Python程序设计与实践

第四课：抽象与函数

2025. 3

Today

- 抽象化思维
- 函数的编写
- 函数作为对象
- 函数的其他用法

■ 抽象化的一个例子：手机

- 对用户来说是一个黑盒子，人们只需要关注：
  - ➢ 它的输入：触屏点击
  - ➢ 它的输出：执行功能
  - ➢ 输出与输入的映射关系（无需知道内部原理）
  - ➢ 实现过程是模糊的

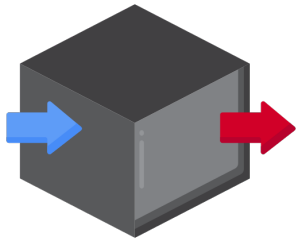
- 抽象化使复杂问题分解为简单任务
  - 一只手机有一百多个组成部分
  - 每个部分由不同的企业设计和生产
  - 每个部分的制造商需要知道它与其他部分的交互关系
  - 每个部分的制造商可以独立解决他们的子问题
  - 对于硬件和软件部分都是一样的逻辑

Functions

- 抽象化（abstraction）：
  - 将功能封装成“黑盒子”，只需了解它的功能、输入、输出
- 分解（decomposition）：
  - 将复杂任务分为多个组成部分，每个部分专注于解决一个子问题
- 抽象化和分解让复杂的问题简单化，让不同的角色专注于不同的子问题，促进分工合作，提升用户体验
- 我们需要将抽象化和分解应用到编程中

■ 抽象化

- 在编程中，考虑让一段代码成为一个“黑盒子”，从而
  - ➢ 把繁杂的代码细节对用户隐藏起来（对用户）
  - ➢ 在其他代码中重复使用这段代码的功能（对开发者）
- 编程人员需要编写细节后，设计对外的交互接口
- 使用人员往往不需要看到这些细节



■ 抽象化

- 抽象化的实现需要用到函数（function）
- 一个函数让我们获取到一段代码的能力而无需知道代码细节
- 一旦创建了函数，它将实现从某种输入到某种输出的能力，同时隐藏它是如何实现这种过程的细节
- 我们之前用到了一些函数：
  - `max(1, 4)`
  - `abs(-3)`
  - `len(“happy birthday”)`

■ 抽象化

- 一个创建好的函数有它自己的规格（specifications），展示在python说明文档（docstrings）中
- 一般定义一个新函数，要给出它的说明文档
- 通过.__doc__可以获取一个函数的说明文档

```
def multiply_numbers(a, b):
    """
    Multiplies two numbers and returns the result.
    Args:
        a (int): The first number.
        b (int): The second number.
    Returns: int: The product of a and b.
    """
    return a * b
```

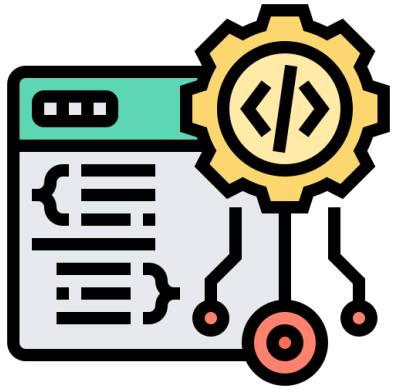
```
>>> def multiply_numbers(a, b):
...     """
...     Multiplies two numbers and returns the result.
...     Args:
...         a (int): The first number.
...         b (int): The second number.
...     Returns: int: The product of a and b.
...     """
...     return a * b
...
>>> print(multiply_numbers.__doc__)

    Multiplies two numbers and returns the result.
    Args:
        a (int): The first number.
        b (int): The second number.
    Returns: int: The product of a and b.
```

通过分解创建逻辑结构

- 采用抽象化的函数，将代码分为若干个模块
- 模块可以：
  - 将代码切割为逻辑片段
  - 让代码有效组织
  - 让代码连贯易读
- 创建函数是实现任务分解的一种手段（另一种是类）
- 进而让复杂的问题由一系列简单的功能组成

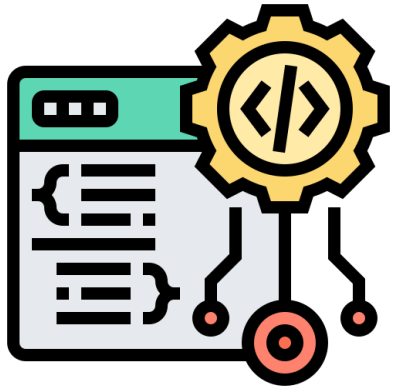
■函数

- 一段可以被重复使用的代码
- 实现了一系列计算步骤
- 可以输入任何满足要求的数据



■函数

- 定义一个函数实际在告诉Python，一段代码现在需要存在内存中
- 函数只有在运行（被调用）时才是有实际作用的
- 你只需要编写一次该函数，便可以运行它无数次



■函数的一些特性

- 函数具有一个名字：一个变量被绑定到了一个函数对象上
- 函数具有一些参数：输入的数据要求，0个或多个
- 函数具有一个说明文档：可选但是建议写，用三个隐含包裹，提供了函数的规格说明（功能、输入、输出等），用`__doc__`可以获取
- 函数具有一个函数体：一系列代码指令，在函数被调用时执行
- 函数通常需要返回一些数据：用`return`关键字

■如何编写一个函数

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

■如何思考怎么编写一个函数

- 先想清楚函数要实现的功能是什么
- 例如：提供一个整数i，判断它是否为偶数
- 用这些信息来写函数名和它的说明文档

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
```

■如何思考怎么编写一个函数

- 再思考如何解决这个问题
- 针对这个例子：可以看这个数字除以2之后余数是否为0
- 再思考你要让函数返回一个什么值

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

■如何思考怎么编写一个函数

- 最后考虑下怎么让这个函数更简洁
- 针对这个例子：i%2==0返回的就是True或False

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    return i%2 == 0
```

■如何调用一个函数

- 只需要提供函数名，并提供符合要求的参数即可

```
is_even(3)
is_even(8)
```

- 调用后的结果就是`return`出来的值

■当调用一个函数时，发生了什么

- Python将函数的参数变量替换为输入的值
- 然后用这个值去执行函数体中的代码
- 最后用计算的结果代替函数调用的代码

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

i 被 3 代替

执行3%2 == 0 得到False

```
is_even(3)
```

把调用函数的表达式用False代替

■编写一段代码满足以下规格:

```
def div_by(n, d):
    """
    n 和 d 为整数且 > 0
    返回 True 如果 d 可以整除 n , 否则返回False
    """
```

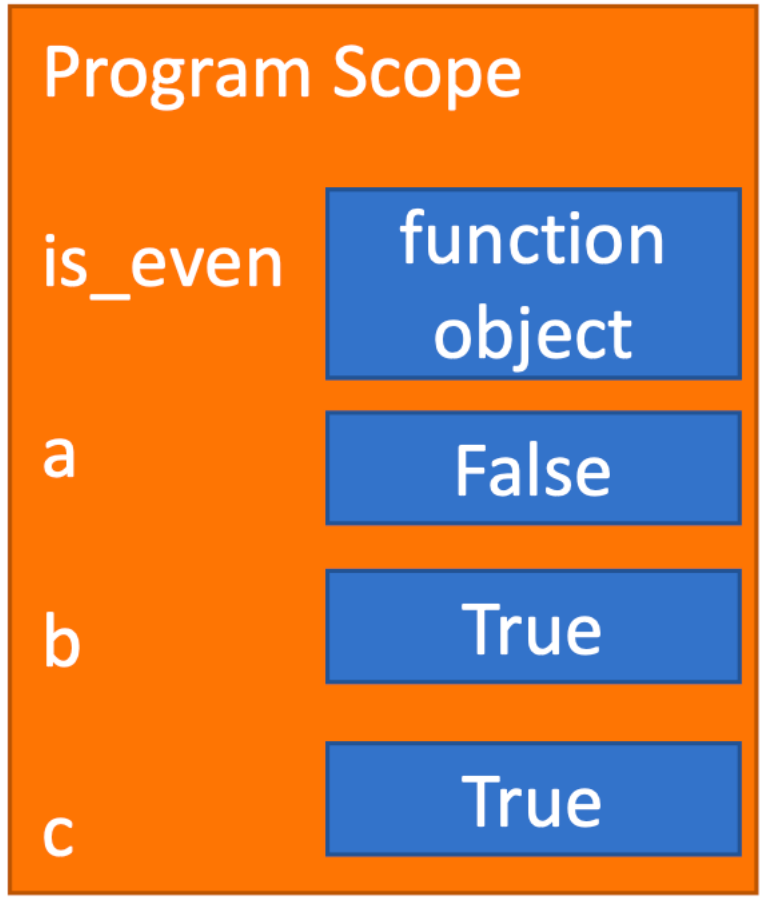
测试你的代码:

- n = 10 and d = 3
- n = 195 and d = 13

■直观感受函数在程序中的角色

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


a = is_even(3)
b = is_even(10)
c = is_even(123456)
```

均在主程序中运行



■将函数嵌入到代码中

- 函数调用会被返回的值替换掉

```
print("Numbers between 1 and 10: even or odd")
for i in range(1,10):
    if is_even(i):
        print(i, "even")
    else:
        print(i, "odd")
```

替换为True或False

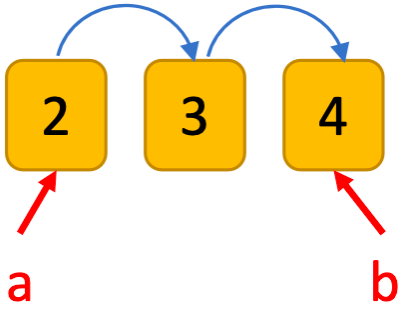
■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 我们需要定义一个函数实现这个功能
  - 输入是什么？a和b的值
  - 输出是什么？sum_of_odds

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

■一个编写函数的例子

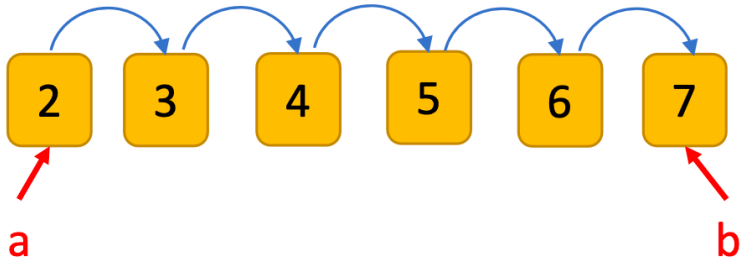
- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 不要马上开始写代码，先仔细分析问题
- 不妨先举几个例子推算一下
- 假如a=2，b=4会怎样?
- sum_of_odds为3




```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 不要马上开始写代码，先仔细分析问题
- 不妨先举几个例子推算一下
- 假如a=2，b=7会怎样?
- sum_of_odds为15




```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 不要马上开始写代码，先仔细分析问题
- 不妨降低一下问题的难度
- 一个类似但简单的问题是：
  - 将两个数字之间的所有数字相加

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 一个类似但简单的问题是：
  - 将两个数字之间的所有数字相加
  - 要遍历一个范围内的每一个数字
  - 需要用到循环，`for` 或者 `while`都行
  - 我们试试用循环实现解决这个问题

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 一个类似但简单的问题是：
  - 将两个数字之间的所有数字相加

先把循环结构写出来

for 循环

