name, age):
```

```

3         self.name = name
4         self.age = age
5
6     def __str__(self):
7         return f"A bear named '{self.name}' of age {self.age}."
8
9     def color(self):
10        return "unknown"
11
12    class GrizzlyBear(Bear):
13        def __init__(self, name, age, num_people_killed=0):
14            super().__init__(name, age)
15            self.num_people_killed = num_people_killed
16
17        def color(self):
18            return "brown"
19
20    ## 使用示例
21    yog = Bear("Yogi the Bear", 23)
22    print(yog)    ## A bear named 'Yogi the Bear' of age 23.
23    yog.color()  ## unknown
24
25    num399 = GrizzlyBear("Jackson Hole Grizzly 399", 35)
26    print(num399)    ## A bear named 'Jackson Hole Grizzly 399' of age 35.
27    num399.color()  ## brown
28    num399.num_people_killed    ## 0

```

`GrizzlyBear` 类具有超出基类继承的额外字段/方法. Python 先使用特定于 `GrizzlyBear` 类的方法 (如果存在), 然后再回退到 `Bear` 类的方法

#### ⚠ Remark ▾

以上是多态性的一个例子, `GrizzlyBear` 类的实例是多态的, 因为它们可以同时具有 `GrizzlyBear` 和 `Bear` 类的行为: `color` 方法是多态的, 因为它可以用于两个类, 但根据不同的类定义了不同的行为

## 5.2 Attributes

### 5.2.1 Attributes 是什么?

fields 和 methods 都是属性.

### 5.2.2 Class attributes 与 Instance attributes

Class attributes 允许我们操作与 class 的所有 instances 相关的信息

在下例中, `count` 是类属性, 而 `name` 和 `age` 是实例属性

```

Python
1    class Bear:
2        count = 0    ## 类属性
3
4        def __init__(self, name, age):
5            self.name = name    ## 实例属性
6            self.age = age      ## 实例属性
7            Bear.count += 1
8
9    yog = Bear("Yogi the Bear", 23)
10    yog.count    ## 1
11

```

```
12 smokey = Bear("Smokey the Bear", 77)
13 smokey.count ## 2
```

### 5.2.3 添加属性

可以在某些情况下 add instance attributes on the fly

```
Python
1 yog.bizarre = 7
2 yog.bizarre ## 7
3
4 def foo(x):
5     print(x)
6
7 foo.bizarre = 3
8 foo.bizarre ## 3
```

## 5.3 Generic function OOP

### Logic ▾

Generic function (泛型函数) 的使用很方便, 它允许我们使用熟悉的函数处理各种对象, 可以理解为函数层面的多态

### 5.3.1 Generic function 的例子

考虑 Python 中的 `len` 函数. 它似乎神奇地适用于各种对象.

- Python 通过调用 argument 所属的 class 的 `__len__` 方法来实现 `len` 函数.
- `__len__` 是一个 **Double-UNDERscore 双下划线方法**.

类似的情况也发生在运算符上:

```
Python
1 x = 3
2 x + 5 ## 8
3 x.__add__(5) ## 8
4
5 x = 'abc'
6 x + 'xyz' ## 'abcxyz'
7 x.__add__('xyz') ## 'abcxyz'
```

### 5.3.2 为什么使用泛型函数

Python 开发人员本可以将 `len` 编写为具有一堆 if 语句的常规函数, 以便处理不同类型的输入对象. 但这有一些缺点:

1. 需要编写进行 checking 的代码
2. 所有不同情况的代码都存在于一个可能很长的函数中
3. 最重要的是, `len` 仅适用于现有类

## 5.4 dunder methods

### 5.4.1 常见的 dunder methods

**dunder (双下划线)** 方法是一种特殊方法, 在以下情况中, Python 将调用这些方法

- 在类的实例上调用某些函数
- 调用其他标准操作

一些重要的双下划线方法:

- `__init__`: 构造函数
- `__len__`: 被 `len()` 调用
- `__str__`: 被 `print()` 调用
- `__repr__`: 在调用对象名称时调用
- `__call__`: 如果实例作为函数调用调用
- `__add__`: 被 `+` 运算符调用
- `__getitem__`: 被 `[]` 切片运算符调用

## 5.4.2 dunder methods 示例

```
Python
1 class Bear:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6     def __str__(self):
7         return f"A bear named {self.name} of age {self.age}."
8
9     def __repr__(self):
10        return f"Bear(name={self.name}, age={self.age})"
11
12    def __add__(self, value):
13        self.age += value
14        return None
15
16 yogi = Bear("Yogi the Bear", 23)
17 print(yogi)    ## 调用 __str__: A bear named Yogi the Bear of age 23.
18 yogi          ## 调用 __repr__: Bear(name=Yogi the Bear, age=23)
19 yogi + 12     ## 调用 __add__
20 print(yogi)   ## A bear named Yogi the Bear of age 35.
```

## 5.4.3 Python object protocols: 示例 1 - iterators

支持 iteration 的 container class 应提供 `__iter__` 和 `__next__` 方法来实现迭代器协议.

```
Python
1 mytuple = ("apple", "banana", "cherry")
2 for item in mytuple:
3     print(item)
4
5 ## 手动创建迭代器
6 myit = iter(mytuple) ## 等价于 mytuple.__iter__()
7 type(myit) ## <class 'tuple_iterator'>
8
9 print(next(myit)) ## apple
10 print(next(myit)) ## banana
11 myit.__next__() ## cherry, 等价于 next(myit)
```

### ⚠ Remark ▾

Tuple 是 iterable container, 但它们本身不是 iterator

## 5.4.4 Python 对象协议: 示例 2 - 使用 with 的上下文管理器

在 Python 中读写文件的标准方式使用 `with`：

```
Python
1 with open('myfile.txt', 'r') as file:
2     lines = file.readlines()
```

这创建了一个 "context manager", 等效于:

```
Python
1 file = open('myfile.txt', 'r')
2 try:
3     lines = file.readlines()
4 finally:
5     file.close()
```

通过对 class 提供 `__enter__` 和 `__exit__` 方法, 可以使用 `with` 实现 context manager protocol

```
Python
1 import time
2
3 class MyTimer(object):
4     def __enter__(self):
5         print(f"Starting at {time.ctime()}.")
6
7     def __exit__(self, exception_type, exception_value, traceback):
8         print(f"Ending at {time.ctime()}.")
9
10 with MyTimer():
11     x = np.random.normal(size=50000000)
12     del x
13
14 ## Starting at Mon Sep 29 08:34:10 2025.
15 ## Ending at Mon Sep 29 08:34:12 2025.
```

## 6 Functional Programming

### 6.1 函数式编程

#### 6.1.1 函数式编程概述

- **定义与目标**
  - 一种强调 **模块化、可复用、无副作用** 的编程范式：函数只依赖传入的参数，不使用全局变量，且尽量不产生 side effects。
  - 函数被视为 **first-class citizens**：可以作为参数传递、作为返回值、赋给变量使用。
- **匿名函数**
  - 在 Python 与 R 中可使用 **anonymous functions / lambda functions** 按需即时创建，用于临时的函数式操作场景。
- **在 Python 中进行 functional programming**
  - 通过编写 **自包含的函数** 而非类，利用函数的一等公民特性实现组合与复用。
  - **不完全符合的地方**：
    - Python 的 **pass-by-reference** 行为可能导致副作用：若在函数内修改 **mutable** 实参（如 list、numpy array），调用方对象会被更改；**immutable**（如 tuple）不受影响。
    - 一些操作以 **statements** 形式出现而不是函数调用（例如 `import`、`def`），与“万物皆函数”的理念不完全一致。
- **与 R 的对比**

- R 函数更接近 **pass-by-value** 语义，更符合“纯”函数式风格；从范式一致性上更贴近 functional programming 的原则。

## 6.1.2 无副作用原则

### • 核心思想

- 函数式编程追求“**无副作用 (no side effects)**”：函数只依赖输入参数并返回输出，而不改变程序外部状态（如全局变量或外部环境）。
- 这样每个函数都可以被看作一个“**黑箱 (black box)**”，用户无需关心函数内部实现，也不必担心它会修改已有变量。
- 程序运行结果可被视为数学意义上的函数复合（function composition）。

### • R 的实现方式

- R 中的大多数函数（以及理想的自定义函数）都遵循“输入 → 输出”的模式，不影响外部环境。
- R 采用 **pass-by-value (按值传递)**：
  - 当对象被作为参数传入函数时，会在函数作用域中创建一个**局部副本**。
  - 在函数中修改该副本不会影响外部的原始对象。
- 示例：

