# Python程序设计与实践

## 第六课：推导式、测试调试、文件

2025. 4

# Today



- 列表推导式
- 函数的特定场景用法
- 测试与调试
- 异常与断言
- 文件的处理

# List Comprehensions



■ 场景：对一个序列中每一个元素做处理，然后创建一个新列表包含这些新元素

- 传统的做法：

```
def f(L):
    Lnew = []
    for e in L:
        Lnew.append(e**2)
    return Lnew
```

新列表

给每个元素施加处理逻辑

- Python提供了一种更简洁、一行代码表示的方式，叫做列表推导式

# List Comprehensions

■ 场景：对一个序列中每一个元素做处理，然后创建一个新列表包含这些新元素

```
def f(L):
    Lnew = []
    for e in L:
        Lnew.append(e**2)
    return Lnew
```

新列表

遍历每个元素

施加处理逻辑

```
Lnew = [e**2 for e in L]
```

新列表

施加处理逻辑　　遍历每个元素

# List Comprehensions



- 场景：对一个序列中每一个元素做处理，然后创建一个新列表包含这些新元素

```
def f(L):
    Lnew = []
    for e in L:
        if e%2==0:
            Lnew.append(e**2)
    return Lnew
```

新列表

遍历每个元素

在满足条件时处理

施加处理逻辑

```
Lnew = [e**2 for e in L if e%2==0]
```

# List Comprehensions



■ 列表推导式:

```
[expression for elem in iterable if test]
```

```
[e**2 for e in range(6)]
```

→ [0, 1, 4, 9, 16, 25]

```
[e**2 for e in range(8) if e%2 == 0]
```

→ [0, 4, 16, 36]

```
[[e,e**2] for e in range(4) if e%2 != 0]
```

→ [[1,1], [3,9]]

# List Comprehensions



- 思考以下列表推导式的输出是什么，并尝试执行：

```
[len(x) for x in ['xy', 'abcd', 7, '4.0'] if type(x) == str]
```

思考过程：

1. 列表中都有哪些元素？
2. 条件表达式能够筛选出哪些元素？
3. 给这些元素施加处理逻辑后变成什么？

# Default Parameter



■ 设计函数参数的另一种场景：默认参数值

需求：

1. 想在函数中添加一个参数，它有一个标准用法，输入的参数具有标准值
2. 但是也允许用户修改输入参数的值，保持灵活性
3. 例如：人工智能中训练一个模型用到的参数（最佳实践默认参数）

- 在设计函数时使用默认参数

# Default Parameter



- 设计函数参数的另一种场景：默认参数值

```
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    guess = (high + low)/2.0
    while abs(guess**2 - x) >= epsilon:
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (high + low)/2.0
    return guess
```

想保留这个阈值的设定，同时让用户可以选择其他值

# Default Parameter

浙江科技大学 ZHEJIANG UNIVERSITÄT FÜR WISSENSCHAFT UND TECHNIK

■ 设计函数参数的另一种场景：默认参数值

```
def bisection_root(x, epsilon=0.01):
    low = 0
    high = x
    guess = (high + low)/2.0
    while abs(guess**2 - x) >= epsilon:
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (high + low)/2.0
    return guess
```

设置默认参数，默认值0.01

忽略默认参数，用0.01

```
print(bisection_root(123))
print(bisection_root(123, 0.5))
```

修改默认参数为0.5

## 设计函数参数的另一种场景：默认参数值

- 注意：在调用函数时默认/关键参数必须放在位置参数的后面
- 调用函数时以下方式都是对的：

```
bisection_root_new(123)
```
位置参数（positional argument）

```
bisection_root_new(123, 0.001)
bisection_root_new(123, epsilon=0.001)
bisection_root_new(x=123, epsilon=0.1)
bisection_root_new(epsilon=0.1, x=123)
```

- 以下调用不正确：

关键参数（keyword argument）

```
bisection_root_new(epsilon=0.001, 123) #error
bisection_root_new(0.001, 123) #no error but wrong
```

```
SyntaxError: positional argument follows keyword argument
```

# Function Returns Functions



■ 函数使用的另一种场景：在函数中返回函数

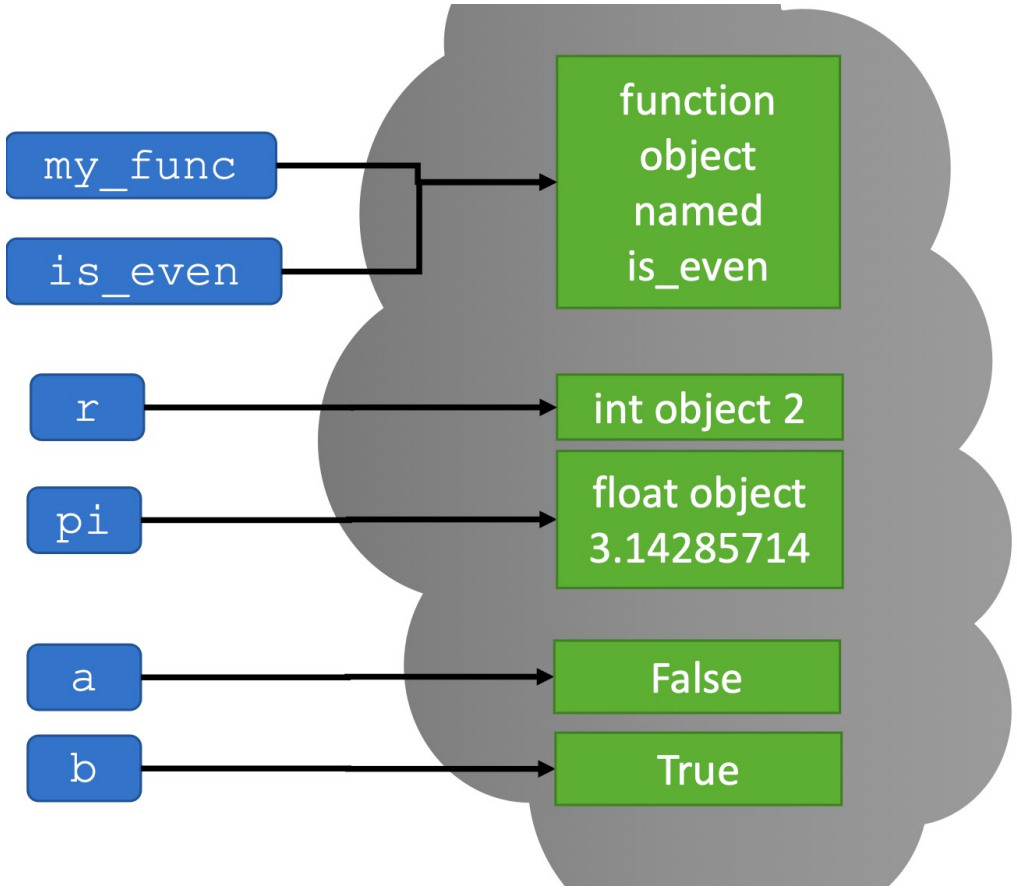
```
def is_even(i):
    return i%2 == 0

r = 2

pi = 22/7
```

不是函数调用，
只是变量名绑定

```
my_func = is_even
```

```
a = is_even(3)

b = my_func(4)
```

函数调用



# Function Returns Functions



■ 函数使用的另一种场景：在函数中返回函数

```
def make_prod(a):
    def g(b):
        return a*b
    return g
```

在一个函数中定义另一个函数

不是函数调用，只是函数绑定的名称

```
val = make_prod(2)(3)
print(val)
```

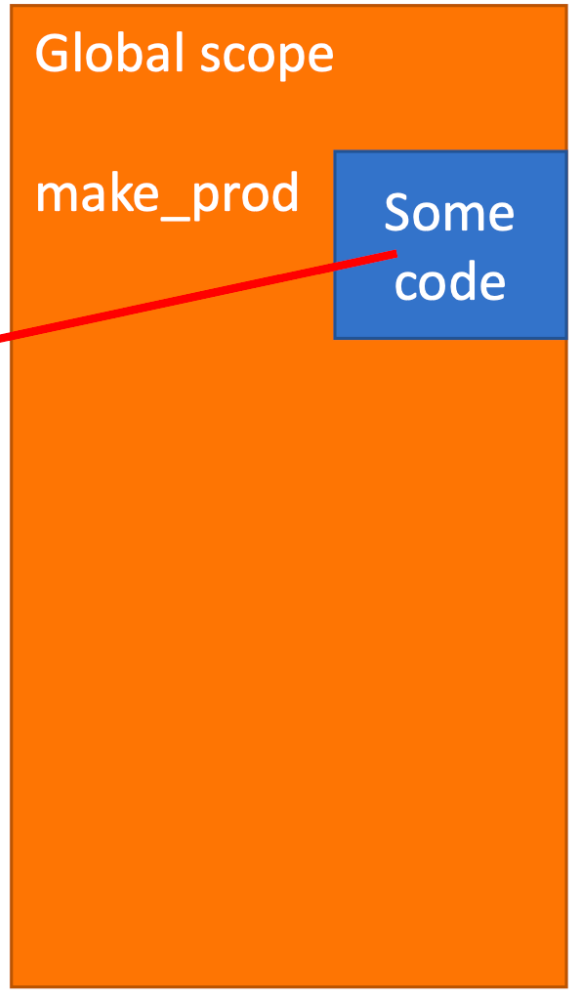
```
doubler = make_prod(2)
val = doubler(3)
print(val)
```