```
def sum_odd(a, b):
    for i in range(a, b):
        # do something
    return sum_of_odds
```

while 循环

```
def sum_odd(a, b):
    i = a
    while i <= b:
        # do something
        i += 1
    return sum_of_odds
```

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 一个类似但简单的问题是：
  - 将两个数字之间的所有数字相加

再把累加过程写出来

for 循环

```
def sum_odd(a, b):
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds
```

while 循环

```
def sum_odd(a, b):
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds
```

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 一个类似但简单的问题是：
  - 将两个数字之间的所有数字相加

加一个初始化的操作

for 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds
```

while 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds
```

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 一个类似但简单的问题是：
  - 将两个数字之间的所有数字相加

打印出函数结果测试一下

for 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
    return sum_of_odds



print(sum_odd(2,4))
```

while 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds



print(sum_odd(2,4))
```

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 一个类似但简单的问题是：
  - 将两个数字之间的所有数字相加

也可以在函数中添加打印来测试

for 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b):
        sum_of_odds += i
        print(i, sum_of_odds)
    return sum_of_odds

print(sum_odd(2,4))
```

```
2 2
3 5
```

5

这里少了结尾数字

while 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        print(i, sum_of_odds)
        i += 1
    return sum_of_odds

print(sum_odd(2,4))
```

```
2 2
3 5
4 9
```

9

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 一个类似但简单的问题是：
  - 将两个数字之间的所有数字相加

修改代码让它正确执行预期逻辑

for 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        sum_of_odds += i
        print(i, sum_of_odds)
    return sum_of_odds


print(sum_odd(2,4))
```

9

while 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        print(i, sum_of_odds)
        i += 1
    return sum_of_odds

print(sum_odd(2,4))
```

```
2 2
3 5
4 9
```

9

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 现在我们可以实现对奇数的累加过程了

添加检测奇数的逻辑

for 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
            print(i, sum_of_odds)
    return sum_of_odds

print(sum_odd(2,4))
```

3

while 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        if i%2 == 1:
            sum_of_odds += i
            print(i, sum_of_odds)
        i += 1
    return sum_of_odds
print(sum_odd(2,4))
```

3

■一个编写函数的例子

- 假设我们希望实现将两个数字之间的所有奇数相加（包括两个数字）
- 现在我们可以实现对奇数的累加过程了

测试另一个例子看是否正确

for 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a, b+1):
        if i%2 == 1:
            sum_of_odds += i
    return sum_of_odds


print(sum_odd(2,7))
```

15

while 循环

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        if i%2 == 1:
            sum_of_odds += i
        i += 1
    return sum_of_odds

print(sum_odd(2,7))
```

15

■编写函数要注意的：

- 先别着急写代码，先对问题进行分析，明确输入输出，找几个例子试一试
- 可以先实现一个简化版的问题，一步步去达到目标
- 在实现的过程中多停下来测试是否按预期的逻辑执行
- 实现完多测试几个例子看是否完全正确，是否考虑了边界情况

■编写一段代码满足以下规格要求（经典问题—回文）

```
def is_palindrome(s):
    """ s is a string
    Returns True if s is a palindrome and False otherwise
    """
```

- 举例:
  - 如果s=“222”，返回True
  - 如果s=“wasitacaroracatisaw”，返回True
  - 如果s=“abc”，返回False

■函数的返回：

- 函数总是返回一些信息的，用return关键字

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even and False otherwise
    """
    return i%2 == 0
```

■函数的返回：

- 如果没有return会怎样

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even and False otherwise
    """
    i%2 == 0
```

- 如果没有使用return，Python会返回一个None值
- 如果调用这个函数并打印它的值，不会输出任何结果，也不会报错
- 相当于在最后加了一行 return None

- 自己试试以下代码，用.py文件执行，看看输出什么

```
def add(x,y):
    return x+y
def mult(x,y):
    print(x*y)
add(1,2)
print(add(2,3))
mult(3,4)
print(mult(4,5))
```

■对比一下 return 和 print

return	print
return只在函数中才有意义	print在函数内外都可以使用
在一次函数调用时只有一个return被执行	在一次函数调用时可以执行多次print
在执行return后，剩下的代码不会被执行	执行了print后还可以继续执行后续代码
return的结果可以替换调用函数的代码	print的结果只输出到终端中展示 print自己也有返回值None

■尝试修改以下代码让它按照说明文档执行

```
def is_triangular(n):
    """ n is an int > 0
    Returns True if n is triangular, i.e. equals a continued
    summation of natural numbers (1+2+3+...+k), False otherwise
    """
    total = 0
    for i in range(n):
        total += i
        if total == n:
            print(True)
    print(False)
```

■回顾下之前做二分查找平方根的例子，将它写成函数

```
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    ans = (high + low)/2.0
    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    print(ans, 'is close to the root of', x)
    return ans
```

初始化变量

循环搜索

如果猜的数字不够接近

根据猜的数字太小或太大
更新上限或下限

重新猜一个数字

打印信息，返回结果

- 回顾下之前二分查找平方根的例子，将它写成函数
  - 现在我们可以调用这个函数来计算不同数字的平方根了

```
print(bisection_root(4))
print(bisection_root(123))
```

■利用写好的平方根查找函数来实现复杂的逻辑

- 实现一个函数，打印出平方根接近10的数字个数

```
def count_nums_with_sqrt_close_to_10 (epsilon):
    """ epsilon is a positive number < 1
    Returns how many integers have a square root within epsilon of 10
    """
    count = 0
    n = 1
    while bisection_root(n)-10 < epsilon:
        if math.abs(bisection_root(n)-10) < epsilon:
            count += 1
        n += 1
    return count
```

■函数的作用域（function scope）

- Python在调用一个函数时在做些什么？
- Python在调用每一个函数时会创建一个临时的“小环境”
  - ➢ 这个小环境会使用赋予它的参数来执行程序
  - ➢ 执行完函数体的代码后会返回一个值
  - ➢ 这个小环境会在返回值后消失

■函数的作用域（function scope）

- 全局环境（Global Environment）：
  - 用户与Python交互的地方
  - Python程序启动的地方
- 调用函数相当于临时创建一个新环境
  - 这个环境是函数的作用域

■函数的作用域（function scope）

- 函数的作用域是变量名与数据对象的映射有效区域
- 在一个函数作用域内使用正式定义的参数进行计算
- 在一个函数作用域外使用实际传入的参数进行调用

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

正式定义的参数

函数的作用域

实际传入的参数，名称可以任意

全局的作用域

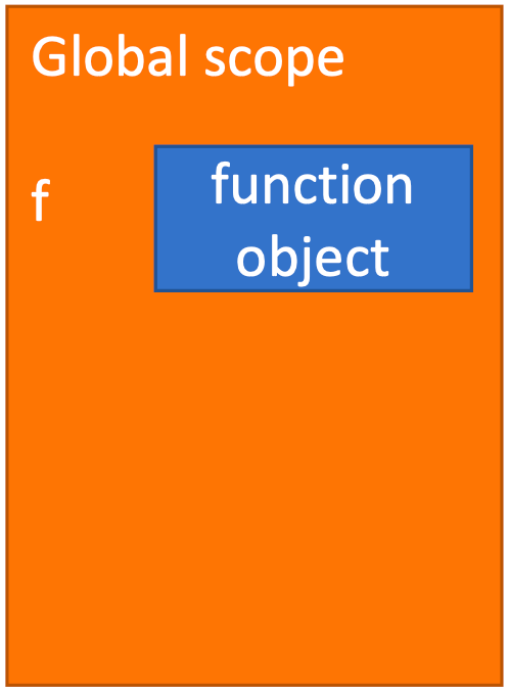
■函数的作用域（function scope）

- 当我们执行一个Python程序，是从全局作用域开始的
- 已经定义的函数会先加载到全局作用域中

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x
```