```
 R
1 x <- 1:3
2 myfun <- function(x) {
3   x[2] <- 7
4   print(x)
5   return(x)
6 }
7 new_x <- myfun(x) ## [1] 1 7 3
8 x ## [1] 1 2 3 (未被修改)
```


- 因此，R 的默认行为天然支持“无副作用”的原则。

### • Python 的对比

- Python 使用 **pass-by-reference (按引用传递)**：
  - 当可变对象（如 list、numpy array）作为参数传入函数时，函数中对该对象的修改会**直接影响原对象**。
- 示例：

```
 Python
1 x = np.array([1,2,3])
2 def myfun(x):
3     x[1] = 7
4     return x
5
6 new_x = myfun(x)
7 x ## array([1, 7, 3]) ← 被修改
```

- 若函数不返回结果而直接修改对象（如下例），副作用更加明显：

```
 Python
1 def myfun(x):
2     x[1] = 7
3     return None
4 myfun(x)
```

```
5 x ## array([1, 7, 3])
```

- 因此，在 Python 中**非常容易违反无副作用原则**。  
若要避免，应在函数内**显式复制对象**：



Python

```
1 def myfun(x):
2     y = x.copy()
3     y[1] = 7
4     return y
```

### • 总结：R 与 Python 的差异

特性	R	Python
参数传递方式	Pass-by-value	Pass-by-reference
默认是否易产生副作用	否	是（尤其是 mutable 对象）
一致性与可预测性	高	需谨慎设计函数以避免意外修改

### • 例外情况

- 即使在 R 中，也存在必要的副作用函数（出于语言或交互性需求）：
  - `par()`：修改图形参数
  - `library()`：加载包并改变环境状态
  - `plot()`：绘制图形（产生视觉输出）
- 这些副作用是受控且必要的，不影响函数式编程整体的可理解性。

## 6.1.3 函数是 first-class objects

- 在 Python 中，**一切皆对象（Everything is an object）**
  - 在 Python 中，**函数、类、数值、字符串等**都被视为对象。
  - 函数可以像普通变量一样被赋值、传递、或作为返回值，这体现了函数的“**一等公民（first-class citizens）**”地位。

### • 名称与对象的关系

- 变量名只是对内存中对象的**引用（reference）**。
- 当执行赋值语句时（例如 `x = 3`），Python 会将名称 `x` 绑定到一个对象（此处为整数 3）。
- 可以通过 `type(x)` 查看对象类型。

### • 函数的赋值与重绑定



Python

```
1 x = 3
2 x([1,3,5]) ## ❌ 报错: 'int' object is not callable
3 x = sum     ## 将内建函数 sum 赋给 x
4 x([1,3,5]) ## ✅ 输出 9
5 type(x)    ## <class 'builtin_function_or_method'>
```

- 说明：变量名 `x` 可以从指向整数变为指向函数对象。
- 因此函数可以像数据一样被动态操作、重绑定。

- **按名称动态调用函数**

- 可以使用 `getattr()` 根据函数名字符串从模块中获取函数对象：

```
Python
1 function = getattr(np, "mean")
2 function(np.array([1,2,3]))  ## 输出 np.float64(2.0)
```

- 这允许基于 **字符串函数名** 进行灵活的函数调用（如反射机制）。

- **函数作为参数（高阶函数）**

- 函数可以被传入另一个函数作为参数或返回值：

```
Python
1 def apply_fun(fun, a):
2     return fun(a)
3
4 apply_fun(round, 3.5)  ## 输出 4
```

- 这种 **接收或返回函数对象** 的函数被称为 **higher-order functions（高阶函数）**。
- 许多内建函数（如 `map`, `filter`, `reduce`）正是此类高阶函数。

## 6.1.4 哪些操作是 function calls?

- **哪些操作是函数调用？**

- Python 中一些语句（statements）**不是函数调用**，但会影响当前环境：
  - `import`：导入模块或包
  - `def`：定义函数或类
  - `return`：从函数返回结果
  - `del`：删除对象
- 虽然语法上不是函数调用，但它们本质上执行类似的动作。

- **运算符与面向对象函数调用（OOP）**

- 运算符在底层对应类方法调用。例如：

```
Python
1 x = np.array([0,1,2])
2 x - 1          ## array([-1, 0, 1])
3 x.__sub__(1)   ## 与上式等价
4 x             ## array([0, 1, 2])（原对象未被修改）
```

- 运算符 `+`, `-`, `*`, `/` 等实质上是对对象方法（如 `__add__`, `__sub__`）的封装调用。
- 这体现了 **操作符重载（operator overloading）** 与 **泛型函数式（generic function OOP）** 的思想。

- 语法糖与解释器行为

- 表面上可写为 `return(x)` 或 `del(x)`，但解释器实际上将其解析为：
  - `return x`
  - `del x`
- 圆括号仅是语法允许的“装饰性符号”，并不改变解释器的底层解析逻辑。


## 6.1.5 Map Operations（映射操作）

- 定义与作用

- 映射（map）操作：接受一个函数，并将该函数依次作用于集合中每个元素。
- 类似数学中的“函数映射”概念，是函数式编程中最常用的操作之一。
- 优点：代码简洁、可读性高、避免显式循环。


- Python 中的映射类型函数

- Python 提供多种 map 类函数：
  - 内置函数 `map()`
  - `pandas.DataFrame.apply()`
- 它们都是 高阶函数（higher-order functions），即以函数作为参数。
- 列表推导式（list comprehension）也是映射操作的一种形式：


```
 Python
1 x = [1, 2, 3]
2 y = [pow(val, 2) for val in x]
3 ## [1, 4, 9]
```

- 基本用法： `map()`

- `map()` 会在可迭代对象（iterable）的所有元素上运行指定函数。
- 可迭代对象包括 `list`，`range()` 以及其他返回可迭代结构的函数。

```
 Python
1 x = [1.0, -2.7, 3.5, -5.1]
2 list(map(abs, x))
3 ## [1.0, 2.7, 3.5, 5.1]
```

- `map()` 还支持 多个可迭代对象：

```
 Python
1 list(map(pow, x, [2, 2, 2, 2]))
2 ## [1.0, 7.29, 12.25, 26.01]
```

- 使用匿名函数（lambda）

- 可通过 `lambda` 函数 即时定义一个临时函数，无需命名：

```
 Python
1 x = [1.0, -2.7, 3.5, -5.1]
2 result = list(map(lambda val: val * 2, x))
3 ## [2.0, -5.4, 7.0, -10.2]
```

- **lambda 函数** 是一种 **匿名函数 (anonymous function)**，用于一次性的小操作。

- 使用 **functools.partial** 预设参数

- 如果要为函数预设部分参数，可用 **partial()** 创建“半固定函数”：

```
Python
1 from functools import partial
2 round3 = partial(round, ndigits=3)
3 list(map(round3, [32.134234, 7.1, 343.7775]))
4 ## [32.134, 7.1, 343.777]
```

- 与 for 循环的对比

- **Pandas 的 apply 方法** 支持将函数直接映射到分组数据上：