# Function Returns Functions



- 函数使用的另一种场景：在函数中返回函数

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```

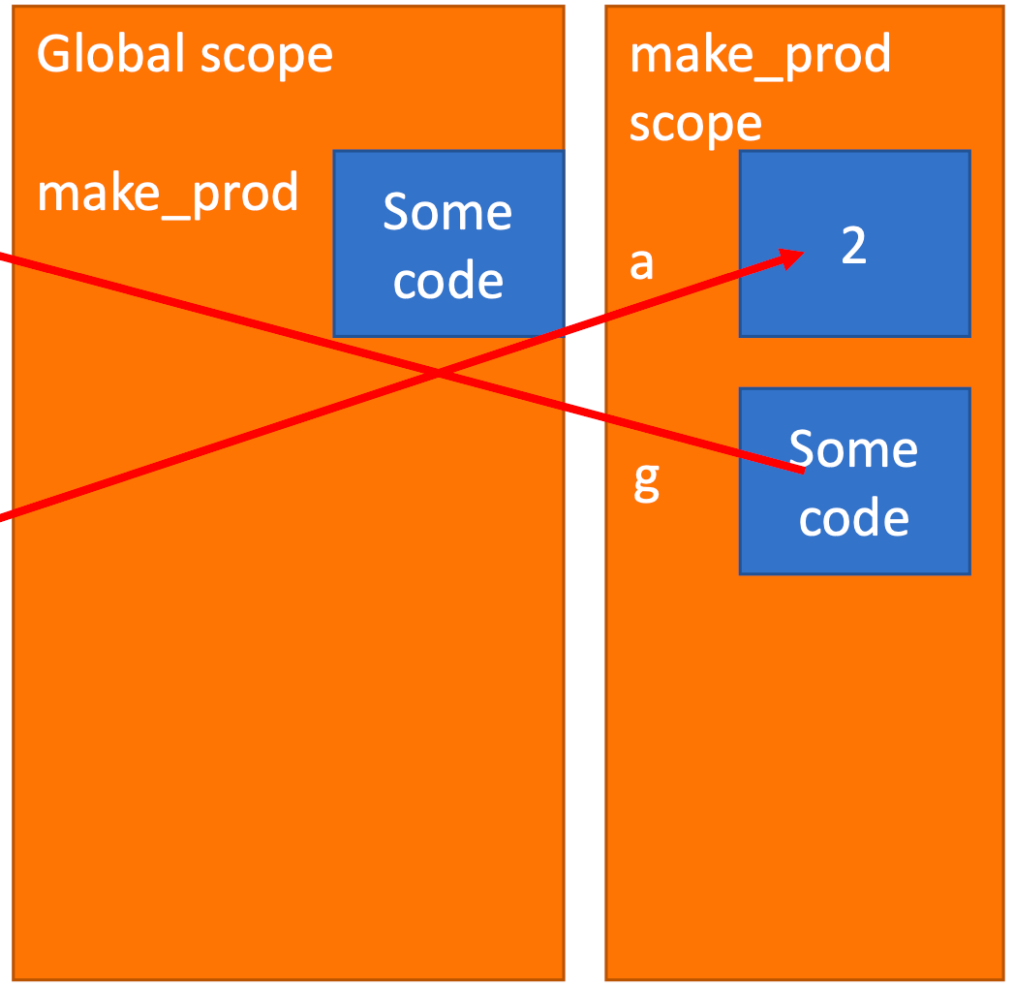


# Function Returns Functions



■ 函数使用的另一种场景：在函数中返回函数

```
def make_prod(a):
    def g(b):
        return a*b
    return g


val = make_prod(2)(3)
print(val)
```