从这里开始执行

```
x = 3

z = f( x )
```

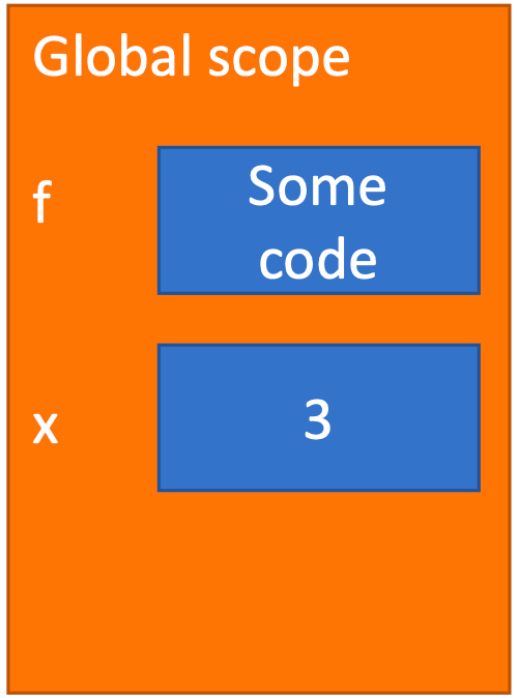


■函数的作用域（function scope）

- 当我们执行一个Python程序，是从全局作用域开始的
- 已经定义的函数会先加载到全局作用域中

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

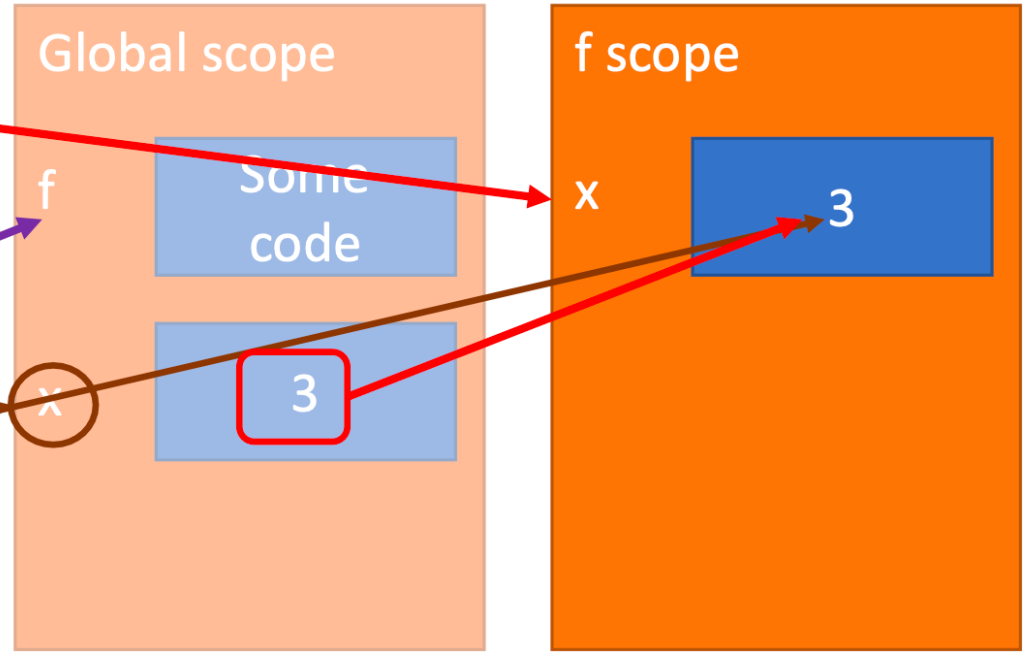
执行到这里



■函数的作用域（function scope）

- 当我们执行一个Python程序，是从全局作用域开始的
- 已经定义的函数会先加载到全局作用域中

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x


x = 3
z = f( x )
```



执行到函数时，进入到函数作用域

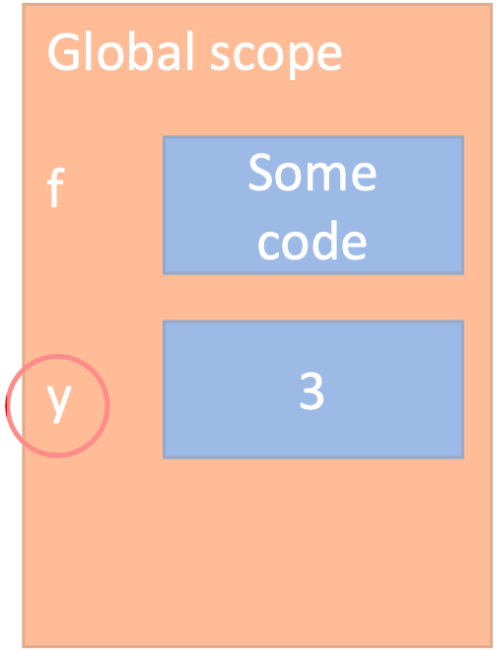
■函数的作用域（function scope）

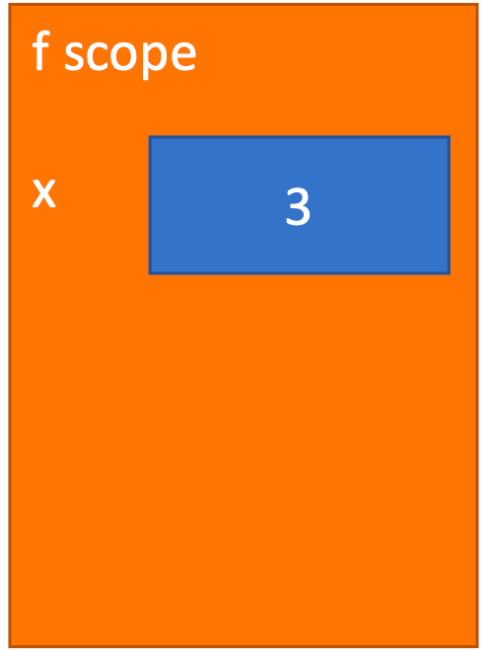
- 当我们执行一个Python程序，是从全局作用域开始的
- 已经定义的函数会先加载到全局作用域中

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x
```

```
y = 3
```

```
z = f( y )
```



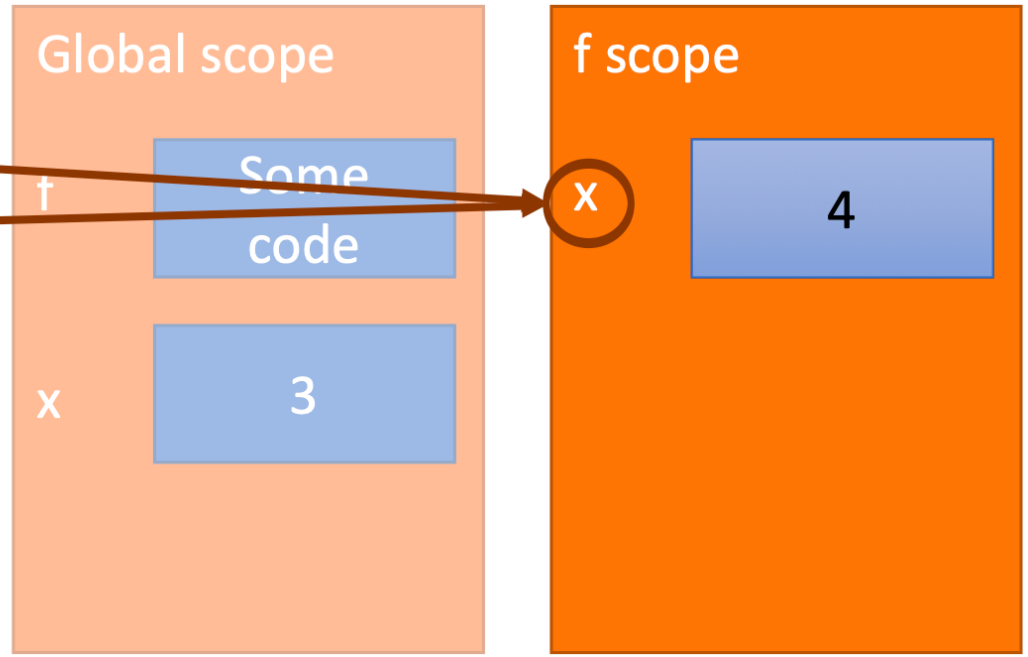


全局环境的变量名与函数无关，只有值是相关的

■函数的作用域（function scope）

- 当我们执行一个Python程序，是从全局作用域开始的
- 已经定义的函数会先加载到全局作用域中

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

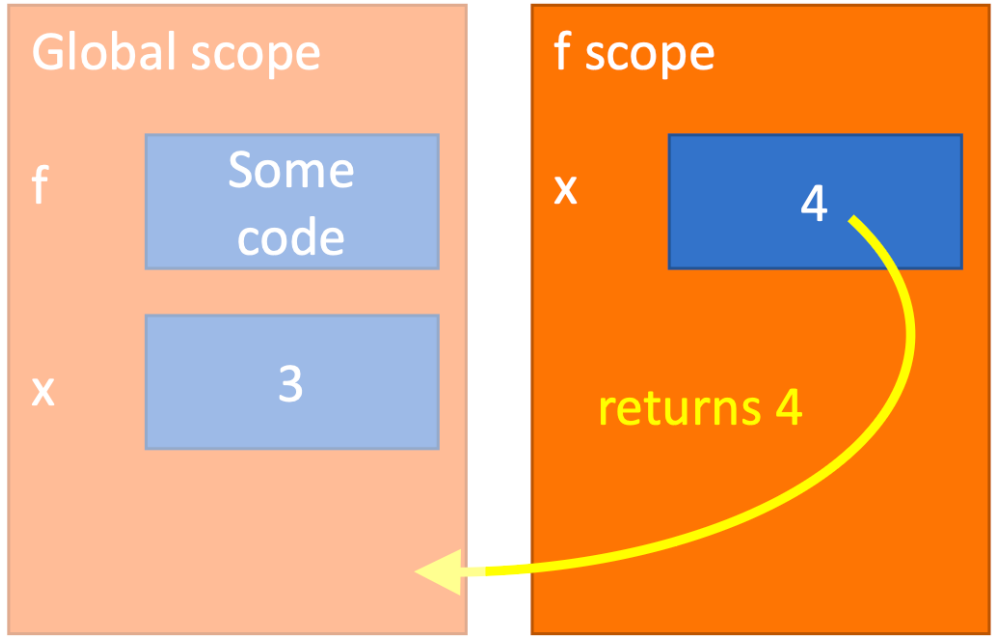


在函数作用域中执行代码并输出结果

■函数的作用域（function scope）

- 当我们执行一个Python程序，是从全局作用域开始的
- 已经定义的函数会先加载到全局作用域中

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



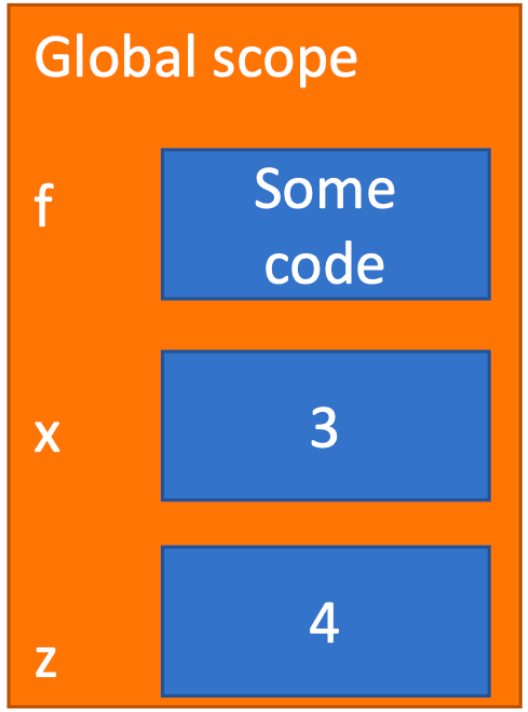
函数调用的返回值替换掉全局作用域中的调用位置

■函数的作用域（function scope）

- 当我们执行一个Python程序，是从全局作用域开始的
- 已经定义的函数会先加载到全局作用域中

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

这时候在全局作用域中打印x的值，输出的仍是3



■函数的作用域（function scope）

- 在一个函数内部，可以定义一个跟在外部同名的变量
- 在一个函数内部，可以访问在外部定义的变量
- 在一个函数内部，不能修改在外部定义的变量

```
def f(y):
    x = 1
    x += 1
    print(x)
x = 5
f(x)
print(x)
```

```
2
5
```

同名但作用域不同

```
def g(y):
    print(x)
    print(x+1)
x = 5
g(x)
print(x)
```

```
5
6
5
```

在函数作用域访问外部变量

```
def h(y):
    x += 1
x = 5
h(x)
print(x)
```

```
Error
Local variable 'x'
referenced before
assignment
```

在函数作用域
不能修改外部变量

■函数作为参数

- Python中的对象拥有类型

  ```
  int, float, str, Boolean, NoneType, function
  ```

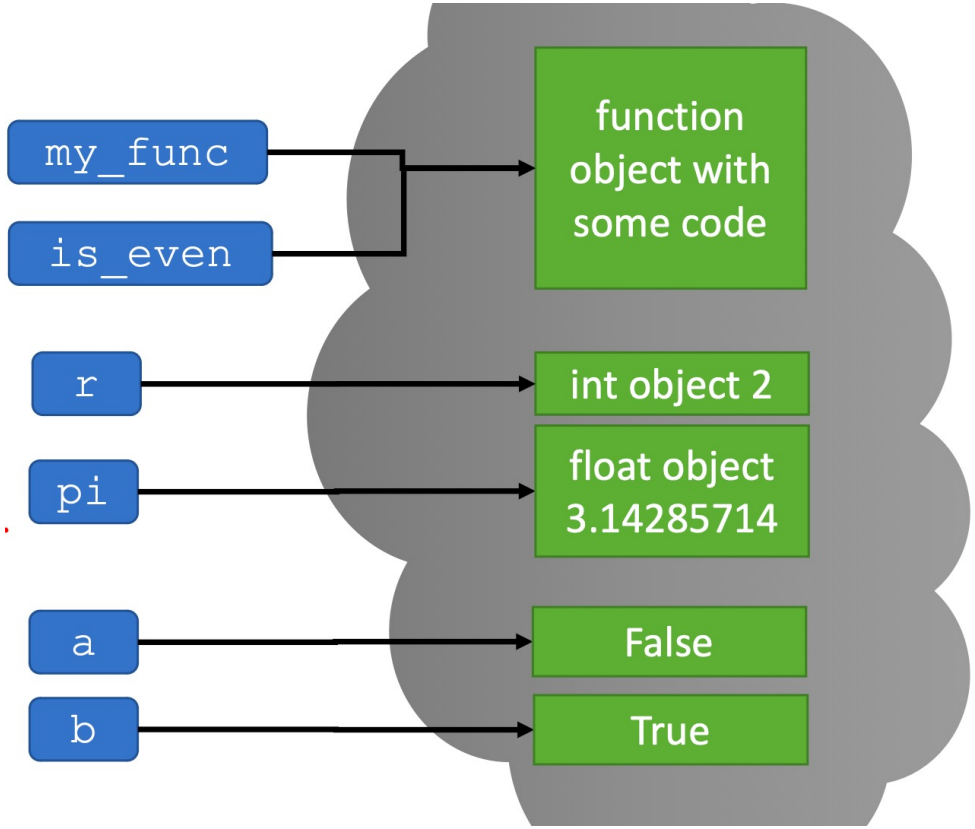
- 对象既可以作为参数也可以作为返回值
- 函数也是对象之一，与其他类型的对象拥有类似的特性
- 在Python中任何事物都是对象