```
Python
1 ## Stratified analysis 示例
2 subsets = df.groupby('grouping_variable')
3
4 ## map风格 (apply): 简洁且易读
5 results = subsets.apply(analysis_function)
6
7 ## for 循环写法: 更冗长
8 results = []
9 for _, subset in subsets:
10     results.append(analysis_function(subset))
```

- 相比之下，**apply()** 风格更加函数式，简洁且更符合数据分析流程。

- 延伸：**MapReduce 框架**

- 映射操作是 **MapReduce** 的核心机制：
  - “Map” 阶段：并行地将函数作用于数据块。
  - “Reduce” 阶段：聚合映射结果。
- 被广泛应用于 **Hadoop** 和 **Spark** 等大数据平台中，用于分布式数据处理。

## 6.2 Function evaluation, frames, and the call stack (函数求值, 帧和调用栈)

### 6.2.1 概述

- 概览 (Overview)

- 在程序执行过程中，函数往往会在其他函数内部被调用，从而形成一个**嵌套调用链**。
- 这些函数调用按顺序被压入和弹出，组成了 **调用栈 (call stack)**。
- 可以将调用栈想象为“**食堂托盘堆**”：
  - 当一个函数被调用时，它被**压入栈顶 (push)**；
  - 当该函数执行完毕时，它被**弹出栈 (pop)**。

- 调用栈在调试中的作用

- 理解调用栈对于阅读错误信息和调试至关重要。
- 在 Python 中，当错误发生时，会显示完整的 **调用栈追踪 (traceback)**，能帮助我们定位错误来源。

- 优点：提供了完整的函数调用历史，有助于理解错误的上下文。
- 缺点：输出可能非常冗长，难以快速阅读。
- 对比 R：
  - R 默认只显示错误发生的函数。
  - 若需查看完整调用链，可使用 `traceback()`。

## • Python 函数求值 (Function Evaluation) 流程

### 1. 参数求值与匹配

- 用户传入的实参会先在 **调用作用域 (calling scope)** 中求值。
- 计算结果与函数定义中的形参名进行 **匹配绑定**。

### 2. 创建新帧 (Frame) 与命名空间 (Namespace)

- 调用函数时，Python 会为该调用创建一个新的 **执行帧 (frame)**，并分配一个独立的 **局部命名空间 (local namespace)**。
- 该帧会被 **压入调用栈 (push onto the stack)**。
- 函数的参数（包括默认参数）会在此命名空间中赋值。

### 3. 函数体求值 (Evaluation in local scope)

- 函数体的执行发生在 **局部作用域** 中。
- 若在局部找不到某个变量名，Python 会根据 **词法作用域 (lexical scoping)** 规则依次查找：
  1. 当前局部作用域 (local scope)
  2. 外层嵌套函数作用域 (enclosing scopes, if any)
  3. 全局作用域 (global/module scope)
  4. 内建作用域 (built-in scope)

### 4. 函数返回与栈清理

- 函数执行结束后，`return` 的值会被传回调用点所在的作用域。
- 当前帧被 **弹出调用栈 (popped off the stack)**。
- 若该命名空间不是其他活动作用域的外层，则会被销毁（释放内存）。

## 6.2.2 Frames and the call stack (帧和调用栈)

### • 基本概念

- Python 会持续追踪当前的 **调用栈 (call stack)**。
- 每次函数调用时，系统会创建一个 **帧 (frame)**，其中包含该函数调用的 **局部命名空间 (local namespace)**，即当前函数中的局部变量。
- 这些帧按调用顺序被压入栈中 (push)，当函数结束时再被弹出 (pop)。

### • 访问与查看调用栈

- Python 提供多种方法来 **查询栈中的帧信息** 并访问其中的对象。
- 借助 `traceback` 模块，可以直接查看当前调用栈的完整结构。
- 常用函数：
  - `traceback.print_stack()`：打印当前调用栈。
  - `traceback.format_stack()`：返回调用栈信息的字符串列表，适合程序化处理。

### • 示例：使用 `traceback` 打印调用栈



Python

```
1 import traceback
```

```

2
3 def function_a():
4     function_b() ## line 2 within function, line 4 overall
5
6 def function_b():
7     ## some comment
8     ## another comment
9     function_c() ## line 4 within function, line 9 overall
10
11 def function_c():
12     traceback.print_stack() ## line 2 within, line 12 overall
13     ## raise RuntimeError("A fake error")
14
15 function_a() ## line 1 relative to the call, line 15 overall

```

#### 执行结果（交互模式）

```

1 File "<stdin>", line 1, in <module>
2 File "<stdin>", line 2, in function_a
3 File "<stdin>", line 4, in function_b
4 File "<stdin>", line 2, in function_c

```

- 说明：
  - 栈顶是当前执行的函数 `function_c`。
  - 每一层函数的调用顺序都被完整记录。

#### • 在脚本中运行（例如 `trace.py`）

```

1 File "file.py", line 15, in <module>
2     function_a()
3 File "file.py", line 4, in function_a
4     function_b()
5 File "file.py", line 9, in function_b
6     function_c()
7 File "file.py", line 12, in function_c
8     traceback.print_stack()

```

- 此时行号对应的是脚本中的绝对位置（相对于整个文件）。
- 可以清晰看到调用链的层次结构。

#### • 错误追踪（Traceback 与异常）

- 若将 `traceback.print_stack()` 注释掉，并改为：

```

Python
1 raise RuntimeError("A fake error")

```

- 输出结果会显示**相同的调用栈信息**。
- 这正是 Python 在抛出异常时显示的 **traceback**，即从错误发生处向上追溯至最初调用点的完整路径。

## 6.3 函数输入和输出

### 6.3.1 Arguments（函数输入）

#### • 查看函数参数与默认值

- 可使用 **help 系统** 或 `?function_name`（适用于 R 语言）查看函数的参数信息与默认值。
- 示例：

```

Python
1 def add(x, y, z=1, absol=False):
2     if absol:
3         return abs(x + y + z)
4     else:
5         return x + y + z

```

- 参数定义规则

- 定义函数时，无默认值参数必须位于默认值参数之前。
- 示例调用：

```

Python
1 add(3, 5)                ## 9
2 add(3, 5, 7)             ## 15
3 add(3, 5, absol=True, z=-5) ## 3
4 add(z=-5, x=3, y=5)      ## 3

```

- 若缺少必要位置参数会报错：

```

Python
1 add(3)
2 ## TypeError: add() missing 1 required positional argument: 'y'

```

- 位置参数与关键字参数

- 位置参数 (positional arguments)：按顺序提供。
- 关键字参数 (keyword arguments)：使用 `name=value` 形式指定。
- 规则：位置参数必须在关键字参数之前。

```

Python
1 add(z=-5, 3, 5)
2 ## SyntaxError: positional argument follows keyword argument

```

- 可变数量的参数：\*args 与 \*kwargs\*\*

- 使用 \*args 表示不定数量的位置参数。
  - args 通常是一个 tuple，可以被遍历或求和。

```

Python
1 def sum_args(*args):
2     print(args[2])    ## 第三个参数
3     total = sum(args)
4     return total
5
6 result = sum_args(1, 2, 3, 4, 5)
7 ## 输出:
8 ## 3
9 ## 15

```

- 应用示例：os.path.join 可接收任意数量参数或解包列表：

```

Python
1 os.path.join('a', 'b', 'c')  ## 'a/b/c'
2 x = ['a', 'b', 'c']
3 os.path.join(*x)             ## 'a/b/c'

```

- 若要同时接收关键字参数，需定义：

```

Python
1 def func(*args, **kwargs):
2     ...

```

- `args` 是 tuple, `kwargs` 是 dict。

## 6.3.2 Function Outputs (函数输出)

### 基本返回语句

- `return x` 用于指定函数输出，可在函数中任意位置出现。
- 当 `return` 执行时，函数立即退出。

### 多个返回值

- 可返回多个对象，返回结果会被打包成 **tuple**。