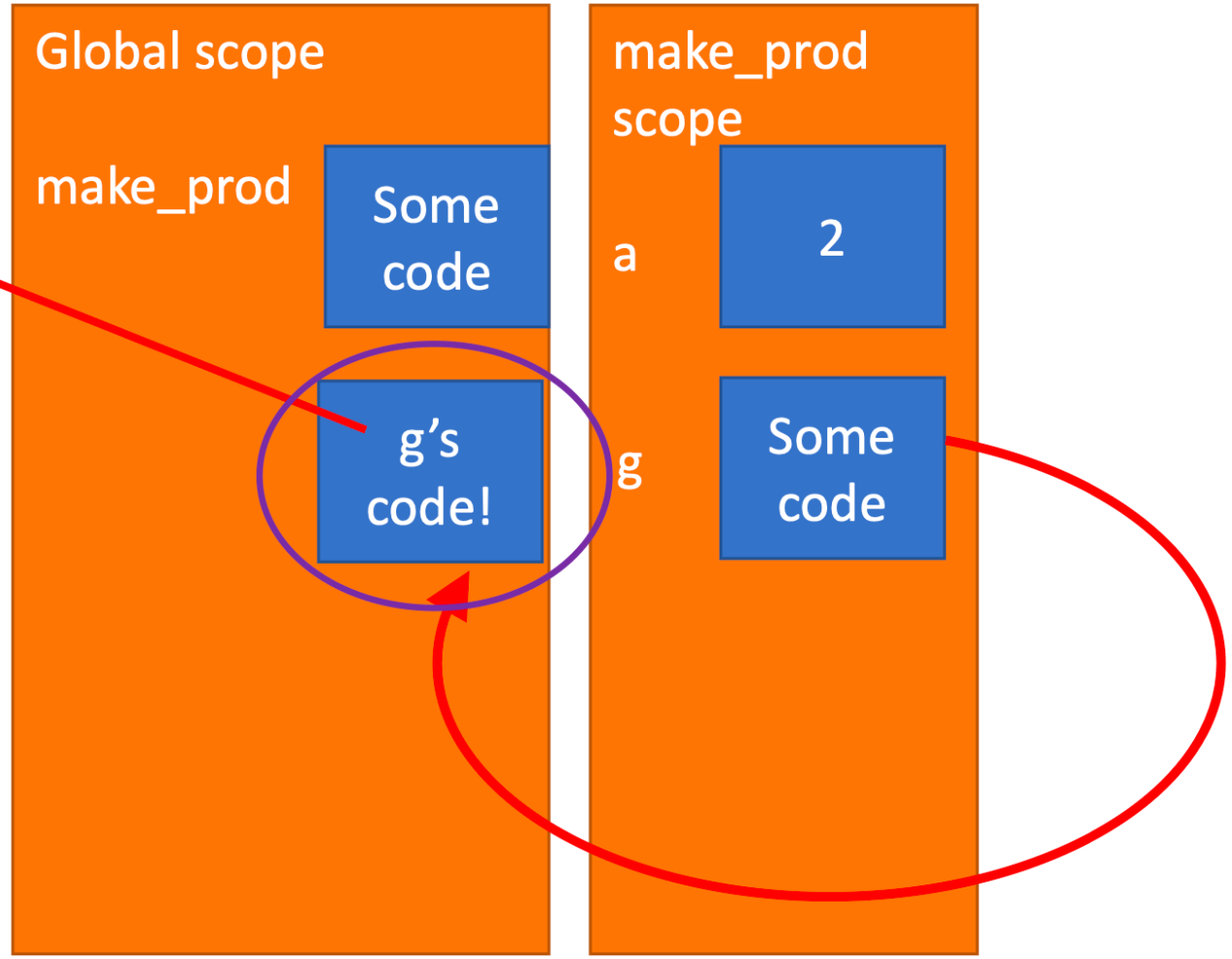


# Function Returns Functions

- 函数使用的另一种场景：在函数中返回函数

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```

This is g

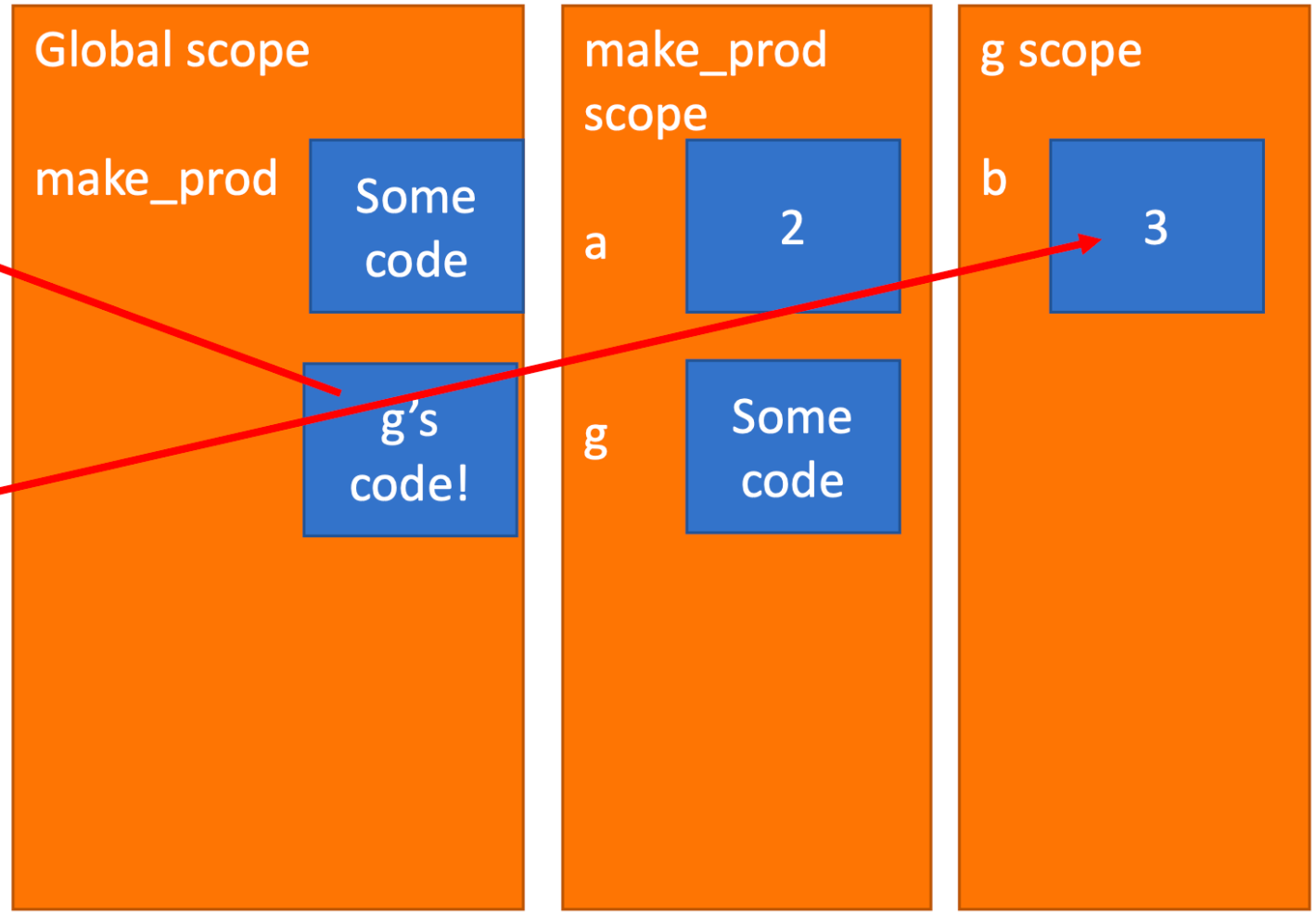


# Function Returns Functions



■ 函数使用的另一种场景：在函数中返回函数

```
def make_prod(a):
    def g(b):
        return a*b
    return g


val = make_prod(2)(3)
print(val)
```