■函数作为参数

```
def is_even(i):
    return i%2 == 0
r = 2
pi = 22/7
my_func = is_even
a = is_even(3)
b = my_func(4)
```

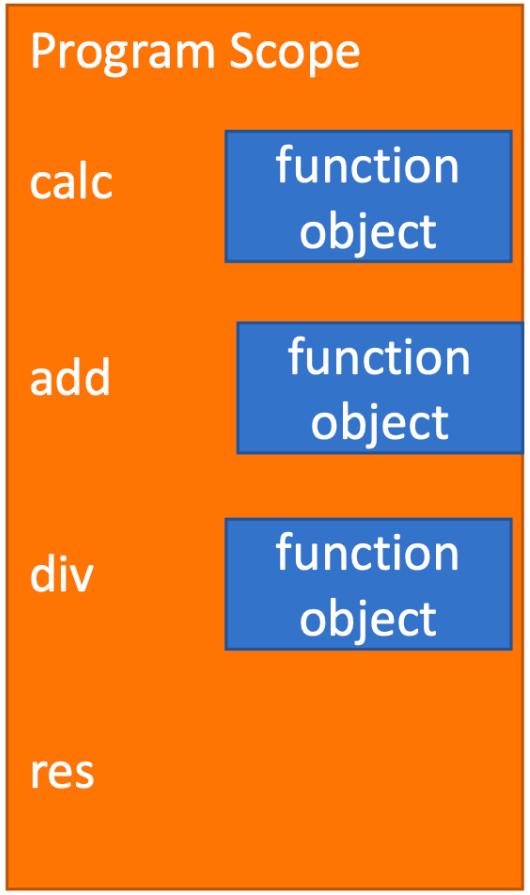
不是函数调用，
只是名字绑定

都是函数调用



■函数作为参数

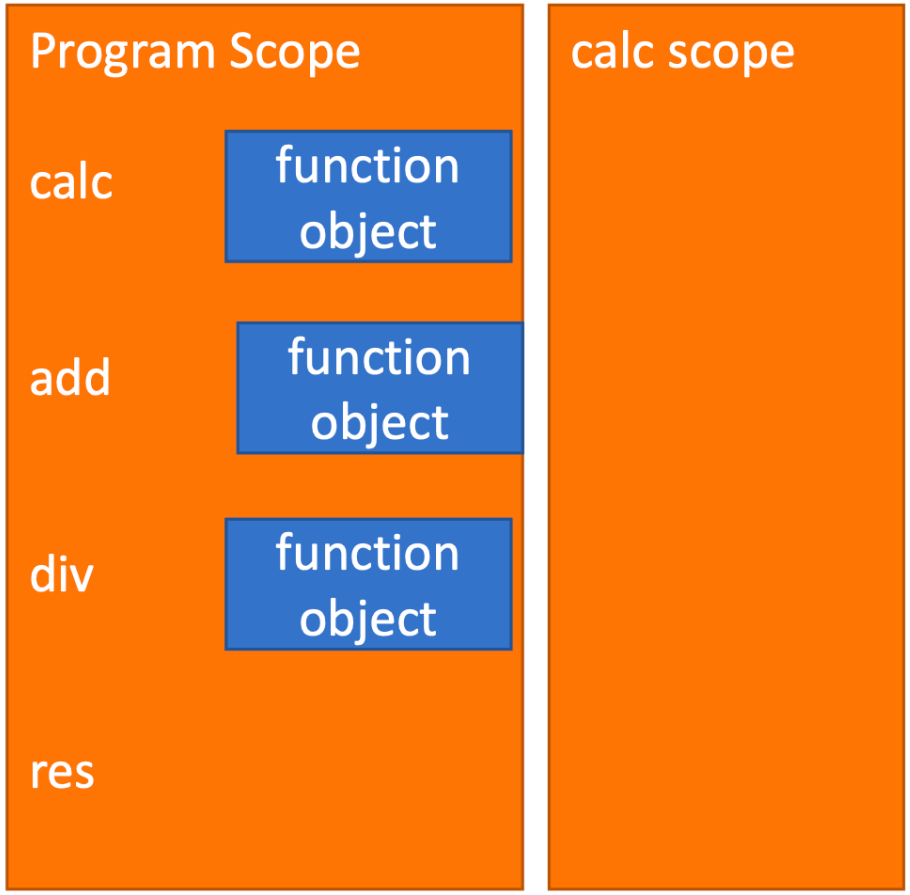
```
def calc(op, x, y):
    return op(x,y)

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

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

res = calc(add, 2, 3)
```



■函数作为参数

```
def calc(op, x, y):
    return op(x,y)

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

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

Res = calc(add, 2, 3)
```

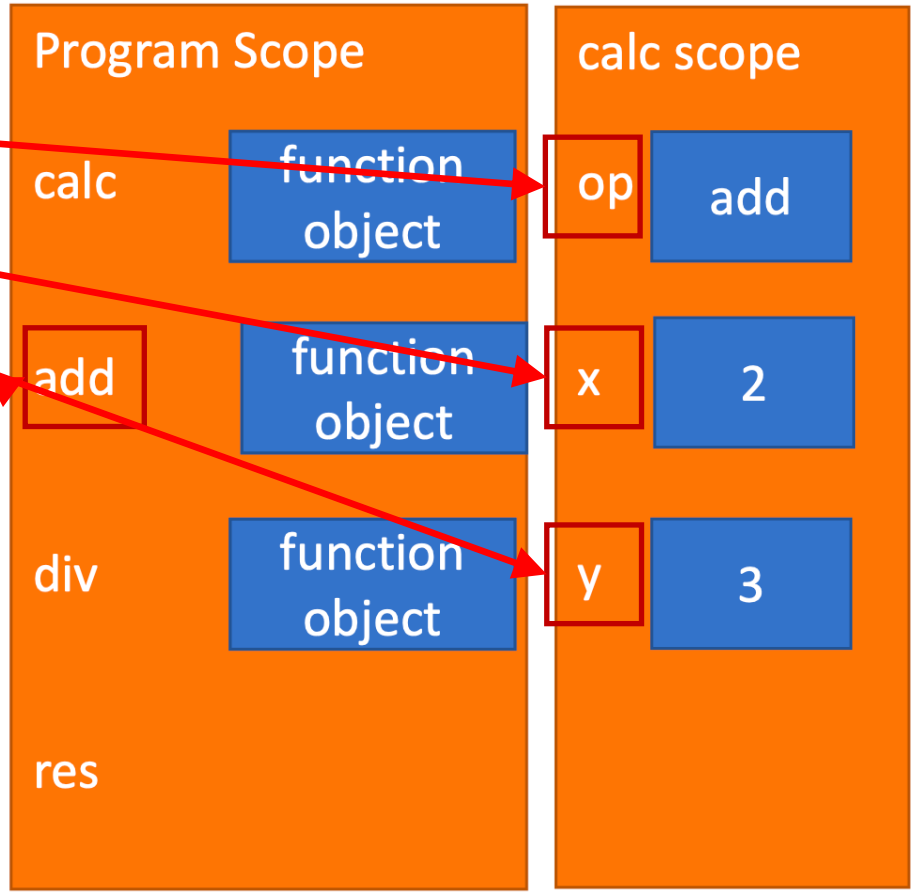
在全局作用域函数调用



■函数作为参数

```
def calc(op, x, y):
    return op(x,y)

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

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

Res = calc(add, 2, 3)
```

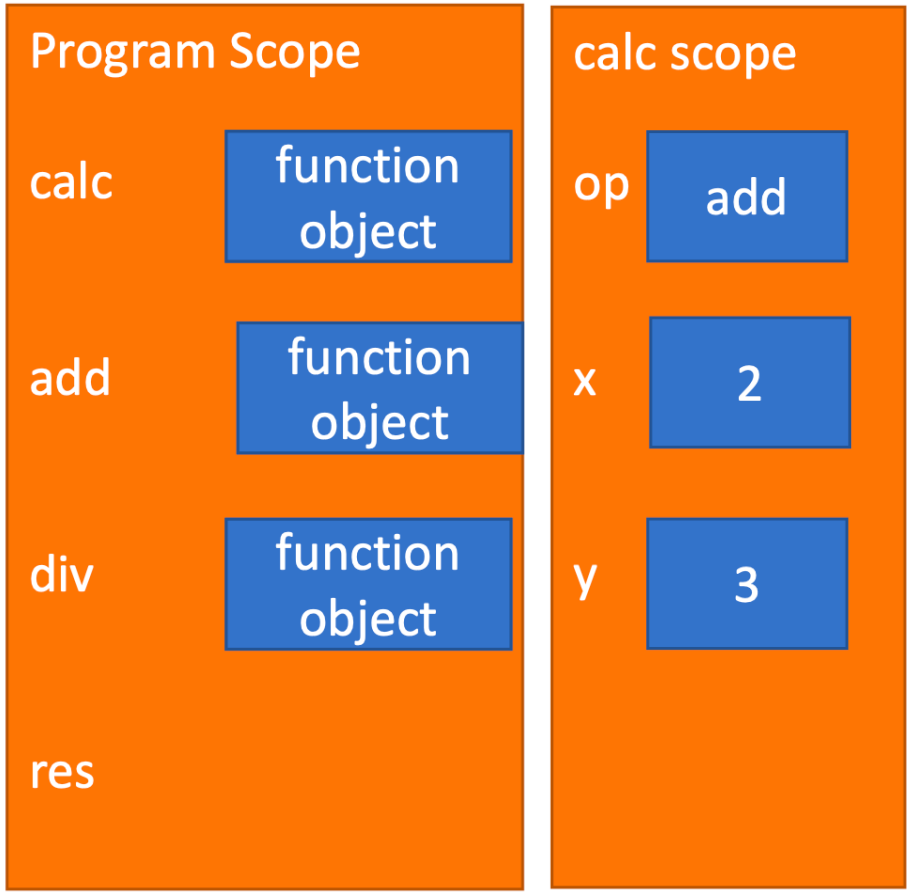
函数对象



■函数作为参数

```
def calc(op, x, y):
    return op(x,y)

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

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

res = calc(add, 2, 3)
```