```

Python
1 def f(x):
2     if x < 0:
3         return -x**2
4     else:
5         res = x^2  ## 注意：这是按位异或，不是幂运算
6         return x, res
7
8 f(-3)  ## -9
9 f(3)   ## (3, 1)
10
11 out1, out2 = f(3)

```

- 可通过解包 (unpacking) 将多个输出赋值给不同变量。

### 无返回值函数

- 若函数主要用于副作用 (如打印、修改外部状态等)：
  - 可省略 `return`；
  - 或显式写作 `return None`；
  - 或仅写 `return` (效果相同)。

## 6.4 Pass by Value vs. Pass by Reference

### 6.4.1 Core Concepts

#### Pass-by-value (值传递)

- 函数调用时，实参会被复制一份到函数内部。
- 函数中对参数的修改 **不会影响** 原始对象。
- 特点：

- 模块化、无副作用 (side effects)；
- 但可能占用更多内存、计算效率低。
- **Pass-by-reference (引用传递)**
  - 传入函数的不是副本，而是 **指向原对象的引用**。
  - 函数内部修改对象内容时，会直接影响函数外部的状态。
  - 特点：
    - 高效（避免复制大型对象）；
    - 但更“危险”，可能引入隐藏的副作用。

#### 🔗 Logic ▾

Pass-by-value 更符合函数式编程 (functional programming) 的思想：  
函数的效果仅体现在返回值，而非对外部状态的修改。

## 6.4.2 Python 的行为

- Python 中，**数组与其他可变对象 (mutable types)** 采用 **引用传递 (pass-by-reference)**。
- 不可变对象 (immutable types)，如 **tuple**、**int**、**str**，即便传引用，也不会被修改。



Python

```
1 def myfun(x):
2     x[1] = 99
3
4 y = [0, 1, 2]
5 z = myfun(y)
6 type(z)    ## <class 'NoneType'>
7 y          ## [0, 99, 2]
```

#### ⚠ Remark ▾

尽管 Python 并非严格意义上的“pass-by-reference”，  
但由于可变对象是共享引用的，因此表现上等价于引用传递。

## 6.4.3 示例分析：哪些修改会影响外部状态？



Python

```
1 def myfun(f_scalar, f_x, f_x_new, f_x_newid, f_x_copy):
2     print(id(f_scalar))
3     print(id(f_x))
4
5     f_scalar = 99          ## 不影响外部 scalar
6     f_x[0] = 99           ## 修改外部 x
7     f_x_new = [99, 2, 3]  ## 不影响外部 x_new
8     y = f_x_newid
9     y[0] = 99             ## 修改外部 x_newid
10    z = f_x_copy.copy()
11    z[0] = 99              ## 不影响外部 x_copy
```



Python

```

1 scalar = 1
2 x = [1,2,3]
3 x_new = [1,2,3]
4 x_newid = [1,2,3]
5 x_copy = [1,2,3]
6
7 myfun(scalar, x, x_new, x_newid, x_copy)

```

- 状态保留（未修改）

```

1 scalar = 1
2 x_new = [1, 2, 3]
3 x_copy = [1, 2, 3]

```

- 状态被修改

```

1 x = [99, 2, 3]
2 x_newid = [99, 2, 3]

```

#### Logic ▾

若在函数中**重新赋值（rebind）**一个变量名，则创建的是新的局部引用，不会修改外部状态。  
若仅**修改对象内容（mutate）**，则会影响外部对象。

## 6.4.4 Mutable 对象的通性

- 与 `list` 类似，**NumPy 数组**等可变对象在函数中被修改时，外部状态同样会受到影响。

## 6.4.5 Pointers（指针, optional）

#### Logic ▾

理解“引用传递”的底层机制，可从 C 语言中的 **指针（pointer）** 理解。

在 C 中：

#### C c

```

1 int x = 3;
2 int* ptr;
3 ptr = &x; // 获取 x 的地址
4 *ptr * 7; // 解引用，返回 21

```

- `int* ptr`：声明 `ptr` 为指向整数的指针；
- `&x`：获取 `x` 的内存地址；
- `*ptr`：通过指针访问该地址的内容。

数组本质上也是指针：

#### C c

```

1 int x[10]; // x 实际上是指向数组首元素的指针

```

可以通过 `x[0]` 或 `*x` 访问首元素。

## 函数中使用指针的效果

C

c

```
1  int myCal(int* ptr){
2      *ptr = *ptr + *ptr;    // 原地修改对象
3  }
4
5  myCal(&x);    // 传入 x 的地址, x 会被修改
```

### ⚠ Remark ▾

在 C 中, 函数接收的是 **地址 (指针)**, 因此函数内部可直接修改外部对象的值。

Python 的行为与此类似:

它通过**对象引用 (object reference)** 传递参数,  
从而在函数中可以“间接地”修改原对象 (尤其是 mutable 类型)。

## 6.5 Namespaces 和 Scopes

### 6.5.1 Namespaces

**Namespace (命名空间)** 是从 names 到 objects 的 mapping, 允许 Python 通过 name 找到 object

Namespace 在执行 Python 代码的过程中创建和移除:

- 当函数运行时, 会为函数中的 local variables 创建一个 namespace
- 在函数执行完成时删除 namespace
- 每个的函数调用有单独的 namespace

### 6.5.2 Scope

**Scope (作用域)** 决定了代码中的某个位置可以访问哪些 namespaces

- Scope 是嵌套的
- 决定 Python 以何种顺序搜索各种 namespaces 以查找对象

### 6.5.3 Key Scopes

关键作用域, 按搜索 namespaces 的顺序排序 (LEGB):

- **Local scope 局部作用域**: given function/class method 内可用的 objects
- **Non-local (Enclosing) scope 非局部作用域**: 包含 given function 的函数中的可用的 objects
- **Global (Module) scope 全局作用域**: 定义 given function 的 module 中可用的 objects

### ⚠ Remark ▾

若代码不是在函数内部执行, global scope 将可以被当作 local scope

- **Built-ins scope 内置作用域**: 通过 Python 提供的 built-ins module 中可用的 objects, 可从任何地方访问

### ⚠ Remark ▾

- `import` 将导入 module 的 name 添加到 current (local) scope 的 namespace
- 可以使用 `locals()` 和 `globals()` 查看局部和全局命名空间。

>-

Shell

```
1 cat local.py
```

```
1 gx = 7
2
3 def myfun(z):
4     y = z*3
5     print("local: ", locals())
6     print("global: ", globals())
```



Python

```
1 import local
2
3 gx = 99
4 local.myfun(3)
```

```
1 local: {'z': 3, 'y': 9}
2 global: {'__name__': 'local', '__doc__': None, '__package__': '', '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x1038e3d30>, ..., 'gx': 7, 'myfun':
<function myfun at 0x103903010>}
```

## 6.5.4 Lexical scoping 和 enclosing scopes

Enclosing scope 是定义函数的 scope, 而不是调用函数的 scope

一旦搜索了 enclosing scope, 如果找不到对象名称, 那么 Python 会在定义函数的 global/module scope 中查找, 而不是从调用它的地方查找

这种方法称为 lexical scoping. R 和许多其他语言也使用 lexical scoping

### 6.5.4.1 词法作用域示例

案例 1:



Python

```
1 x = 3
2 def f2():
3     print(x)
4 def f():
5     x = 7
6     f2()
7 f() ## 输出 3
```

案例 2:



Python

```
1 x = 3
2 def f2():
3     print(x)
4 def f():
5     x = 7
6     f2()
7 x = 100
8 f() ## 输出 100
```

案例 3:

```
Python
1 x = 3
2 def f():
3     def f2():
4         print(x)
5     x = 7
6     f2()
7 x = 100
8 f() ## 输出 7
```

案例 4:

```
Python
1 x = 3
2 def f():
3     def f2():
4         print(x)
5     f2()
6 x = 100
7 f() ## 输出 100
```

案例 5:

```
Python
1 y = 100
2 def fun_constructor():
3     y = 10
4     def g(x):
5         return x + y
6     return g
7
8 ### fun_constructor() creates functions
9 myfun = fun_constructor()
10 myfun(3) ## 输出 13
```

1. What is the enclosing scope for the function `g()` ?  
The local scope of `fun_constructor()`
2. Which `y` does `g()` use?  
10
3. Where is `myfun` defined (this is tricky – how does `myfun` relate to `g`)?  
`myfun` is simply a reference to `g`
4. What is the enclosing scope for `myfun()` ?  
The enclosing scope of `myfun()` is still the local scope of `fun_constructor()` where `g` was created.
5. When `fun_constructor()` finishes, does its scope (and namespace) disappear? What would happen if it did?  
Normally the scope would disappear, but because `g` is a closure, Python preserves the needed variables ( `y = 10` ). If the scope vanished completely, calling `myfun()` would raise a `NameError` .
6. What does `myfun` use for `y` ?  
`myfun` uses the closed-over `y = 10` .

## 6.5.5 Global 和 non-local variables

- 可以使用 `global` 创建和修改 global variable 中的变量
- 可以使用 `nonlocal` 创建和修改 enclosing scope 中的变量