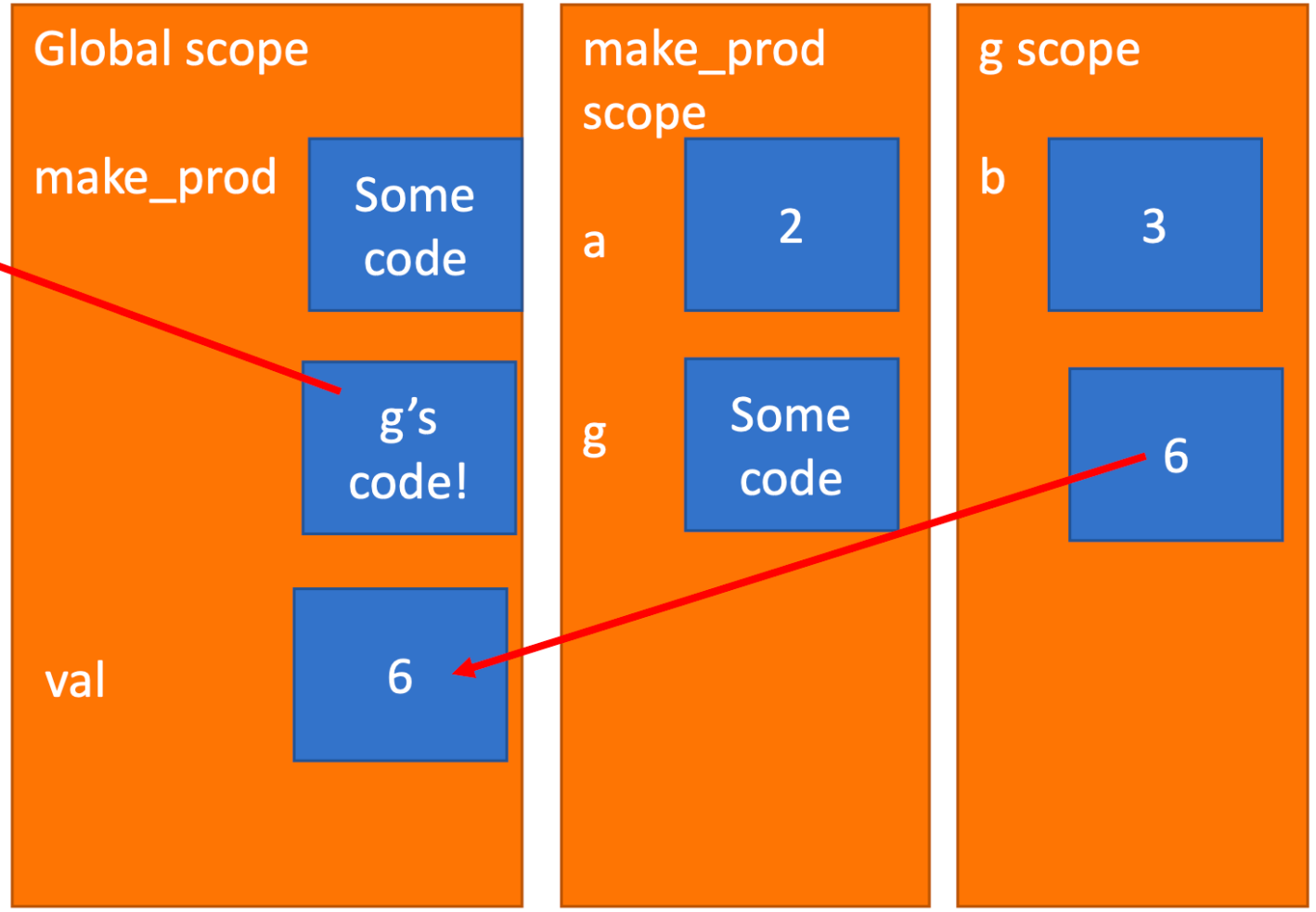


# Function Returns Functions



■ 函数使用的另一种场景：在函数中返回函数

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```



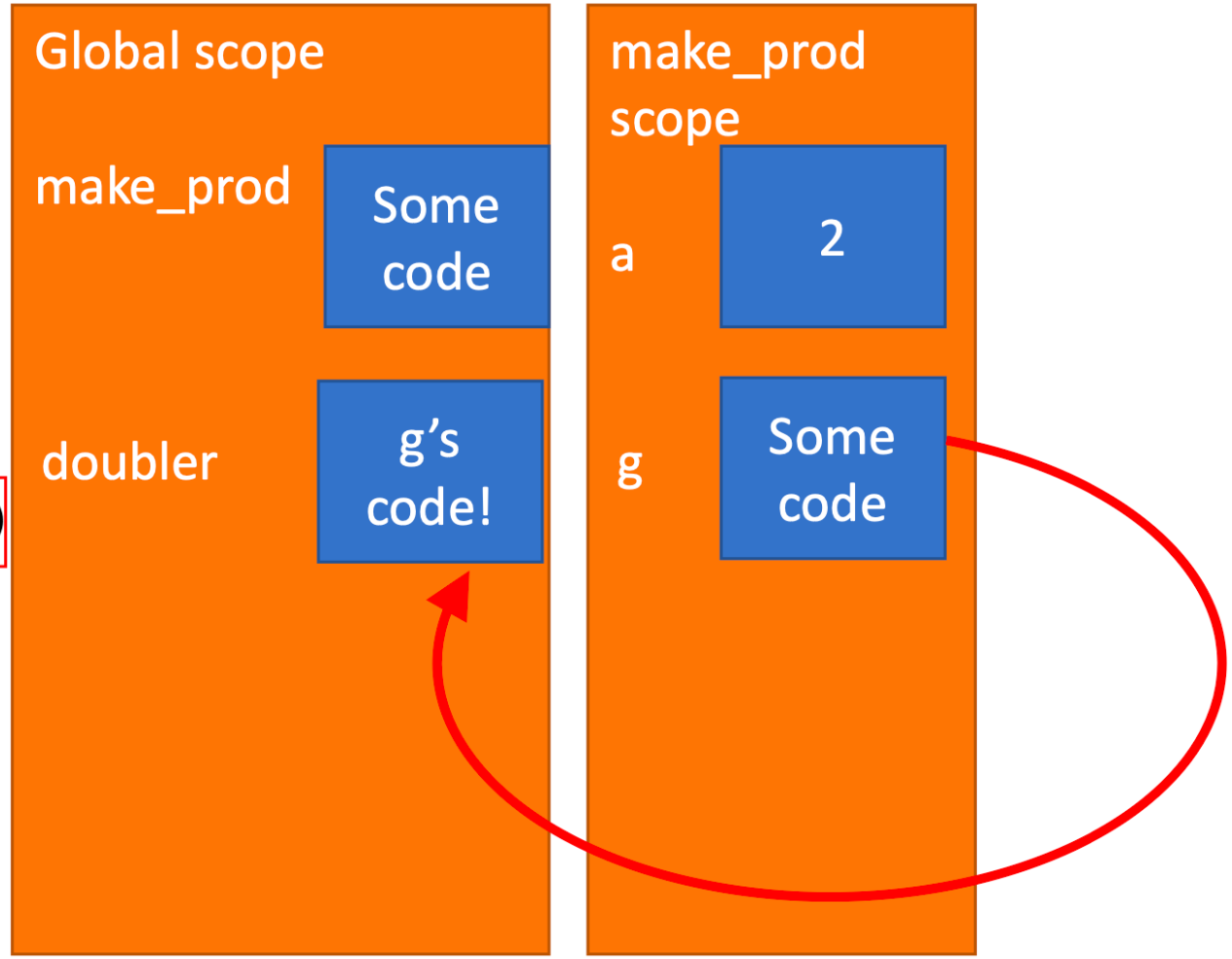
# Function Returns Functions



- 函数使用的另一种场景：在函数中返回函数

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```

g



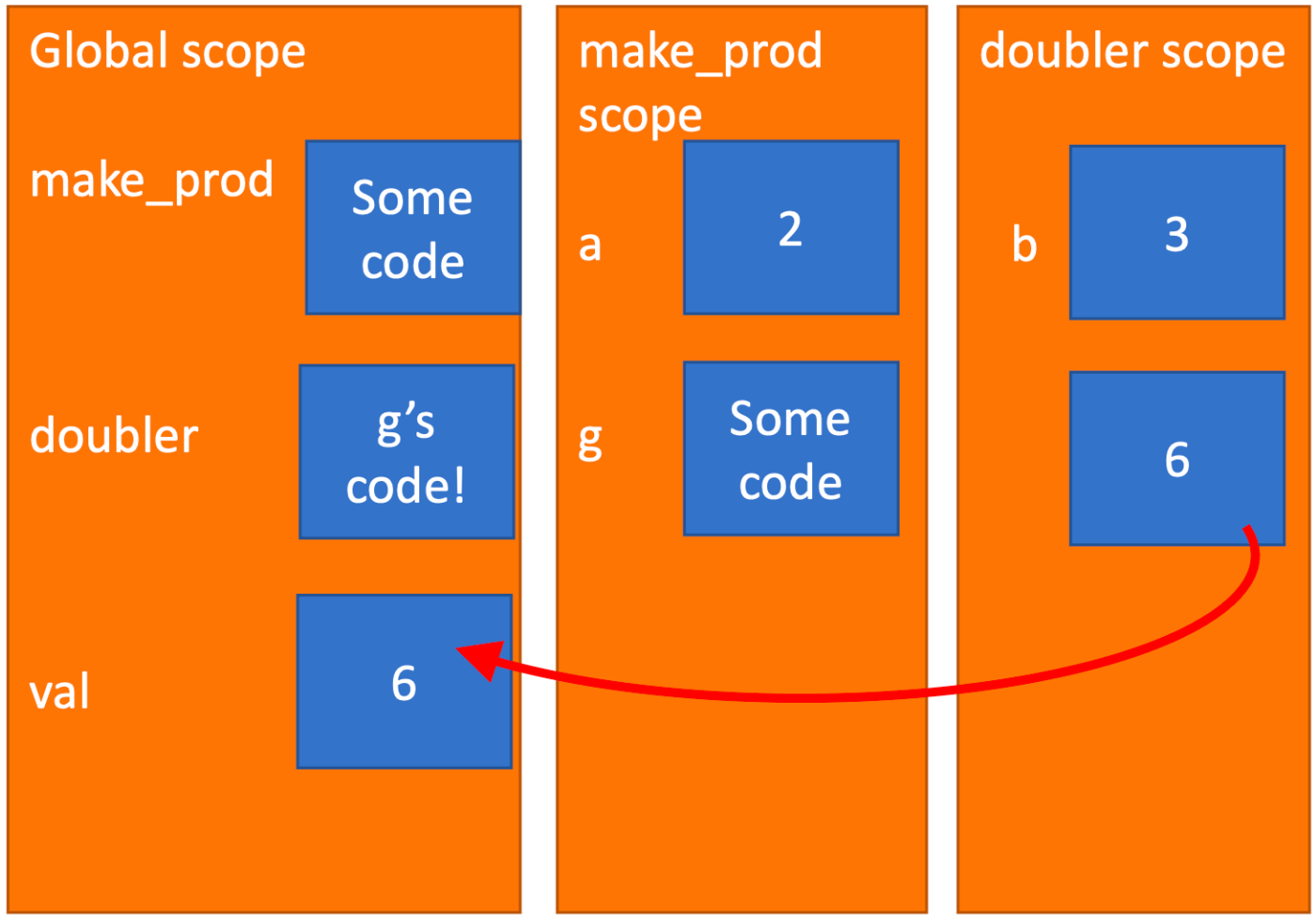
# Function Returns Functions



■ 函数使用的另一种场景：在函数中返回函数

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```

Now invoking g(3)



# Function Returns Functions



## ■ 来看另一种情况

```
def create_funcs():
    funcs = []
    for i in range(3):
        def inner():
            return i
        funcs.append(inner)
    return funcs
f1, f2, f3 = create_funcs()
print(f1(), f2(), f3())
```

内部定义函数中的变量是在调用时动态获取值的，并非定义时的值。

在循环中定义另一个函数

输出什么?

2 2 2

# Function Returns Functions



## ■ 来看另一种情况

```
def create_funcs():
    funcs = []
    for i in range(3):
        def inner():
            return i
        funcs.append(inner)
    return funcs
f1, f2, f3 = create_funcs()
print(f1(), f2(), f3())
```

内部定义函数中的变量是在调用时动态获取值的，并非定义时的值。

解决办法：使用默认参数立即获取变量值

```
def inner(i=i):
    return i
```

```
0 1 2
```

# Function Returns Functions



## ■ 再来看一种情况

```
def outer():
    data = []
    def inner(x):
        data.append(x)
        return data
    return inner
```

外部变量为可变对象

内部对变量修改

```
f1 = outer()
f2 = outer()
print(f1(1))
print(f2(2))
```

调用两次外部函数

[1]

[2]

因为每次调用outer()会创建新的data对象

# Function Returns Functions



■ 再来看一种情况

```
def outer(data=[]):
    def inner(x):
        data.append(x)
        return data
    return inner
```

默认参数可变

内部对变量修改

```
f1 = outer()
f2 = outer()
print(f1(1), f2(2))
```

调用两次外部函数

```
[1,2] [1,2]
```

默认参数在多次调用中共享使用（指向同一个list对象）

导致结果意外共享

- 防御式编程：
  - 为函数编写规格说明（docstring）
  - 模块化编写代码（函数设计）
  - 为输入输出检查是否满足条件要求（assertions）
  - 测试（testing）
    - 按照规格说明比较输入输出是否正确
  - 调试（debugging）
    - 找到什么事件导致了错误的产生

- 防御式编程：
  - 从编程的一开始就把代码设计得易于测试和调试
  - 将完整的代码分解为多个模块，每个模块更易于测试和调试
  - 记录下每一个模块的预期输入与对应的输出