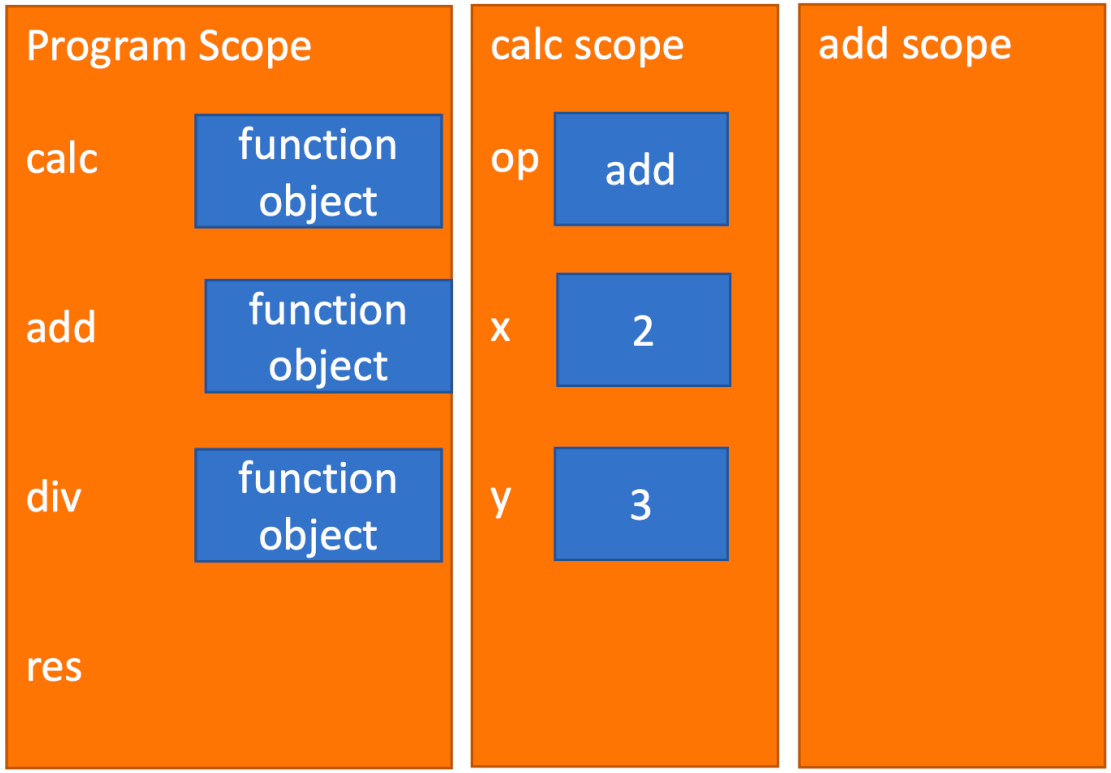
等于 `return add(2, 3)`
相当于将每一个变量
替换为实际的对象



■函数作为参数

```
def calc(op, x, y):
    return op(x,y)

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

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

res = calc(add, 2, 3)
```

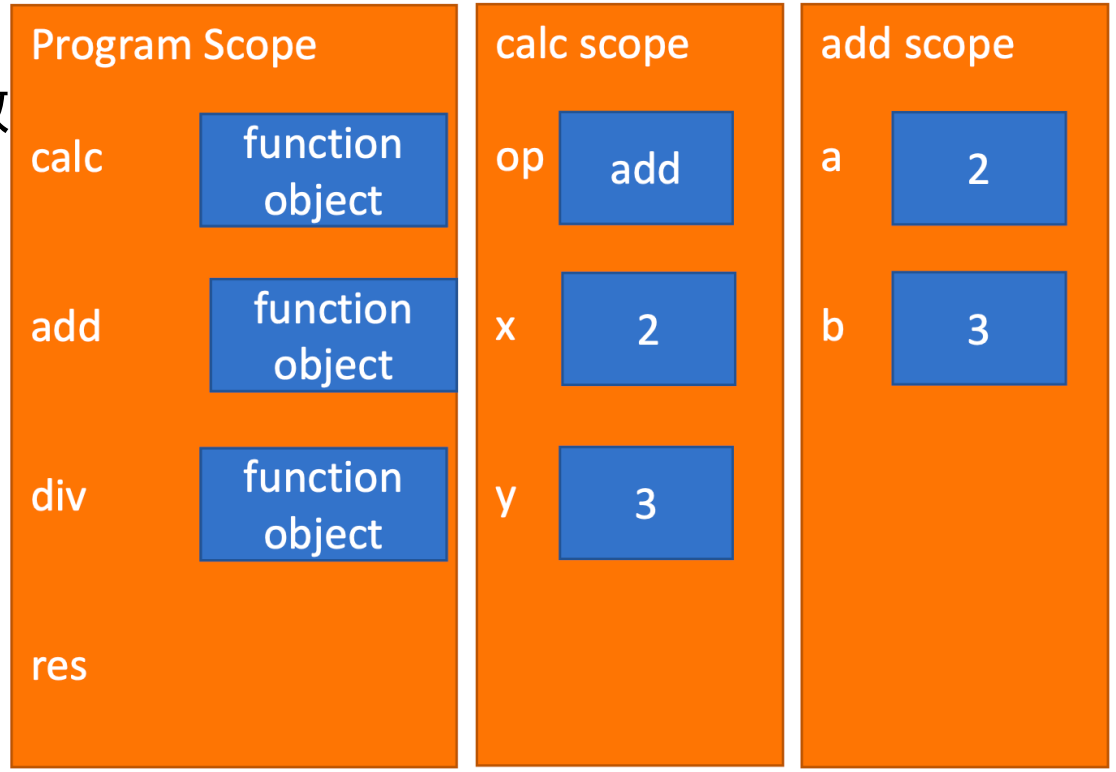
在calc作用域执行函数调用



■函数作为参数

```
def calc(op, x, y):
    return op(x,y)

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

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

res = calc(add, 2, 3)
```

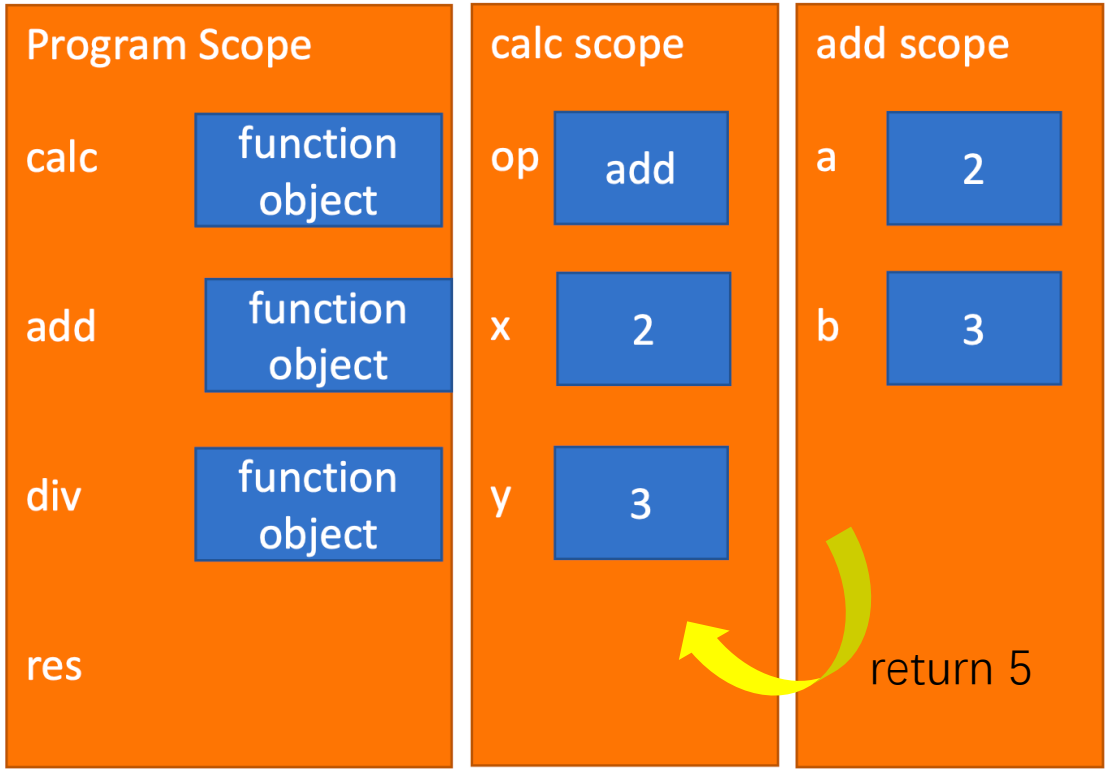
调用add函数，传入2和3参数



■函数作为参数

```
def calc(op, x, y):
    return op(x,y)

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

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

res = calc(add, 2, 3)
```

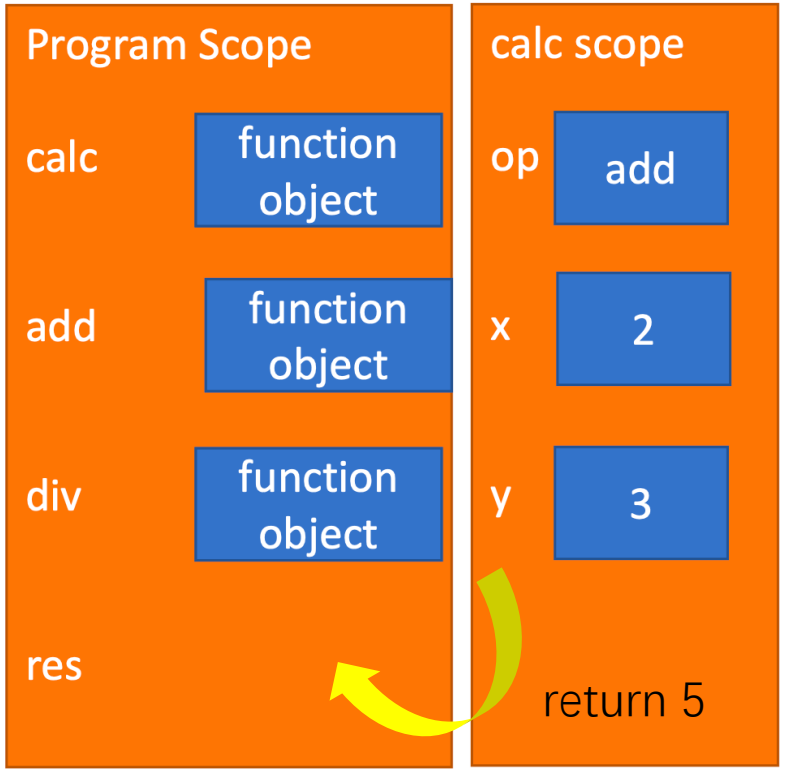
返回5



■函数作为参数

```
def calc(op, x, y):
    return op(x,y)

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

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

res = calc(add, 2, 3)
```