```
Python
```

```

1  del x
2  def myfun():
3      global x
4      x = 7
5
6  myfun()
7  print(x)  ## 7
8
9  x = 9
10 myfun()
11 print(x)  ## 7
12
13 def outer_function():
14     x = 10
15     def inner_function():
16         nonlocal x
17         x = 20
18     print(x)  ## 10
19     inner_function()
20     print(x)  ## 20
21
22 outer_function()

```

#### ⚠ Remark ▾

在 R 中, 可以使用全局赋值运算符 `<--` 做类似的事情

## 6.5.6 Closures 闭包

### 🔗 Logic ▾

将 data 与 function 绑定的一种方法是使用 **closure**. 这是一种 functional programming 的方式, 可以实现类似于 OOP 中的 class 的东西

#### Closure 的实现方法:

1. 在函数调用中创建一个或多个函数, 并将函数作为输出返回
2. 当执行原始函数时, 新函数被创建并返回, 然后可以调用该函数
3. 该函数可以访问 enclosing scope 中的 objects

#### 使用 Closure 的优点:

区别于使用 global variable, closure 中的 data 与 functions 被绑定在一起, 并受到保护, 不会被用户更改

```

Python
1  x = np.random.normal(size=5)
2  def scaler_constructor(data):
3      def g(param):
4          return param * data
5      return g
6
7  scaler = scaler_constructor(x)
8  del x  ## 演示我们不再需要 x
9
10 scaler(3)
11 ## array([-4.5020648 , -3.25168752, -4.05046623, 0.66985868, 3.50796174])
12

```

```
13 scaler(6)
14 ## array([-9.0041296 , -6.50337504, -8.10093246, 1.33971736, 7.01592347])
```

#### ⚠ Remark ▾

调用 `scaler(3)` 会将 3 乘以存储在函数 `scaler` 的 closure (the namespace of the enclosing scope) 中的 data 值

这是一个更实际的例子:

```
Python
1 def make_container(n):
2     x = np.zeros(n)
3     i = 0
4     def store(value=None):
5         nonlocal x, i
6         if value is None:
7             return x
8         else:
9             x[i] = value
10            i += 1
11    return store
12
13 nboot = 20
14 bootmeans = make_container(nboot)
15
16 import pandas as pd
17 iris = pd.read_csv('https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/io/data/csv/iris.csv')
18 data = iris['SepalLength']
19
20 for i in range(nboot):
21     bootmeans(np.mean(np.random.choice(data, size=len(data), replace=True)))
22
23 bootmeans()
24
25 bootmeans.__closure__
```

Closures 也被用作 "function factories" (外侧的 generator function 被称为 "factory function"), 用来轻松生成一组相关函数, 下面是一个例子:

```
Python
1 def number_formatter(notation='US'):
2     """
3     Creates a closure for formatting decimal numbers.
4
5     Args:
6         notation (str): 'US' for US notation (commas for thousands, period for decimal)
7                        'EU' for European notation (periods for thousands, comma for decimal)
8
9     Returns:
10        function: A closure that formats numbers according to the specified notation
11    """
12    def format_number(number):
13        """ GitHub Copilot suggested the `number:,` syntax and the string replace approach. """
14        result = f"{number:,}"
15        if notation == 'US':
```

```

16         ## US notation: 1,234.56
17         return result
18     elif notation == 'EU':
19         ## European notation: 1.234,56
20         ## Swap commas and periods
21         return result.replace(',', 'TEMP').replace('.', ',').replace('TEMP', '.')
22     else:
23         raise ValueError("Notation must be 'US' or 'EU'")
24
25     return format_number
26
27 us_printer = number_formatter('US')
28 eu_printer = number_formatter('EU')
29
30 us_printer(1234.56)    ## '1,234.56'
31 eu_printer(1234.56)    ## '1.234,56'

```

## 6.6 Decorators

- 基本概念

- **Decorator（装饰器）** 是一种 **函数包装器（wrapper）**：它在 **不修改原函数代码** 的前提下，**扩展函数的功能**。
- 核心思想：在函数调用的“前后”自动执行某些逻辑（如打印、计时、检查权限、缓存等）。
- 装饰器的底层是 **高阶函数（higher-order function）** —— 接收函数作为参数并返回新函数。

- 手动创建一个简单装饰器

```

Python
1 def verbosity_wrapper(myfun):
2     def wrapper(*args, **kwargs):
3         print(f"Starting {myfun.__name__}.")
4         output = myfun(*args, **kwargs)
5         print(f"Finishing {myfun.__name__}.")
6         return output
7     return wrapper
8
9 verbose_rnorm = verbosity_wrapper(np.random.normal)
10 x = verbose_rnorm(size=5)
11 x

```

输出：

```

1 Starting normal.
2 Finishing normal.
3 array([ 1.0741,  0.2012, -1.0965, -1.9303, -2.1164])

```

### Logic ▾

`verbosity_wrapper` 接收函数 `myfun` 并返回一个新的函数 `wrapper`。

在调用时：

1. 执行额外逻辑（打印信息）；
2. 调用原始函数；
3. 返回原函数结果。

这种“函数返回函数”的结构依赖于 **闭包（closure）**，使得 `wrapper` 能访问 `myfun`。

- 使用语法糖 @ 语法简化装饰器

```
Python
1 @verbosity_wrapper
2 def myfun(x):
3     return x
4
5 y = myfun(7)
```

输出:

```
1 Starting myfun.
2 Finishing myfun.
3 7
```

等价于:

```
Python
1 myfun = verbosity_wrapper(myfun)
```

### Logic ▾

装饰器语法糖 (`@decorator_name`) 在函数定义时立即生效。  
这意味着函数名 `myfun` 实际上被替换为包装后的版本。

- 常见应用场景

- 调试与日志: 记录函数的输入、输出、执行时间。
- 性能监控: 统计函数运行耗时。
- 访问控制: 验证权限或状态。
- 缓存 (memoization): 保存函数结果以减少重复计算。
- 异步/并行执行: 如 `@dask.delayed` 或 `@numba.jit`。

- 示例: 计时装饰器

```
Python
1 import time
2
3 def timer(func):
4     def wrapper(*args, **kwargs):
5         start = time.time()
6         result = func(*args, **kwargs)
7         end = time.time()
8         print(f"{func.__name__} took {end - start:.4f} seconds.")
9         return result
10    return wrapper
11
12 @timer
13 def slow_function():
14     time.sleep(1)
15     return "Done"
16
17 slow_function()
18 ## 输出: slow_function took 1.000x seconds.
```

## ⚠ Remark ▾

使用装饰器后，函数的元数据（如 `__name__`、`__doc__`）会被包装函数覆盖。  
为保持原信息，应使用 `functools.wraps()`：



Python

```
1 from functools import wraps
2
3 def timer(func):
4     @wraps(func)
5     def wrapper(*args, **kwargs):
6         ...
7     return wrapper
```

否则 `func.__name__` 会显示为 `"wrapper"` 而非原函数名。

### • 装饰器的高级特性

- **带参数的装饰器**：装饰器本身再被一个函数包装，用于接收配置参数。



Python

```
1 def repeat(n):
2     def decorator(func):
3         def wrapper(*args, **kwargs):
4             for _ in range(n):
5                 func(*args, **kwargs)
6             return wrapper
7         return decorator
8
9 @repeat(3)
10 def greet():
11     print("Hello!")
```

输出：