- 测试的类型：
  - 单元测试（Unit Testing）
    - 验证每一个代码块的功能
    - 单独测试每一个函数的正确性
  - 回归测试（Regression Testing）
    - 在找到bug后进行测试
    - 找出在之前没有但是新引入的错误
  - 集成测试（Integration Testing）
    - 测试多个模块连接起来是否正确

- 测试的方法：
  - **黑盒测试**：关注功能，不关心内部实现（用户视角）
  - **白盒测试**：关注代码逻辑，覆盖路径（开发者视角）
  - **常用工具**：

    unittest, pytest, doctest, coverage

## ■ 测试的方法：

- **黑盒测试**：功能验证为核心
  - ➢ **核心思想**：输入 → 输出验证，忽略内部代码结构
  - ➢ **适用场景**：功能验收测试、API接口测试

```
def add(a, b):
    return a + b # 待测试的逻辑
def test_add():
    assert add(2, 3) == 5 # 正常输入
    assert add(-1, 1) == 0 # 边界值
    assert add("a", "b") == "ab" # 潜在错误输入（需处理异常）
```

Assert：断言，后面跟表达式

如果表达式为True，直接跳过

如果表达式为False，返回AssertionError

## ■ 测试的方法：

- **黑盒测试**：功能验证为核心

```
def sqrt(x, eps):
""" Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

| case | x | eps |
|---|---|---|
| 边界测试 | 0 | 0.0001 |
| 平方数测试 | 25 | 0.0001 |
| 小于1时测试 | 0.05 | 0.0001 |
| 无理数平方根 | 2 | 0.0001 |
| 极限值 | 2 | 1.0/2.0**64.0 |
| 极限值 | 1.0/2.0**64.0 | 1.0/2.0**64.0 |
| 极限值 | 2.0**64.0 | 1.0/2.0**64.0 |

- 测试的方法：
  - **白盒测试**：代码逻辑全覆盖
    - **核心思想**：基于代码结构设计测试用例，覆盖分支、循环、条件
    - **适用场景**：单元测试、代码重构等

```
def is_even(num):
    if num % 2 == 0:
        return True
    else:
        return False
def test_is_even():
    assert is_even(4) is True # 分支1
    assert is_even(3) is False # 分支2
```

每一个潜在路径都至少被测试了1次

path-complete

缺点：产生大量测试用例、漏掉关键路径

- 测试的方法：
  - **白盒测试**：代码逻辑全覆盖

```
def abs(x):
    """ Assumes x is an int
    Returns x if x>=0 and -x otherwise """
    if x < -1:
        return -x
    else:
        return x
```

完整路径测试：x=2 和 x=-2

但是-1代入后返回-1

仍然需要测试边界条件

- 调试（debugging）：
  - 一旦你发现了代码不能正常运行，你希望：
    - 找到bug的位置
    - 根除掉bug
    - 重写测试代码直到能正确运行所有用例
  - 最耗时的环节
  - 利用工具提高效率
    - 使用IDLE中自带的调试功能
    - 使用Python自带的调试包（pdb）
    - 善用print函数

## ■ 调试（debugging）：

- 发现代码错误的难度不同
  - ➢ 通过报错信息就能定位到的错误：

    访问超过限制的列表：

    `test = [1,2,3]` 然后 `test[4]` → `IndexError`

    转换不合适的数据类型：

    `int(test)` → `TypeError`

    Python语法错误：

    `a = len([1,2,3]` → `SyntaxError`

- 调试（debugging）：
  - 发现代码错误的难度不同
    - 逻辑错误——难以定位
      - 在写新的代码前就充分想清楚
      - 可以通过画逻辑图、流程图提高对代码的认识
      - 向别人解释你代码的逻辑

- 调试（debugging）：
  - 调试代码的步骤：
    - 先理解你的代码
    - 然后时刻想着为什么你的代码出现了不满足预期的结果
    - 根据你的代码形成假设，应该产生什么结果
    - 充分运行你的代码确认大致出错位置：模块、函数
    - 使用最简单的例子测试出错位置
    - 一步步缩小测试范围，直到定位到语句

- 调试（debugging）：
  - 用print语句：
    - 在哪里使用print？——模块的开始与结束

      把进入函数时的参数值打印出来看看

      在循环之后把处理过的变量打印出来看看

      把跳出函数后的返回值打印出来看看
    - 高效使用print定位bug——二分查找法

      把print放在一段代码中间的位置

      根据打印的值确定bug的可能位置

- 调试（debugging）：
  - 用pdb调试代码的优势：
    - **无需IDE**：直接通过命令行调试，适用于服务器/无图形界面环境
    - **灵活嵌入**：通过代码插入断点，精准控制调试位置

```
import pdb

# your code

pdb.set_trace()
```

    - **动态观察**：运行时查看变量、修改状态、逐行跟踪逻辑
    - **学习代码**：深入理解程序执行流程，适合调试复杂逻辑（如递归、循环）

## ■ 调试（debugging）：

- pdb调试的用法：

在需要检查的代码位置插入断点：

```
import pdb
def calculate(a, b):
    pdb.set_trace() # 插入断点
    result = a * b
    return result + 10
print(calculate(3, 5))
```

```
> example.py
(Pdb) p a, b # 输出：3, 5
(Pdb) n # 执行下一行
(Pdb) p result # 输出：15
```

## ■ 调试（debugging）：

- pdb调试的用法：

```
l (list)      # 查看当前代码
n (next)      # 执行下一行
s (step)      # 进入函数内部
c (continue)  # 继续运行到下一个断点
p var         # 打印变量值
q (quit)      # 退出调试
```

## ■ 异常（Exceptions）

- 当程序执行到不符合预期的条件会抛出异常
- 例如：
  - ➢ 访问超出列表范围的元素： `IndexError`

```
test = [1,7,4]
test[4]
```

  - ➢ 对不合适的类型进行转换，或数据类型混用：`TypeError`

```
int(test)
'a'/4
```

  - ➢ 使用一个不存在的变量：`NameError`

```
a
```

## ■ 处理异常（Exceptions）

- 遇到异常时，程序会抛出错误，执行会被停止
- Python代码可以手动处理异常：

```
try:
    # do some potentially
    # problematic code
except:
    # do something to
    # handle the problem
```

  - ➢ 如果try内的代码都能成功执行，继续执行except后面的代码
  - ➢ 如果try内的代码会抛出异常，会执行except内的代码，然后继续执行之后的代码

## ■ 处理异常（Exceptions）

- 例子：将一个字符串中所有数字加起来

```
def sum_digits(s):
    """ s is a non-empty string
    containing digits.
    Returns sum of all chars that
    are digits """
    total = 0
    for char in s:
        if char in '0123456789':
            val = int(char)
            total += val
    return total
```

如果字符串中有非数字字符会抛出异常

```
def sum_digits(s):
    """ s is a non-empty string
    containing digits.
    Returns sum of all chars that
    are digits """
    total = 0
    for char in s:
        try:
            val = int(char)
            total += val
        except:
            print("can't convert", char)
    return total
```

## ■ 处理异常（Exceptions）

- 用户输入可能引发异常

```
a = int(input("Tell me one number:"))
b = int(input("Tell me another number:"))
print(a/b)
```

用户可能输入字符
用户可能输入b=0

在可能存在异常的代码周围使用try/except

```
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

## 处理特定类型的异常

- 我们可以根据不同类型的异常编写不同的处理逻辑

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
    print("a/b = infinity")
    print("a+b =", a+b)
except:
    print("Something went very wrong.")
```

只有在特定异常出现时处理

在其他异常出现时处理

## 跟异常处理同时使用的语句

- 当try中代码执行完成且没有异常出现时：
  - else
- 在try、except、else之后总是需要执行的代码，即使在代码中执行了break、continue、return也要执行
  - finally
  - 可用于清除代码信息，如关闭文件
- 需要知道这些用法，但不常用

## 遇到异常时如何处理?

- 需求一：安静地处理错误
  - 去掉默认的抛出异常的方式，让程序继续执行
  - 不建议，用户无法获得异常信息
- 需求二：返回一个错误的值
  - 问题：用什么值代替错误信息?
  - 不建议，处理起来较复杂，提高代码复杂度
- 需求三：停止执行，给出详细的错误信号
  - `raise ValueError("something is wrong")`

手动抛异常关键字　异常类型　　异常信息

## ■ 遇到异常时如何处理?

- 例子：当遇到异常时，处理异常信息并手动抛出

```
def sum_digits(s):
    """ s is a non-empty string containing digits.
    Returns sum of all chars that are digits """
    total = 0
    for char in s:
        try:
            val = int(char)
            total += val
        except:
            raise ValueError("string contained a character")
    return total
```

如果遇到非数字就停止执行，并抛出自定义信息的异常

## ■ 遇到异常时如何处理?

- 例子：当遇到异常时，处理异常信息并手动抛出