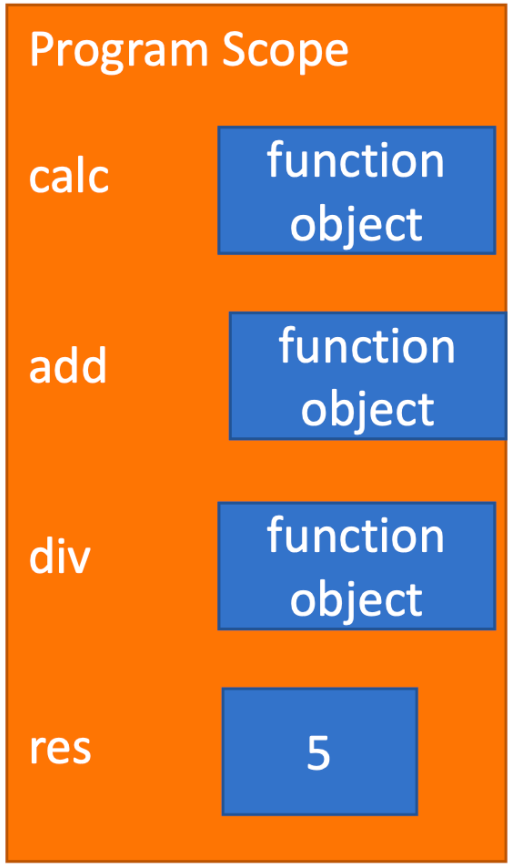
再返回5



■函数作为参数

```
def calc(op, x, y):
    return op(x,y)

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

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

res = calc(add, 2, 3)
```

得到5



- 自己尝试下，以下代码res的值是什么？打印出什么？

```
def calc(op, x, y):
    return op(x,y)

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(div,2,0)
```

■函数作为参数的另一个例子

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b‘)
    return y
def func_c(f, z):
    print('inside func_c‘)
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```

调用func_a，没有输入参数

调用func_b，输入一个整数参数

调用func_c，输入两个参数，一个函数一个整数

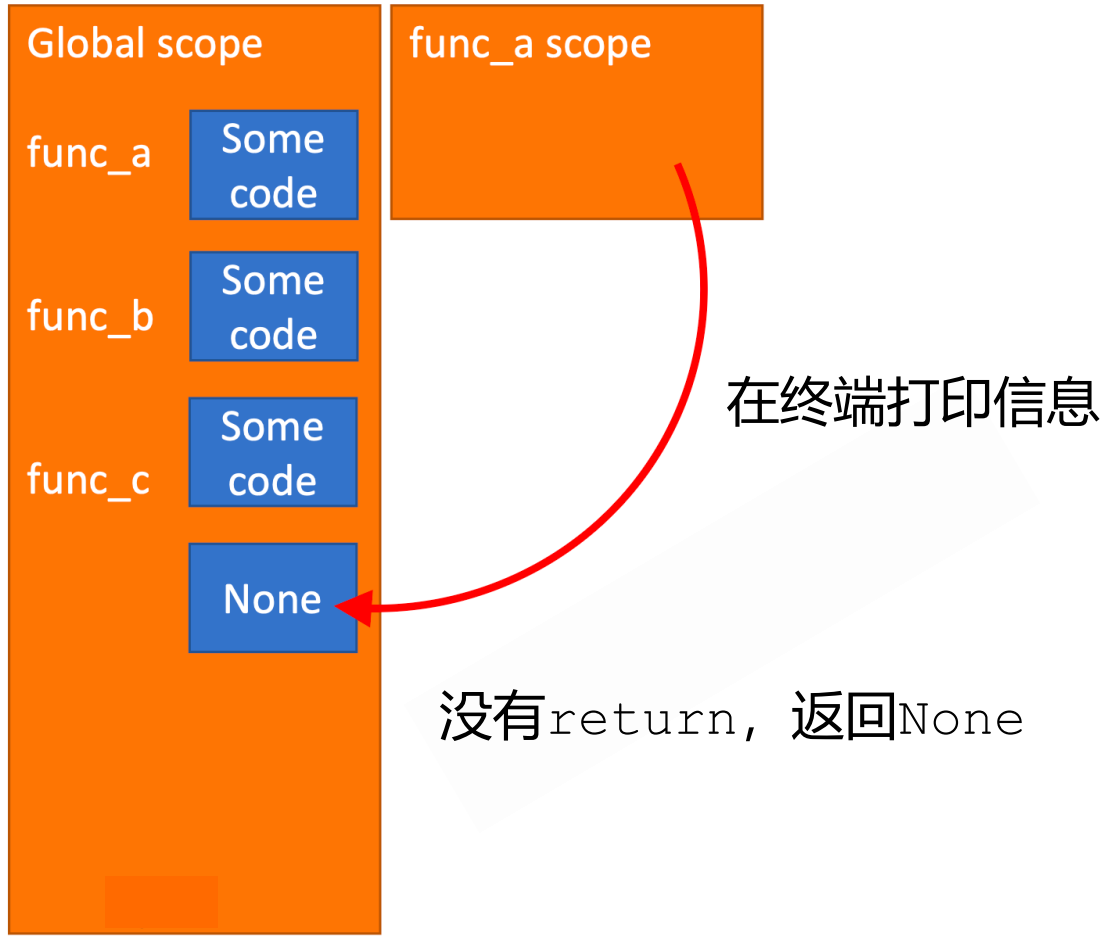
■函数作为参数的另一个例子

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b‘)
    return y
def func_c(f, z):
    print('inside func_c‘)
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```

由于没有参数，
没发生变量绑定

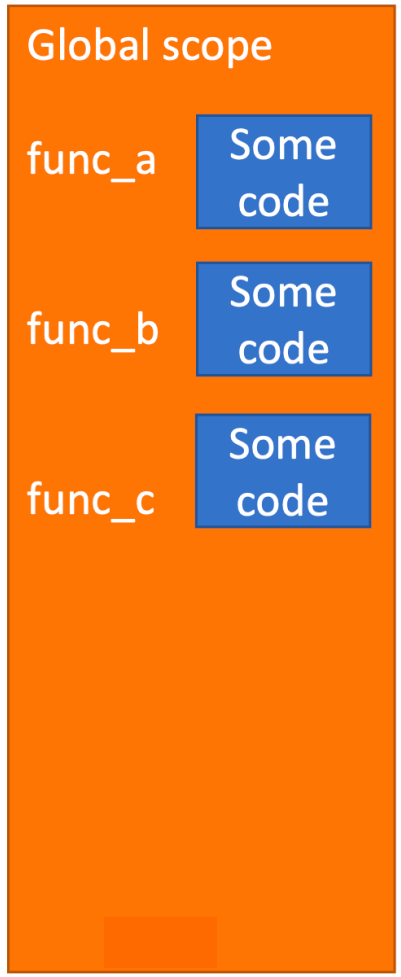
■函数作为参数的另一个例子

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



■函数作为参数的另一个例子

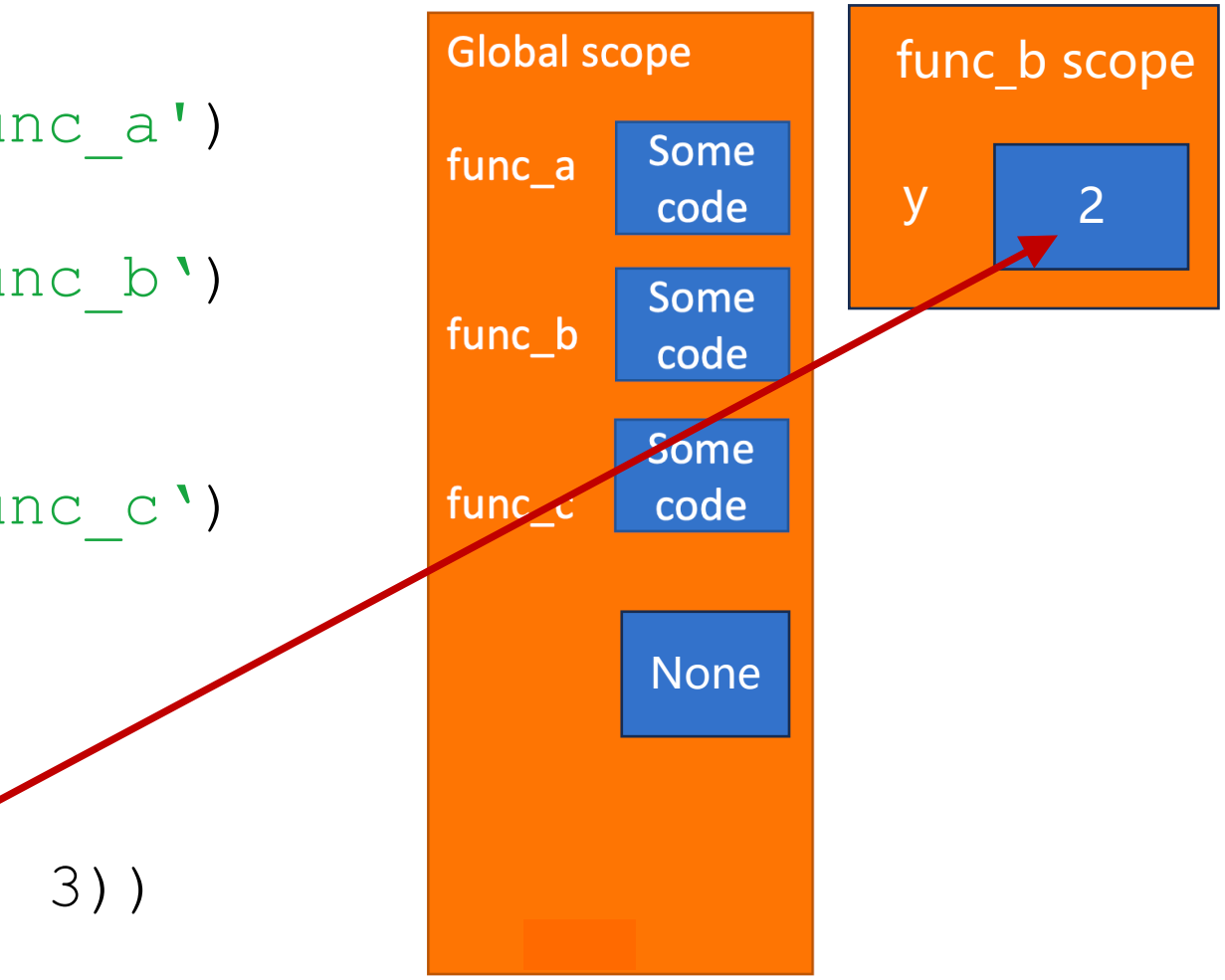
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b‘)
    return y
def func_c(f, z):
    print('inside func_c‘)
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