```
1 Hello!
2 Hello!
3 Hello!
```

- **装饰器链 (decorator stacking)**：  
多个装饰器可叠加执行，按自下而上的顺序调用。

### • 现实中的装饰器示例

- **Dask 并行计算**：  
`@dask.delayed` 将普通函数转为延迟计算任务。
- **Numba JIT 编译**：  
`@numba.jit` 将 Python 函数即时编译为高性能机器码。
- **Flask 路由注册**：  
`@app.route('/home')` 绑定 URL 与处理函数。

## 🔗 Logic ▾

装饰器的强大之处在于——  
它让函数的“行为增强”可以独立定义、复用、组合，  
从而实现更简洁、更模块化的程序结构。

## 7 Memory and Copies

### 7.1 Memory and Copies Overview

#### 7.1.1 Overview

- 需要记住的两点：
  - 数值数组（**numeric arrays**）每个元素占用 **8 bytes**；
  - 需要留意何时创建了大型对象的**副本**（**copy**），以及何时只是复制了对象的**引用**（**reference**）。



Python

```
1 x = np.random.normal(size=5)
2 x.itemsize  ## 每个元素8字节
3 ## 8
4 x.nbytes    ## 总字节数 = 8 * 5 = 40
5 ## 40
```

#### Logic ▾

若数组包含 **n** 个元素且每个元素占 8 字节，则总内存大小约为 **8n bytes**。  
对于大型 **NumPy** 数据结构，这一点尤其关键。

#### 7.1.2 Allocating and Freeing Memory

- Python 不需要显式内存分配**：  
与 C 这类编译语言不同，Python 会自动管理内存。  
当对象不再被引用时，系统通过 **垃圾回收（garbage collection）** 释放内存。
- 删除对象**
  - del x**：仅解除变量名与对象之间的绑定；
  - 实际内存回收由垃圾回收器自动完成。
  - 一般无需手动调用 **gc.collect()**，除非要立即回收大对象占用的空间。

#### ⚠ Remark ▾

在 Python 中调用 **del** 并不会立刻释放内存，只是让该对象的引用计数归零。  
当没有变量引用该对象时，Python 才会在下一次垃圾回收中释放它。

- 与 C 的比较**
  - 在 C 中，开发者必须手动分配（**malloc**）与释放（**free**）内存。
  - 若循环中忘记释放，会导致 **内存泄漏（memory leak）**。
  - Python 自动回收内存，**通常不会发生内存泄漏**，但当存在循环引用或外部资源时仍可能出现。

Python 的内存管理基于“引用计数（reference counting）+ 垃圾回收器（GC）”。  
当对象的引用计数降为零时，其占用的内存会被标记为可回收。

### 7.1.3 The Heap and the Stack

- **堆 (Heap)**
  - 程序运行时用于动态创建对象的内存区域。
  - 在 Python 中，大部分对象（包括列表、字典、NumPy 数组）都分配在堆上。
  - 当对象被删除或失去引用时，Python 负责将其内存释放回堆。
- **栈 (Stack)**
  - 用于存储函数调用时的局部变量与执行帧（frames）。
  - 每次函数调用时会在栈上分配新的**栈帧（stack frame）**，函数结束后自动弹出。
  - “**Stack trace（调用栈追踪）**”即指当前所有活动函数帧的堆叠结构。

#### ⚠ Remark ▾

- 栈是**函数调用级**的内存结构，生命周期短且自动管理。
- 堆是**对象级**的内存结构，生命周期可跨函数存在。

因此：

- 在递归过深时会出现 **StackOverflowError**；
- 而在频繁创建大型对象时可能出现 **heap memory exhaustion**（堆内存不足）。

- **Stack**：临时变量、函数调用上下文（自动管理）。
- **Heap**：对象与动态数据（垃圾回收管理）。

它们的区别类似于短期与长期存储：

栈就像 CPU 的“笔记本”，堆更像电脑内存中长期保存的数据区。

## 7.2 Monitoring Memory Use

### 7.2.1 Monitoring Overall Memory Use on UNIX Systems

- **理解操作系统的内存缓存（Disk Caching）**
  - 操作系统会将从磁盘读取的文件或数据 **缓存（cache）到内存中**。
  - 若下次访问相同数据时仍在内存中，则无需重新从磁盘读取，大幅提高访问速度。
  - 尽管缓存会占用内存，但**这些缓存空间在需要时可立即被其他进程使用**，因此它是“可用”的。

“被缓存” ≠ “被占用”。

在 Linux 中，**缓存与缓冲（buff/cache）** 表示“可立即回收”的内存。

因此，系统内存即便显示“使用率高”，也可能仍有大量可用内存。

### 7.2.1.1 Using `free -h --si`

Bash

```
1 free -h --si
```

选项说明：

- `-h` → human-readable，以更友好的单位显示（如 GB）。
- `--si` → 使用 **十进制 (GB)** 而非 **二进制 (GiB)** 单位。

示例输出：

1		total	used	free	shared	buff/cache	available
2	Mem:	135G	24G	30G	7.8M	80G	110G
3	Swap:	274G	44G	230G			

- `used`：当前被进程实际占用的内存（24G）
- `buff/cache`：用于缓存但可被回收的内存（80G）
- `available`：可立即供新进程使用的总量（ $\approx \text{free} + \text{buff/cache} = 110\text{G}$ ）

### 7.2.1.2 Using `top`

Bash

```
1 top
```

输出示例：

1	MiB Mem : 128877.0 total, 28825.9 free, 23856.4 used, 77249.1 buff/cache
2	MiB Swap: 262144.0 total, 220103.3 free, 42040.7 used. 105020.6 avail Mem

解释：

- 系统总内存：129 GiB
- 实际使用：24 GiB
- 可用内存：106 GiB（ $\approx 29 \text{ GiB free} + 77 \text{ GiB buff/cache}$ ）

### 7.2.1.3 Swap 的含义

- **Swap 是磁盘上模拟的“虚拟内存”。**
- 当物理内存不足时，系统会将部分数据转移到 swap 空间。
- 由于磁盘读写速度远慢于内存，**使用 swap 会严重降低性能。**

 Remark ▾

如果 `Swap` 使用率很高（如上例 42 GiB），即使还有充足的物理内存，也可能是系统内存管理策略或 I/O 优化造成。通常，应尽量避免任务依赖 swap 空间。

## Logic ▾

`free` 与 `top` 输出可能略有差异，这可能源自单位差异（GB vs GiB）或系统更新周期不同。不影响总体判断：只需关注 `Mem` 行的 `used` 与 `available`。

## 7.2.2 Monitoring Memory Use in Python

- 使用 **UNIX 工具**
  - 在 Python 程序运行时，可直接在终端使用 `top` 或 `htop` 监控实时内存变化。

### 7.2.2.1 使用 `psutil` 获取进程内存

```
Python
1 import psutil
2 memory_info = psutil.Process().memory_info()
3 print("Memory usage:", memory_info.rss / 10**6, "MB.")
4 ## Memory usage: 310.6816 MB.
```

可封装为函数：

```
Python
1 def mem_used():
2     print("Memory usage:", psutil.Process().memory_info().rss / 10**6, " MB.")
```

## Logic ▾

`rss`（Resident Set Size）表示当前进程在物理内存中的占用量。

### 7.2.2.2 查看对象大小

- `sys.getsizeof()` 返回对象自身所占字节数：

```
Python
1 my_list = [1, 2, 3, 4, 5]
2 sys.getsizeof(my_list)    ## 104 bytes
3 x = np.random.normal(size=10**7)
4 sys.getsizeof(x)          ## ~80 MB (80000112 bytes)
```

- 但若对象引用了其他对象，结果可能低估内存占用：

```
Python
1 y = [3, x]
2 sys.getsizeof(y)    ## 仅72 bytes!
```

## ⚠ Remark ▾

`sys.getsizeof()` 不会递归计算引用对象的大小。  
若对象内部包含大型数组或字典，需使用更深入的测量方法。

### 7.2.2.3 更准确的测量：序列化法