```
def sum_digits(s):
    """ s is a non-empty string containing digits.
    Returns sum of all chars that are digits """
    total = 0
    for char in s:
        try:
            val = int(char)
            total += val
        except:
            raise ValueError("string contained a character")
    return total
```

如果遇到非数字就停止执行，并抛出自定义信息的异常

## ■ 课后练习

```
def pairwise_div(Lnum, Ldenom):
    """ Lnum 和 Ldenom 是非空 lists, 包含了相同长度的数字
    Returns 一个新的 list 它的每一个元素是 Lnum 和 Ldenom 每一个元素成对相除的结果
    抛出一个 ValueError 如果 Ldenom 包含了 0 元素 """
    # 你的代码
```

例子:

```
L1 = [4,5,6]
L2 = [1,2,3]
# print(pairwise_div(L1, L2)) # prints [4.0,2.5,2.0]
L1 = [4,5,6]
L2 = [1,0,3]
# print(pairwise_div(L1, L2)) # raises a ValueError
```

## ■ 断言（Assertions）——一种防御式编程工具

- 确保对于代码计算状态的假设是符合预期的
- 使用assert语句，当假设不被满足时，可以抛出一个AssertionError

```
assert <statement that should be true>, "message if not true"
```

- 一个好的防御式编程效果：
  - 不赋予程序员对意外情况的响应控制权
  - 当预期条件未满足时强制终止程序执行
  - 可用于函数输入参数检查，适用于代码任何需要验证的位置
  - 可验证函数输出结果，防止错误值传播
  - 通过快速失败（fail-fast）机制有效缩小错误定位范围

## ■ 断言（Assertions）——一种防御式编程工具

- 断言的使用例子：从一个非空字符串中把数字加起来

```
def sum_digits(s):
    """ s is a non-empty string containing digits.
    Returns sum of all chars that are digits """
    assert len(s) != 0, "s is empty"
    total = 0
    for char in s:
        try:
            val = int(char)
            total += val
        except:
            raise ValueError("string contained a character")
```

当条件不满足时就停止执行，同时输出错误信息

## ■ 断言（Assertions）——一种防御式编程工具

- 断言的使用例子：从两个非空的列表中逐个元素相除

```
def pairwise_div(Lnum, Ldenom):
    """ Lnum and Ldenom are non-empty lists of equal lengths containing numbers
    Returns a new list whose elements are the pairwise division of Lnum and Ldenom.
    Raise a ValueError if Ldenom contains 0. """

    # 防御性编程断言（符合函数要求）
    assert len(Lnum) == len(Ldenom) and len(Lnum) > 0, "输入列表必须非空且长度相等"

    # 核心逻辑
    result = []
    for numerator, denominator in zip(Lnum, Ldenom):
        if denominator == 0:
            raise ValueError("分母不能为0")
        result.append(numerator / denominator)
    return result
```

## ■ 断言（Assertions）——一种防御式编程工具

- 断言的使用例子：学生成绩计算
  - 我们有一门课的成绩list，每个元素是一个学生的成绩，包括姓、名和成绩列表：

```
test_grades = [['peter', 'parker'], [80.0, 70.0, 85.0]],
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

  - 创建的一个新的list，将每个学生的平均成绩加在最后：

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
 [['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]
```

## ■ 断言（Assertions）——一种防御式编程工具

- 断言的使用例子：学生成绩计算
  - 一个实现的方式：

```
def get_stats(class_list):
    new_stats = []
    for stu in class_list:
        new_stats.append([stu[0], stu[1], avg(stu[1])])
    return new_stats


def avg(grades):
    return sum(grades)/len(grades)
```

## ■ 断言（Assertions）——一种防御式编程工具

- 断言的使用例子：学生成绩计算
  - 如果某个学生没有成绩，计算avg时会出错：

```
test_grades = [[['peter', 'parker'], [10.0,55.0,85.0]],
               [['bruce','wayne'], [10.0,80.0,75.0]],
               [['captain','america'], [80.0,10.0,96.0]],
               [['deadpool'], []]]

ZeroDivisionError: float division by zero
```

## ■ 断言（Assertions）——一种防御式编程工具

- 断言的使用例子：学生成绩计算
  - 更好的处理错误的方式：打印出错误原因

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
```

```
warning: no grades data
[[['peter', 'parker'], [10.0, 55.0, 85.0], 50.0],
[['bruce', 'wayne'], [10.0, 80.0, 75.0], 55.0],
[['captain', 'america'], [80.0, 10.0, 96.0], 62.0],
[['deadpool'], [], None]]
```

## ■ 断言（Assertions）——一种防御式编程工具

- 断言的使用例子：学生成绩计算
  - 更好的处理错误的方式：修改异常处理的逻辑（没成绩0分）

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
        return 0.0
```

```
warning: no grades data
[[['peter', 'parker'], [10.0, 55.0, 85.0], 50.0],
[['bruce', 'wayne'], [10.0, 80.0, 75.0], 55.0],
[['captain', 'america'], [80.0, 10.0, 96.0], 62.0],
[['deadpool'], [], 0.0]]
```

## ■ 断言（Assertions）——一种防御式编程工具

- 断言的使用例子：学生成绩计算
  - 更好的处理错误的方式：设置断言，没有满足就停止执行