print在终端输出None

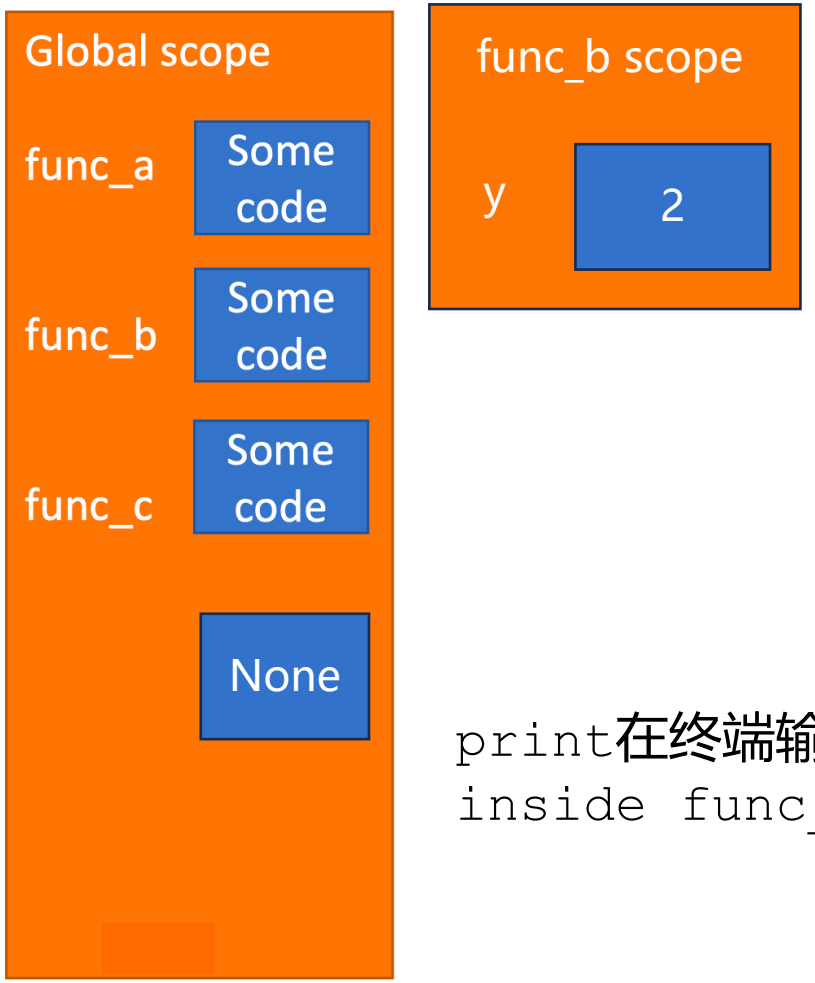
■函数作为参数的另一个例子

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



■函数作为参数的另一个例子

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```

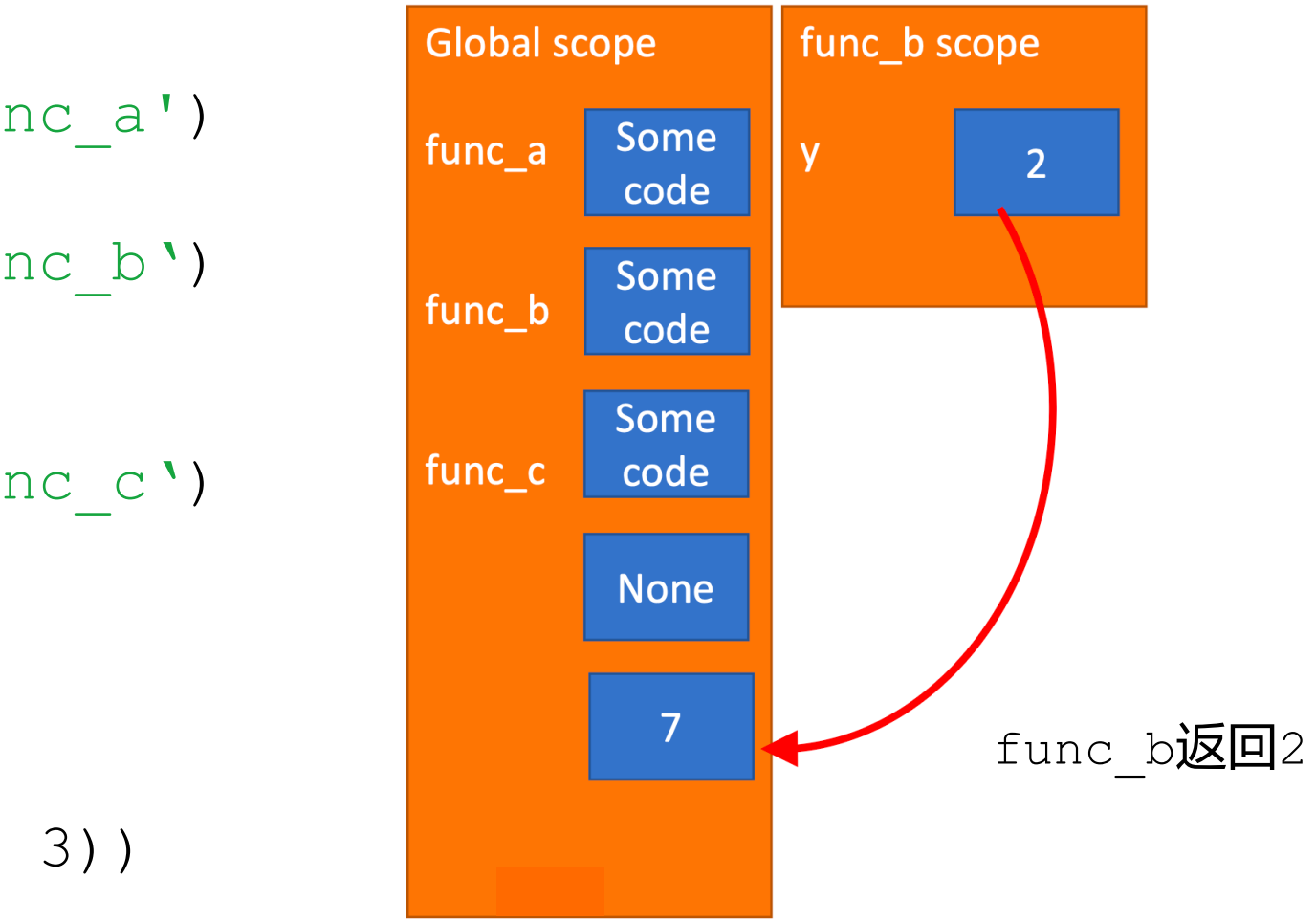


print在终端输出

```
inside func_b
```

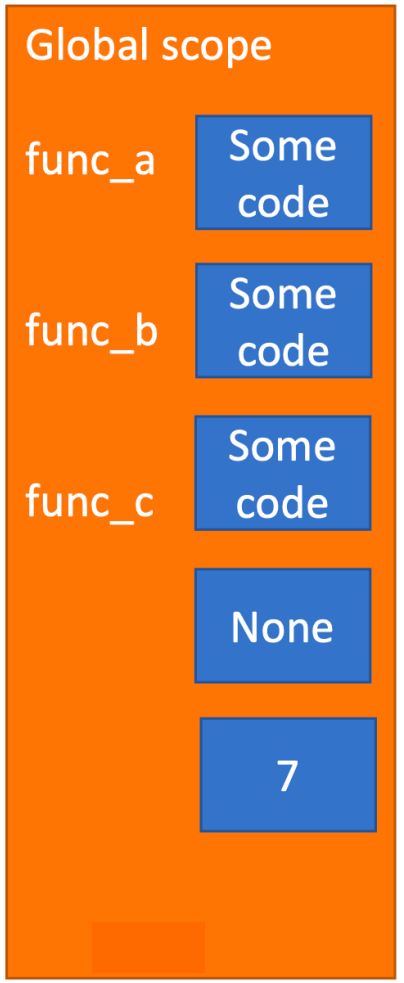
■函数作为参数的另一个例子

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



■函数作为参数的另一个例子

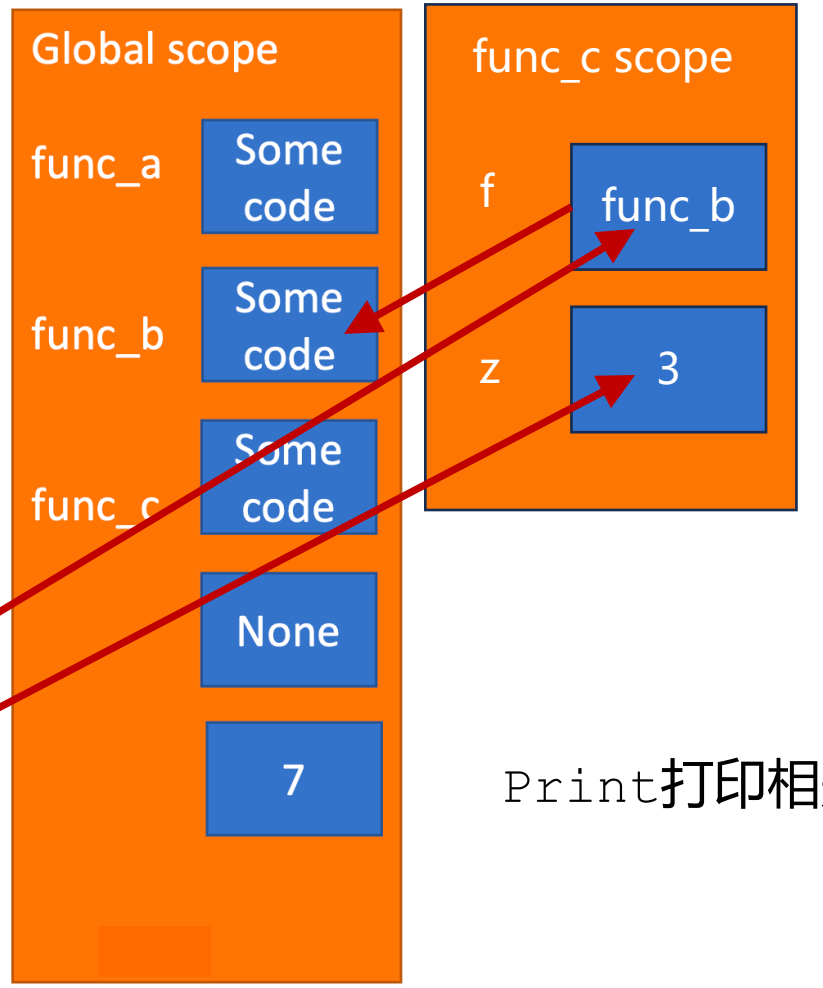
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b‘)
    return y
def func_c(f, z):
    print('inside func_c‘)
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