```
Python
1 import pickle
2 ser_object = pickle.dumps(y)
3 sys.getsizeof(ser_object)  ## 80000202 bytes (~80 MB)
```

通过序列化（`pickle.dumps`）获取对象完整的二进制表示长度，可更准确反映真实内存占用。

### 7.2.2.4 其他工具与方法

- 启动 Python 时可添加调试选项以查看内存分配（详见 `man python`）。
- 专业分析工具：
  - `memory_profiler` – 逐行监测函数内存使用。
  - `pympler` – 提供详细对象级内存报告与可视化支持。

#### Logic ▾

当进行大规模数据分析或机器学习时，  
结合 `psutil` + `memory_profiler` 能有效识别内存瓶颈。

## 7.3 How Memory Is Used in Python

### 7.3.1 Two Key Tools: `id()` and `is`

- `id(obj)`  
返回对象在内存中的唯一标识（通常是对象的内存地址）。
- `is`  
判断两个变量是否指向同一个内存对象。

```
Python
1 x = np.random.normal(size=10**7)
2 id(x)  ## 127279918838576
3 sys.getsizeof(x)  ## 80000112 bytes
4
5 y = x
6 id(y)  ## 127279918838576
7 x is y  ## True
```

此时 `x` 与 `y` 共享同一内存。

```
Python
1 z = x.copy()
2 id(z)  ## 不同内存地址
```

 Logic ▾

`id()` 用于追踪对象身份，而 `is` 判断“对象是否为同一个”。  
若两个对象 `is` 相同，则修改其中一个会影响另一个。

## 7.3.2 Memory Use in Specific Circumstances

### 7.3.2.1 How Lists Are Stored

- 列表 (list) 是 **对象引用的集合**，而非一整块连续的内存。
- 每个元素是指向实际对象的 **指针 (reference)**。



Python

```
1 nums = np.random.normal(size=5)
2 obj = [nums, nums, np.random.normal(size=5), ['adfs']]
```

观察内存地址：



Python

```
1 id(obj)          ## 列表自身地址
2 id(obj[0])       ## 指向 nums
3 id(obj[1])       ## 同 nums
4 id(obj[2])       ## 不同的数组
5 id(obj[3])       ## 子列表对象
```

结果：

- `obj[0]` 与 `obj[1]` 指向同一个对象；
- 不同元素可引用相同对象。

 Remark ▾

列表本身仅存储引用（指针）。  
访问 `obj[0]` 实际上执行一次索引取值并创建临时引用传递给 `id()`。

### 7.3.2.2 How Character Strings Are Stored

字符串（以及整数）在底层也可能被重用（interning），即相同文本值可能共用内存地址。  
这些机制与 Python 的内存优化策略有关。

### 7.3.2.3 How NumPy Arrays Are Stored

- NumPy 数组的数据存储在 **连续内存块 (contiguous memory block)** 中。
- 每个元素不是独立对象，而是 **8 字节浮点值** 的一部分。
- 访问元素时，会创建一个新的临时 Python `float64` 对象。



Python

```
1 x = np.random.normal(size=5)
2 type(x[1])  ## numpy.float64
3 id(x[1])    ## 每次都不同!
```

#### Logic ▾

因为每次 `x[1]` 都会生成一个新的 Python 对象，所以连续两次访问同一个元素，其 `id()` 不相同。

- 若将数组序列化 (pickle)，大小  $\approx$  仅包含数值存储所需的空间 + 112 字节元数据。
- 若序列化一个包含相同数字的列表，则空间翻倍（因需存储引用指针）。

### 7.3.3 Modifying Elements In-Place



Python

```
1 x = np.random.normal(size=5)
2 id(x)
3 x[2] = 3.5
4 id(x)
```

`id()` 不变，说明修改数组元素不会创建副本。

#### Logic ▾

若每次修改都复制整个对象，操作大型数组几乎不可行，因此 NumPy 允许原地修改。

### 7.3.4 Shallow Copying

- **浅拷贝 (shallow copy)**：仅复制最外层容器，内部元素仍指向原对象。



Python

```
1 x = [3, [1,2,3]]
2 y = x.copy()
3 x[0] = 9
4 y[1][2] = 99
5 x  ## [9, [1, 2, 99]]
```

#### Remark ▾

改变嵌套元素（如子列表）会影响原对象，因为它们仍共享相同的引用。

- **深拷贝 (deep copy)**：复制整个结构，不共享任何子对象。



Python

```
1 import copy
```

```

2 x = [3, [1,2,3]]
3 y = copy.deepcopy(x)
4 y[1][2] = 99
5 x ## [3, [1, 2, 3]]

```

### 7.3.5 When Are Copies Made?

```

Python
1 x = np.random.normal(size=10**8)
2 y = x          ## 同一对象，共享内存
3 x = np.random.normal(size=10**8) ## 新分配内存

```

#### Logic

只有当变量重新绑定（reassignment）到新对象时，Python 才会分配新内存。

### 7.3.6 How Does Python Know When to Free Memory?

- Python 采用 **引用计数（reference counting）**。
- 每个对象都有一个引用计数，当计数为 0 时即被销毁并回收内存。

```

Python
1 import sys
2 x = np.random.normal(size=10**8)
3 y = x
4 sys.getrefcount(y) ## 3
5 del x
6 sys.getrefcount(y) ## 2
7 del y          ## 对象被释放

```

#### Remark

`getrefcount()` 返回值比预期多 1，  
因为函数调用本身会临时创建一个引用。

示例：

```

Python
1 x = np.random.normal(size=5)
2 sys.getrefcount(x) ## 实际1，显示2
3 y = x
4 sys.getrefcount(x) ## 实际2，显示3
5 del y
6 sys.getrefcount(x) ## 实际1，显示2

```

#### Logic

这种机制类似于 C++ 的 **shared pointers** 或 R 的 **copy-on-write**。  
当最后一个引用消失时，对象即被垃圾回收（GC）回收释放内存。

## 7.4 Strategies for Saving Memory

### 7.4.1 Basic Strategies

#### 1. Avoid unnecessary copies

- 不要轻易复制大型对象。
- 优先使用引用（reference）而非 `.copy()`，除非确实需要独立副本。

#### 2. Remove unused objects

- 删除不再使用的变量：

```
Python
1 del large_array
```

- 这会解除名称与对象的绑定，让垃圾回收器（GC）自动回收内存。

#### 3. Use iterators and generators

- 避免创建大型中间列表，例如：

```
Python
1 ## 差：占用大量内存
2 nums = [i**2 for i in range(10**7)]
3 ## 好：使用生成器，节省内存
4 nums = (i**2 for i in range(10**7))
```

- 迭代器（iterator）与生成器（generator）**惰性求值**，只在需要时生成元素。

#### Logic

Python 的列表会一次性将所有元素加载入内存，而生成器仅保存计算逻辑与当前位置索引。  
因此，对于大规模数据循环处理，生成器能显著降低内存占用。

### 7.4.2 Advanced Optimization

#### • 使用更紧凑的数据类型

- 浮点型默认是 `float64`（8 bytes），但在精度要求不高时可用更小类型。

```
Python
1 x = np.array(np.random.normal(size=5), dtype="float32")
2 x.itemsize ## 4 bytes
3 x = np.array([3,4,2,-2], dtype="int16")
4 x.itemsize ## 2 bytes
```

- 减小 `dtype` 直接减少数组内存占用。

#### Remark

在数值分析中，使用 `float32` 可能会造成累积误差或舍入误差；  
因此仅在确认结果可接受时才降精度。

- 分块 (chunked) 读取数据

- 对大文件，不必一次性加载全部数据，可采用分块读取：

```
Python
1 for chunk in pd.read_csv("data.csv", chunksize=10000):
2     process(chunk)
```

- 有助于防止内存峰值溢出，尤其在处理数 GB 文件时。

- 使用高效的存储格式 / 工具包

- 可考虑使用 **Apache Arrow**、**Parquet** 等高效的内存结构与文件格式。
- 它们支持零拷贝 (zero-copy) 与压缩编码，大幅减少内存占用。

## 7.4.3 Example: Memory Allocation in Practice

以下代码展示了一个示例函数 `fastcount()`，它用于统计两个大型数组 `x` 与 `y` 中非 NaN 元素的组合，但会引发多次内存分配：