```
def avg(grades):
    assert len(grades) != 0, 'no grades data'
    return sum(grades)/len(grades)
```

如果有空list，就抛出AssertionError，打印出错误信息，停止执行

## ■ 断言（Assertions）与异常（Exceptions）的对比

- 目标都是在bug出现的时候立刻发现它并指出其位置
- 异常提供了一种处理不满足预期输入的方法：
  - 在不需要停止代码执行时使用
  - 如果用户提供了不满足要求的输入就抛出异常
- 使用断言的场景：
  - 作为测试的补充功能
  - 检查参数的类型是否满足
  - 检查返回值的约束条件是否满足
  - 检查执行过程中是否有违反约束的现象

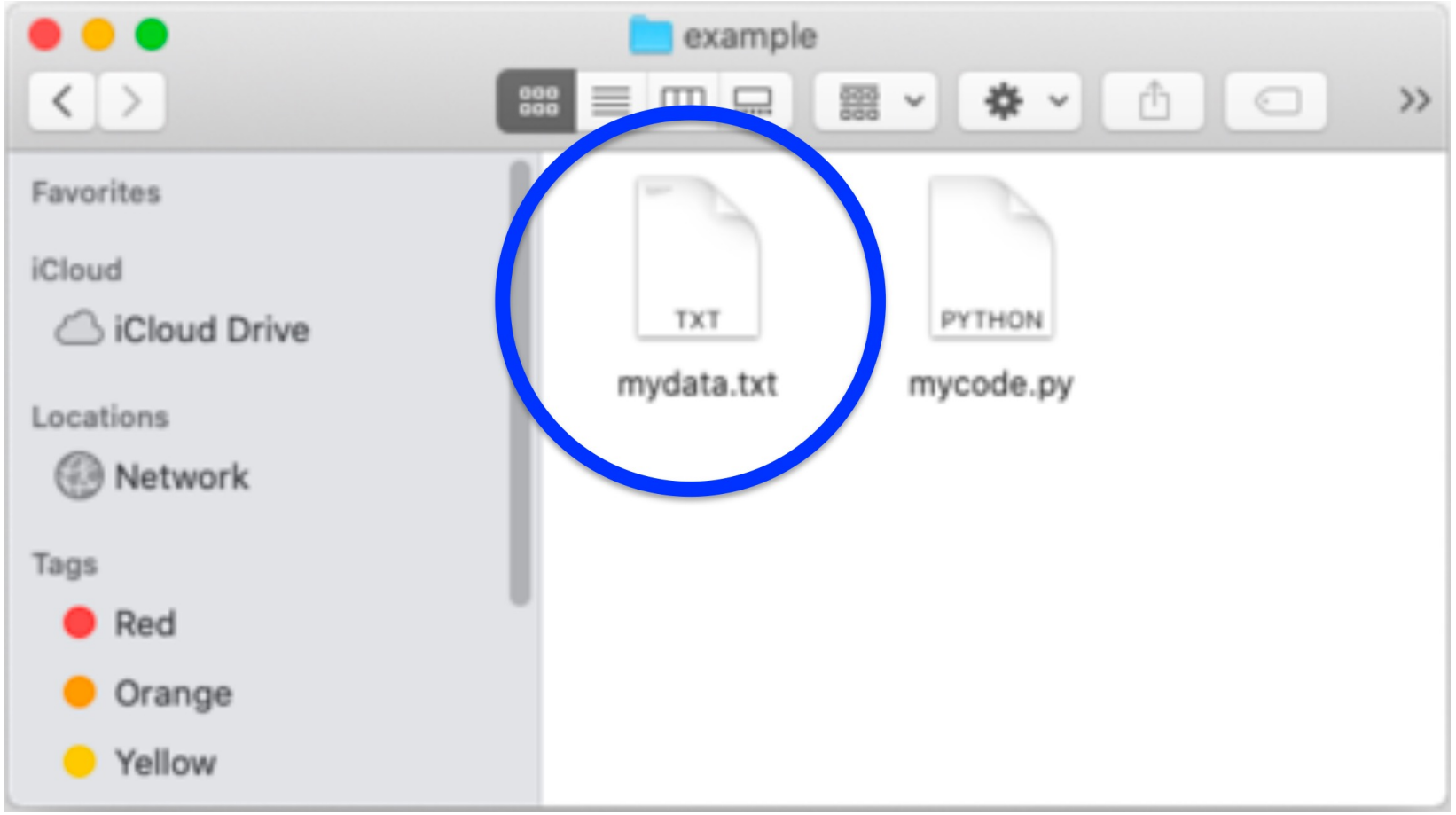
## 获取数据的方法

- 将数据直接放在程序里：变量赋值
- 让用户输入数据：输入框交互
- 随机化生成数据：random
- 从外部资源获取：存在文件中，后续读出来

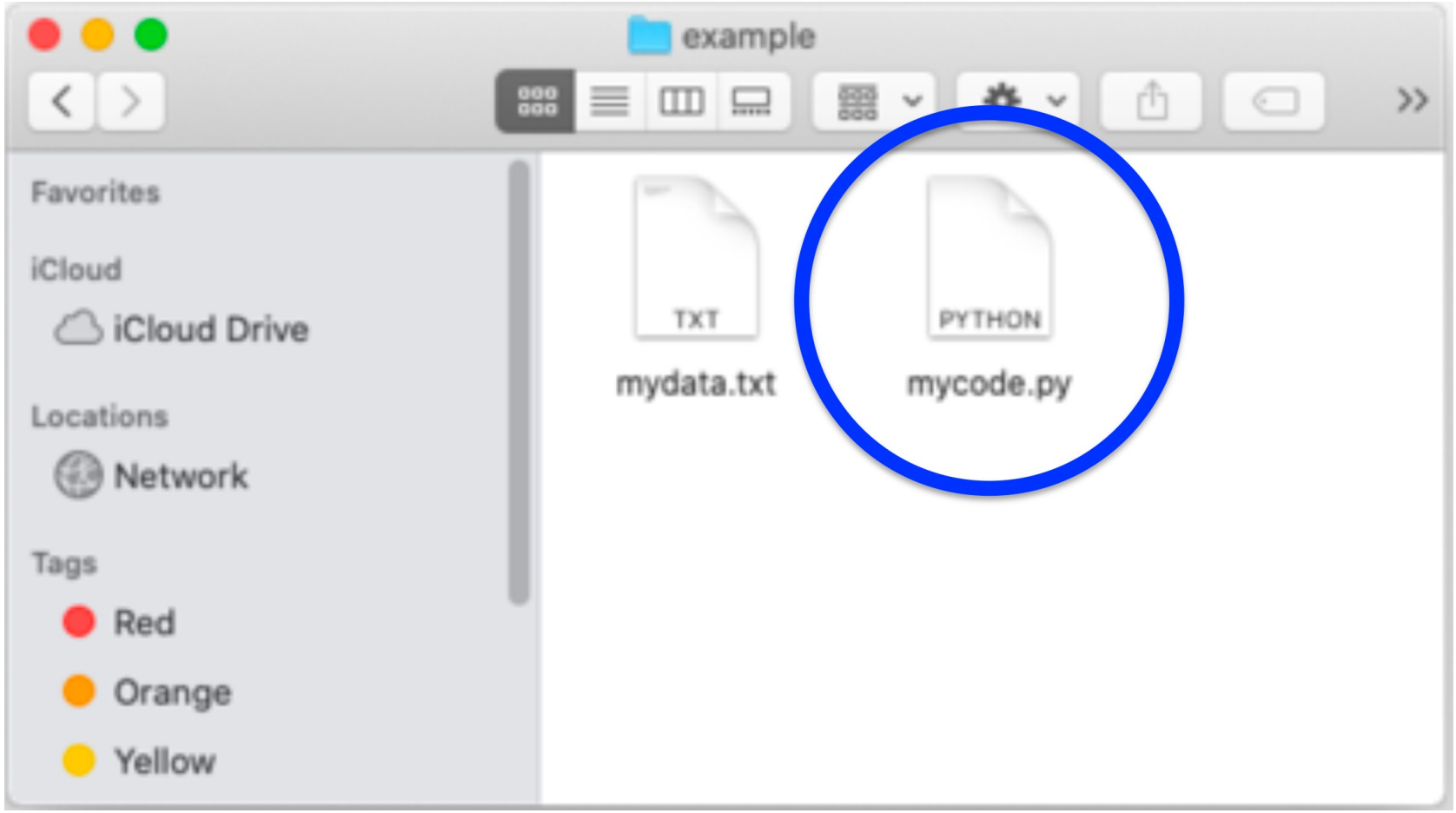
## 从文件中获取数据

- 文件是由一系列字节组成的
- 大量字节通过某种结构组成文件的格式：
  - TXT：由大量字符组成
  - JPEG：通过编码一副图片的结构信息形成
  - MP3: 通过编码一段音乐的音频信息形成
  - …

- 如何用Python处理文件



- 如何用Python处理文件



- 如何用Python处理文件

```
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass.
- Max Ehrmann "Desiderata"
```

## ■ 如何用Python处理文件

```
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass.
- Max Ehrmann "Desiderata"
```

```
file = open('mydata.txt')
for line in file:
    print(line)
```

打开文件，获取文件的访问控制权

## ■ 如何用Python处理文件

```
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass.
- Max Ehrmann "Desiderata"
```

```
file = open('mydata.txt')
for line in file:
    print(line)
```

每次从文件中读取一行内容

## ■ 如何用Python处理文件

```
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass.
- Max Ehrmann "Desiderata"
```

```
file = open('mydata.txt')
for line in file:
    print(line)
```

输出到控制台该行内容

**console**

```
Neither be cynical about love;
```

## ■ 如何用Python处理文件

```
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass.
- Max Ehrmann "Desiderata"
```

```
file = open('mydata.txt')
for line in file:
    print(line)
```

直到把所有行处理完

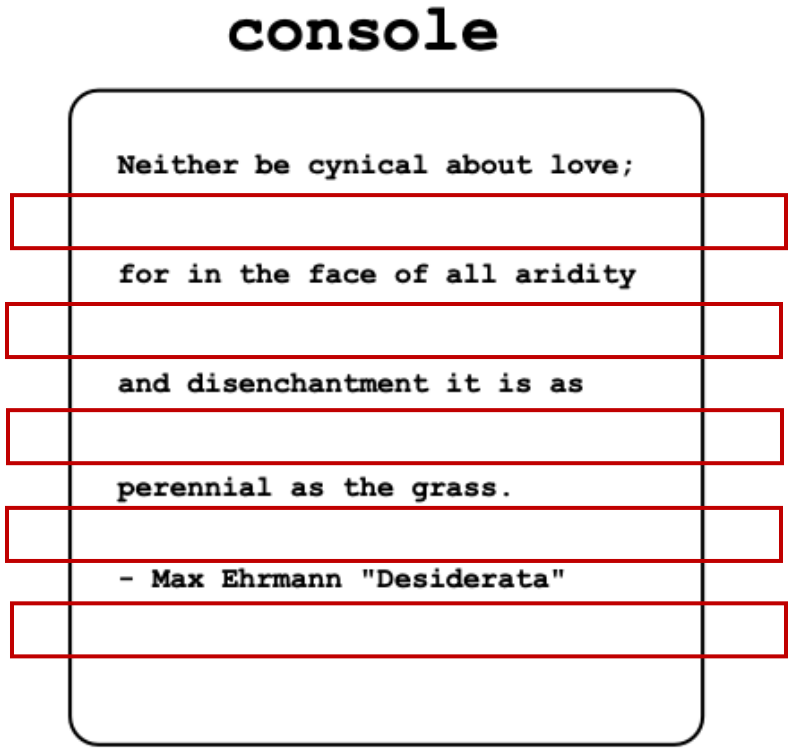
**console**

```
Neither be cynical about love;

for in the face of all aridity

and disenchantment it is as

perennial as the grass.