```
Python
1 def fastcount(xvar, yvar):
2     naline = np.isnan(xvar)
3     naline[np.isnan(yvar)] = True
4     localx = xvar.copy()
5     localy = yvar.copy()
6     localx[naline] = 0
7     localy[naline] = 0
8     useline = ~naline
9     ### ...
```

### ⚠ Remark ▾

#### 内存分配点：

1. `np.isnan(xvar)` → 新建布尔数组（与 `xvar` 同尺寸）
2. `np.isnan(yvar)` → 另一个布尔数组
3. `xvar.copy()` 与 `yvar.copy()` → 两个完整副本（约两倍原内存）
4. `naline` 的逻辑运算（`~naline`）生成新数组 `useline`

由此可见，在处理大型 NumPy 数组时：

- 每次 `.copy()`、逻辑运算或布尔索引操作都会触发新的内存分配；
- 若输入数组在 1 GB 量级，整个函数可能瞬间占用 3–4 GB 内存。

### 💡 Logic ▾

#### 优化建议：

- 尽量使用 **in-place 操作**（如 `np.nan_to_num(xvar, copy=False)`）；
- 复用已有布尔掩码数组而非重复创建；
- 若结果不需长期保存，及时释放局部变量以触发垃圾回收。

## 8 Efficiency

## | 8.1 Interpreters and Compilation

### | 8.1.1 Why Are Interpreted Languages Slow?

- **编译语言（如 C、Fortran）**
  - 源代码在执行前被翻译成 **机器码（machine code）**（0 和 1），CPU 可直接执行这些指令。
  - 各种类型检查、内存定位等工作在编译阶段完成，因此运行时无需重复。
- **解释语言（如 Python、R）**
  - 在运行时由 **解释器（interpreter）** 逐行读取和执行。
  - 每次执行时都需：
    1. 检查变量是否存在；
    2. 确认类型是否合法；
    3. 查找对应作用域（scope）中的值与方法。
  - 这导致每次函数调用、运算符执行都需要大量“检查与查找”步骤。

示例：

```
Python
1 x = 3
2 abs(x)    ## 计算绝对值
3 x * 7
4 x = 'hi'
5 abs(x)    ## 错误：类型不匹配
6 x * 3     ## 字符串重复
```

解释器每次都要重新确认 `x` 的类型与操作的适用性。

### | 8.1.2 Dynamic Typing and Overhead

#### 🔗 Logic ▾

Python 是 **动态类型语言**：变量类型可随时改变。  
因此解释器必须在运行时不断验证每个操作是否合法。

示例（极端情况）：

```
Python
1 x = np.random.normal(size=10)
2 for i in range(10):
3     if np.random.normal() > 0:
4         x = 'hi'
5     if np.random.normal() > 0.5:
6         del x
7     x[i] = np.exp(x[i])
```

解释器必须在每次迭代中检查：

- `x` 是否存在；
- `x` 是否为数组；
- `x` 是否含有数值；

- `x[i]` 是否可修改。

这类动态检查使解释语言较慢。

而 **JIT (Just-In-Time) 编译** 的思想即尝试缓存这些类型与检查结果，以加速后续执行。

### 8.1.3 CPython 的本质

- **CPython** 是 Python 的标准实现，本质上是一个 **C 编译程序**，用于解释 Python 源码。
- 尽管底层是编译好的 C 代码，但每次执行 Python 指令时仍需大量额外操作（解析、检查、绑定）。
- 类比：直接用相同语言交流（编译代码） vs. 通过翻译转述（解释器）。

### 8.1.4 Built-in Safety Checks

- 许多 Python 函数在调用底层 C 库前都会执行大量检查。

例如：

```
Python
1 from scipy.linalg import solve_triangular
2 solve_triangular(A, b, check_finite=False)
```

若将 `check_finite=False`，性能可提高，但若输入中包含 NaN 或 Inf，则结果不可预期。

#### ⚠ Remark ▾

高级库函数（如 SciPy、NumPy）虽调用底层 C/Fortran 实现，但仍会在 Python 层面执行输入检查，这部分往往成为性能瓶颈。

### 8.1.5 What Executes Quickly in Python

#### 1. 调用底层编译函数的操作

- 如 NumPy 的算术与矩阵运算、Pandas 的矢量化操作。

#### 2. 线性代数（BLAS / LAPACK）调用

- 实际运算由高效的 Fortran/C 库完成。

#### 3. 矢量化（Vectorization）

- 避免 Python 层的循环，将循环下放至 C 层。

```
Python
1 ## 慢：Python 逐元素循环
2 for i in range(len(x)): x[i] = np.exp(x[i])
3 ## 快：C 层矢量化
4 x = np.exp(x)
```

在矢量化中，解释器不再对每次迭代做类型与存在性检查。

### 8.1.6 Compilation

#### 8.1.6.1 Overview

- **Compilation (编译)** 是将源码 (如 C++、Fortran) 转换为机器码的过程。
- 编译后的二进制文件 (executable) 可被 CPU 直接执行。
- 编译语言具备：
  - 运行速度快；
  - 类型静态、可优化；
  - 但开发周期相对较长。

#### 🔗 Logic ▾

编程语言的发展不断提升抽象层次：

Machine Code → Assembly → C/Fortran → Python/Julia。

抽象度越高，开发效率越高，但执行速度通常越慢。

### 8.1.6.2 Python Interpreter

- Python 解释器 (CPython) 本身是一个 **已编译的 C 程序**。
- 它负责解析并执行 `.py` 文件，但不会生成可直接执行的机器码。

### 8.1.7 Just-in-Time (JIT) Compilation

- **AOT (Ahead-of-Time)** 编译：  
代码在执行前被完全编译 (如 C、C++)。
- **JIT (Just-in-Time)** 编译：  
代码在运行时被动态编译为机器码，并可缓存结果。

特征：

- **类型推断 (type inference)**：推测变量类型以减少动态检查；
- **即时优化 (runtime optimization)**；
- **缓存 (caching)**：再次调用同一函数时无需重新编译。

示例：

- Julia 语言：大量使用 JIT，首轮运行慢，之后极快；
- Python：通过 `numba` 提供 JIT 编译支持。



Python

```
1 from numba import njit
2
3 @njit
4 def f(x):
5     return np.exp(x)
```

#### 🔗 Logic ▾

`numba.njit` 使用 LLVM 库将 Python + NumPy 代码编译为机器码，运行速度可接近 C。

## 8.1.8 Byte Compiling (可选)


- **字节码编译 (Byte Compilation)**

将 Python 源码转换为 **字节码 (bytecode)**，以便更快执行。


- 跳过了解析与语法分析阶段。
- 但字节码仍需由解释器执行，不是机器码。

生成的 `.pyc` 文件即为字节码文件。


示例：

```
 Python
1 import py_compile
2 py_compile.compile('vec.py')
3 ## '__pycache__/vec.cpython-312.pyc'
```

### 8.1.8.1 Performance Comparison

```
 Python
1 import time
2
3 def f(vals):
4     x = np.zeros(len(vals))
5     for i in range(len(vals)):
6         x[i] = np.exp(vals[i])
7     return x
8
9 x = np.random.normal(size=10**6)
10 ## 普通函数
11 t0 = time.time(); f(x); print(time.time()-t0) ## ~0.75 s
12 ## 矢量化操作
13 t0 = time.time(); np.exp(x); print(time.time()-t0) ## ~0.013 s
```

字节码化后：

```
 Python
1 import vec
2 vec.f(x) ## ~0.73 s
```

#### Remark ▾

`.pyc` 文件仅略微减少解释开销，对 CPU 密集型任务帮助有限。  
真正显著的性能提升通常来自：

- 使用 **NumPy/Cython/Numba** 的编译扩展；
- 或直接在 **C/Fortran** 中实现关键计算逻辑。