- Max Ehrmann "Desiderata"
```

## ■ 如何用Python处理文件

- 读取的每一行内容包含了换行符：
- line:  `'perennial as the grass.\n'`

```
file = open('mydata.txt')
for line in file:
    print(line)
```



## 如何用Python处理文件

- 分析以下代码的效果

```
f = open('mydata.txt')
for line in f:
    print(line.strip())
    print('-----')
for line in f:
    print(line.strip())
```

**Option A**

```
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass
- Max Ehrmann "Desiderata"
-----
```

**Option B**

```
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass
- Max Ehrmann "Desiderata"
-----
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass
- Max Ehrmann "Desiderata"
```

- 如何用Python处理文件
  - 分析以下代码的效果

```
f = open('mydata.txt')
for line in f:
    print(line.strip())
    print('-----')
for line in f:
    print(line.strip())
```

**Option A**

```
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass
- Max Ehrmann "Desiderata"
-----
```

文件对象在第一次for循环之后没有被重置

文件对象已经完全读取结束，指针已走完

不要读取同一个文件两次

- 如何用Python处理文件
  - 分析以下代码的效果

```
f = open('mydata.txt')
next(f)
for line in f:
    print(line.strip())
```

**Option A**

```
Neither be cynical about love;
for in the face of all aridity
and disenchantment it is as
perennial as the grass
- Max Ehrmann "Desiderata"
```

**Option B**

```
for in the face of all aridity
and disenchantment it is as
perennial as the grass
- Max Ehrmann "Desiderata"
```

## 如何用Python处理文件

- 分析以下代码的效果

```
f = open('mydata.txt')
next(f)
for line in f:
    print(line.strip())
```

next对文件操作时会跳过一行内容

同时返回这一行内容

**Option B**

```
for in the face of all aridity
and disenchantment it is as
perennial as the grass
- Max Ehrmann "Desiderata"
```

## ■ 如何用Python处理文件

- 更好的文件处理写法

```
with open('mydata.txt') as f:
    for line in file:
        line = line.strip()
        print(line)
```

with 让 python 知道什么时候可以对文件进行关闭

Python的早期版本没有垃圾回收机制，文件会一直处于打开状态知道程序结束，浪费大量资源

- Python处理csv文件——数据科学的日常

**dataset.csv**

```
Kenya,100,50
Malaysia,50,100

...

Turkey,20,20
Spain,95,95
```

```
with open('dataset.csv') as f:
    for line in file:
        line = line.strip()
        values = line.split(',')
        print(values[1])
```

获取的一行内容：**'Kenya,100,50\n'**

- Python处理csv文件——数据科学的日常

**dataset.csv**

```
Kenya,100,50
Malaysia,50,100

...

Turkey,20,20
Spain,95,95
```

```
with open('dataset.csv') as f:
    for line in file:
        line = line.strip()
        values = line.split(',')
        print(values[1])
```

line的内容：**'Kenya,100,50'**

## Python处理csv文件——数据科学的日常

**dataset.csv**

```
Kenya,100,50
Malaysia,50,100

...

Turkey,20,20
Spain,95,95
```

```
with open('dataset.csv') as f:
    for line in file:
        line = line.strip()
        values = line.split(',')
        print(values[1])
```

values的内容：

| 'Kenya' | '100' | '50' |
|---|---|---|

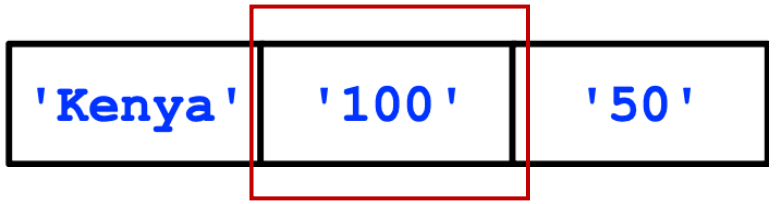
## Python处理csv文件——数据科学的日常

**dataset.csv**

```
Kenya,100,50
Malaysia,50,100

...

Turkey,20,20
Spain,95,95
```

```
with open('dataset.csv') as f:
    for line in file:
        line = line.strip()
        values = line.split(',')
        print(values[1])
```

输出的内容：



## Python处理csv文件——数据科学的日常

**dataset.csv**

```
Kenya,100,50
Malaysia,50,100

...

Turkey,20,20
Spain,95,95
```

更方便的方式：csv 包

```
import csv
with open('dataset.csv') as f:
    reader = csv.reader(f)
    for values in reader:
        print(values[1])
```

## Python处理csv文件——数据科学的日常

- Python中，使用open( )打开文件时，如果不指定mode，默认用只读模式打开文件

```
import csv
with open('dataset.csv', 'r') as f:
    reader = csv.reader(f)
    for values in reader:
        print(values[1])
```

如果文件不存在，抛出FileNotFoundError

## Python处理csv文件——数据科学的日常

- 读取csv文件再写入新的csv文件中

```
import csv
with open('dataset.csv', 'r') as f:
    reader = csv.reader(f)
    data = [row for row in reader]
```

```
with open('output.csv', 'w') as f:
    writer = csv.writer(f)
    writer.writerows(data)
```

## 文件相关的其他用法

- Python中对于文件路径的处理

```
from pathlib import Path

file_path = Path.home() / "my_folder" / "my_file.txt"
print(file_path) # /path/to/your/home/my_folder/my_file.txt
```

## 文件相关的其他用法

- Python中对于文件路径的处理

```
print(file_path.exists()) # False

print(file_path.name) # my_file.txt

print(file_path.parent.name) # my_folder
```

## 文件相关的其他用法

- Python中对于文件路径的处理

```
from pathlib import Path


new_dir = Path.home() / "my_folder"
new_dir.mkdir()
```

## 文件相关的其他用法

- Python中对于文件路径的处理

```
file1 = new_dir / "file1.txt"
file2 = new_dir / "file2.txt"
image1 = new_dir / "image1.png"


file1.touch() # create empty text file
file2.touch()
image1.touch() # create empty image file
```

## 文件相关的其他用法

- Python中对于文件路径的处理

```
file1.unlink() # delete text file

import shutil

shutil.rmtree(new_dir) # remove the whole folder
```

## 文件相关的其他用法

- Python中对于文件路径的处理

```
documents_dir = Path.cwd() / "practice_files" / "documents”
images_dir = Path.home() / "images”
```

递归查找指定模式路径

```
for path in documents_dir.rglob("*.*"):
    if path.suffix.lower() in [".png", ".jpg", ".gif"]:
        path.replace(images_dir / path.name)
```

将path路径下的图片移到image_dir下

## 文件相关的其他用法

- Python读写特定路径文件

```
from pathlib import Path


starships = ["Discovery\n", "Enterprise\n", "Defiant\n",
"Voyager"]
file_path = Path.home() / "starships.txt"
with file_path.open(mode="w", encoding="utf-8") as file:
    file.writelines(starships)
```

## 文件相关的其他用法

- Python读写特定路径文件

```
with file_path.open(mode="r", encoding="utf-8") as file:
    for starship in file.readlines():
        print(starship, end="")
```

```
Discovery
Enterprise
Defiant
Voyager
```

## 文件相关的其他用法

- Python读写特定路径文件

```
with file_path.open(mode="r", encoding="utf-8") as file:
    for starship in file.readlines():
        if starship.startswith("D"):
            print(starship, end="")
```

```
Discovery
Defiant
```

## 文件相关的其他用法

- Python读写CSV文件

```
import csv

from pathlib import Path


numbers = [
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15],
]
file_path = Path.home() / "numbers.csv"
```

## 文件相关的其他用法

- Python读写CSV文件

```
with file_path.open(mode="w", encoding="utf-8") as file:
    writer = csv.writer(file)
    writer.writerows(numbers) # write each row into csv
```

## 文件相关的其他用法

- Python读写CSV文件

```
numbers = []
with file_path.open(mode="r", encoding="utf-8") as file:
    reader = csv.reader(file)
    for row in reader:
        int_row = [int(num) for num in row]
        numbers.append(int_row)
print(numbers) # same as numbers before
```

## 文件相关的其他用法

- Python读写CSV文件

把字典写入csv

```
favorite_colors = [
    {"name": "Joe", "favorite_color": "blue"},
    {"name": "Anne", "favorite_color": "green"},
    {"name": "Bailey", "favorite_color": "red"},
]
```

```
file_path = Path.home() / "favorite_colors.csv"
```

## 文件相关的其他用法

- Python读写CSV文件

```
with file_path.open(mode="w", encoding="utf-8") as file:
    writer = csv.DictWriter(
        file, fieldnames=["name", "favorite_color"])
    writer.writeheader() # write keys into csv first line
    writer.writerows(favorite_colors) # values as rows
```

## 文件相关的其他用法

- Python读写CSV文件

```
favorite_colors = []

with file_path.open(mode="r", encoding="utf-8") as file:
    reader = csv.DictReader(file)
    for row in reader:
        favorite_colors.append(row)

print(favorite_colors) # each dictionary as an item in list
```

# Reading and QA Time

# See you next week !