print在终端显示7

■函数作为参数的另一个例子

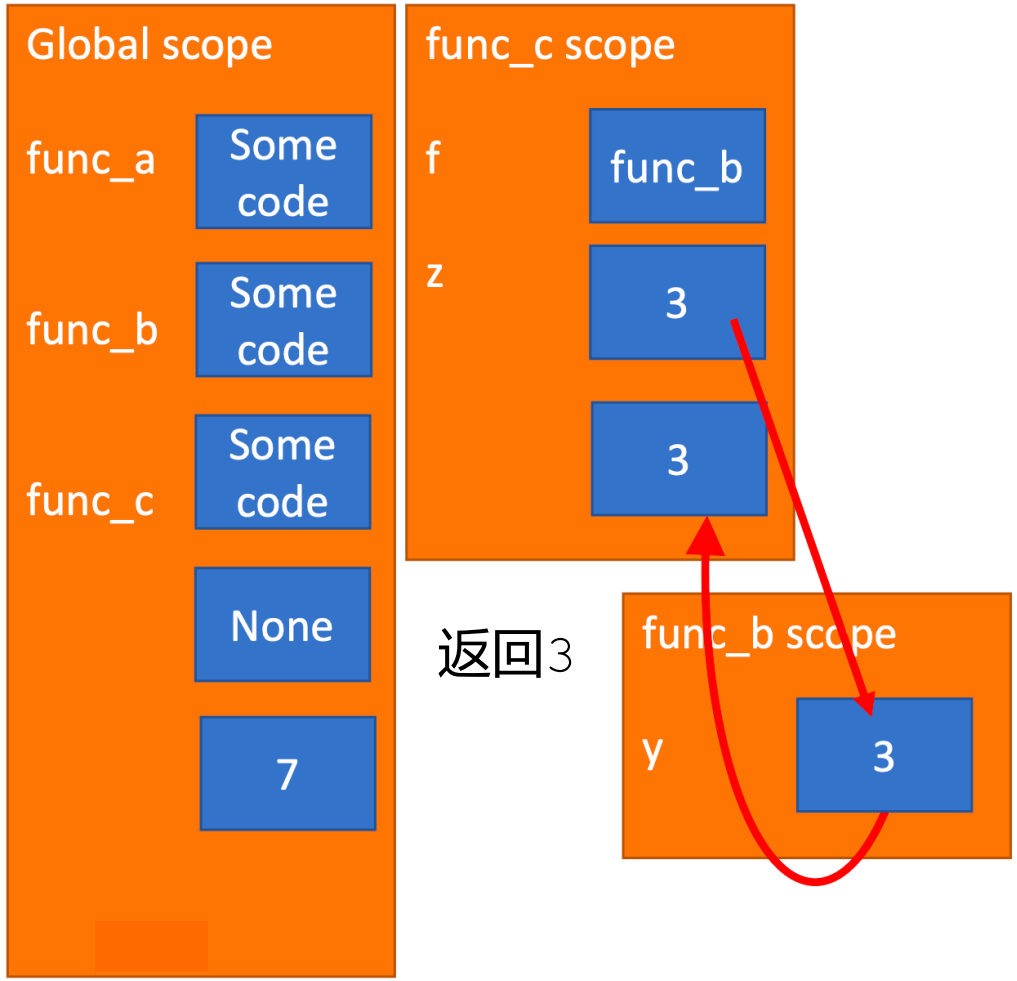
```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



Print打印相关信息

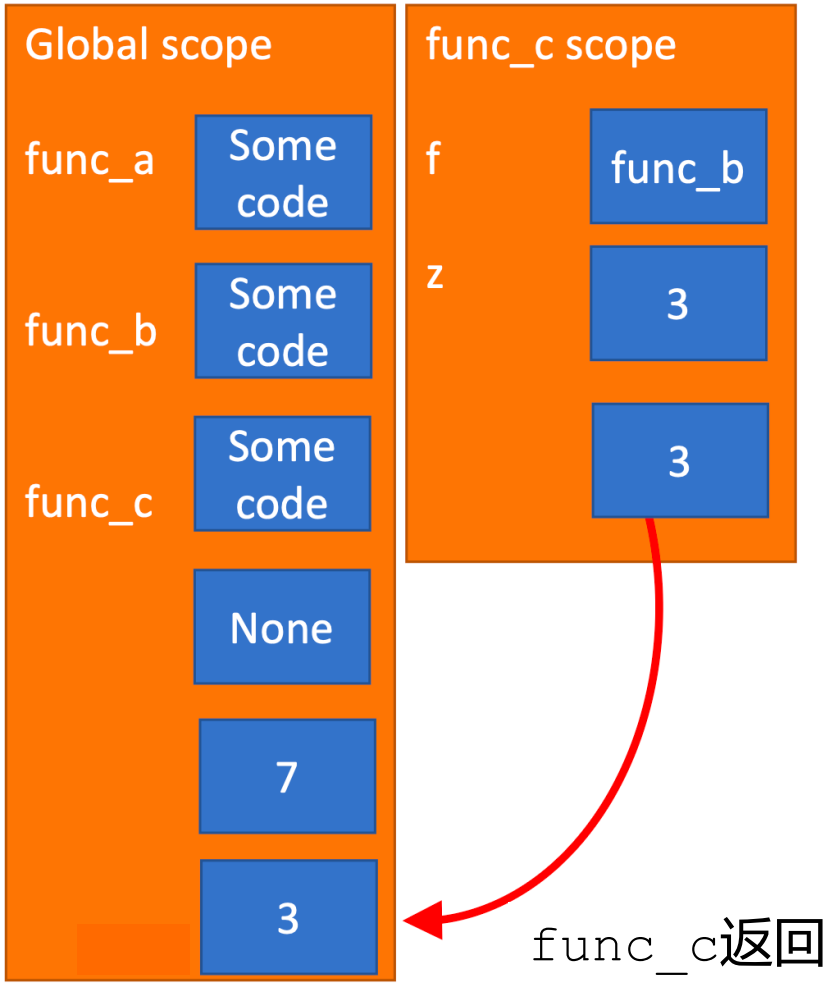
■函数作为参数的另一个例子

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



■函数作为参数的另一个例子

```
def func_a():
    print('inside func_a')
def func_b(y):
    print('inside func_b')
    return y
def func_c(f, z):
    print('inside func_c')
    return f(z)

print(func_a())
print(5 + func_b(2))
print(func_c(func_b, 3))
```



■函数的正式形式与涉及角色

```
def name_of_function(parameters):
    """ some specifications """
    statements
    return value # optionally
```

函数涉及的三种角色

用户

函数调用者

函数作者

■函数角色场景

作者（coder）

```
def meters_to_cm(meters):
    return 100 * meters


def main():
    result = meters_to_cm(5.2)
    print(result)
```

调用者（coder）

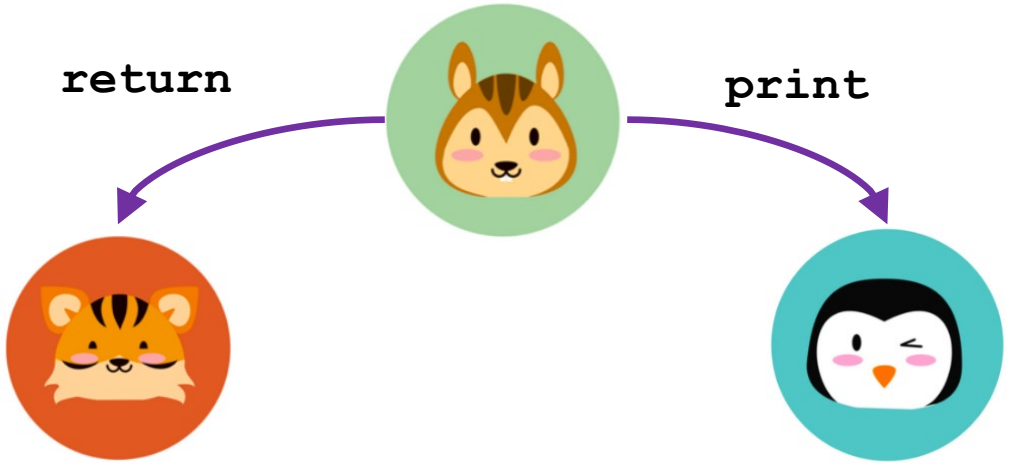
用户

terminal

```
> python m2cm.py
520.0
```

■函数角色场景——输出方式的不同

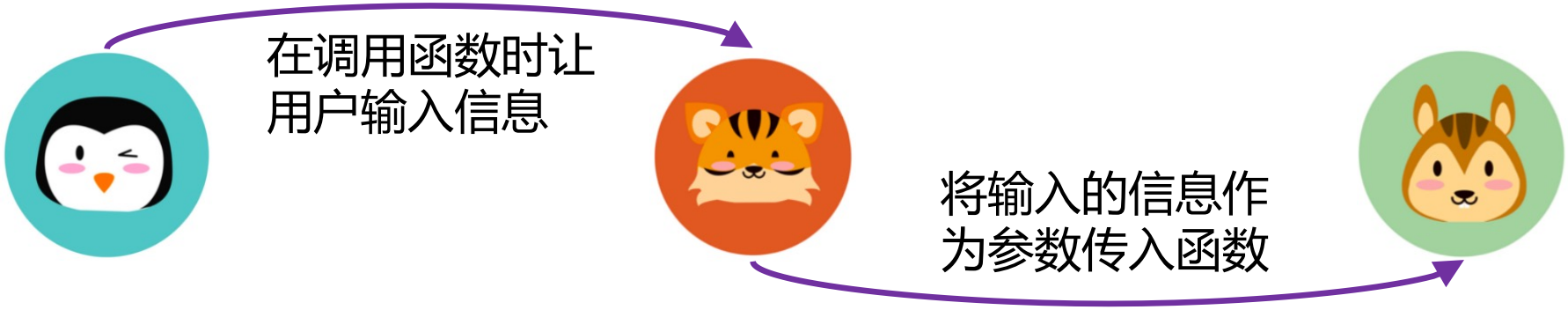
```
def meters_to_cm_case1(meters):
    return 100 * meters
def meters_to_cm_case2(meters):
    print(100 * meters)
```



调用者接收返回值，
可以任意处理

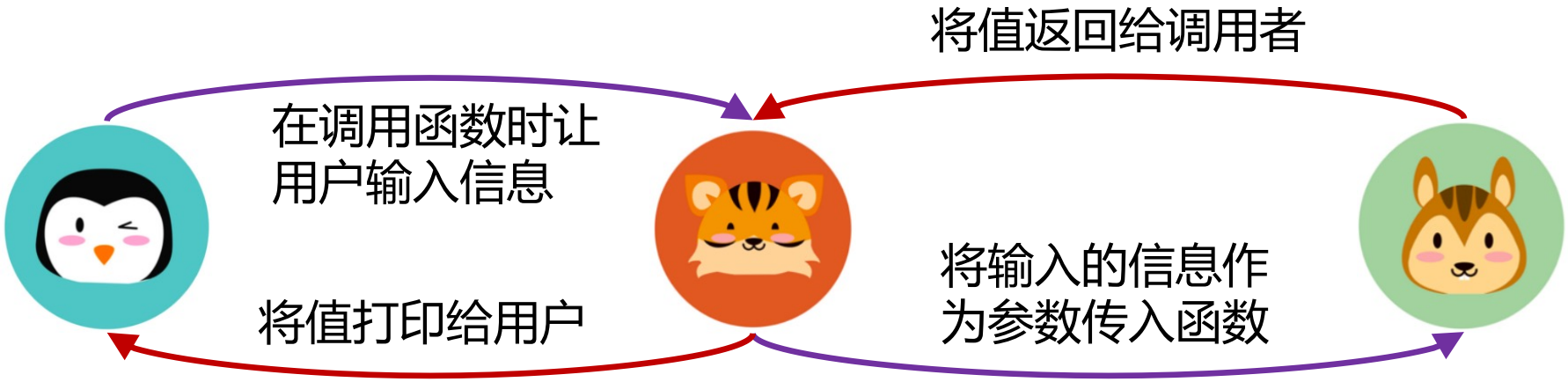
用户在终端看到打印的值

■函数角色场景——理想的信息流




```
def example_caller():
    data = float(input("enter: "))
    call_the_function(data)
```

■函数角色场景——理想的信息流




```
def example_caller():
    data = float(input("enter: "))
    result = call_the_function(data)
    print(result)
```

■函数中进行测试——doctest

```
def factorial(n): """
    This function returns the factorial of n Input: n (number to compute the factorial of)
    Returns: value of n factorial
    Doctests:
    >>> factorial(3)
    6
    >>> factorial(1)
    1
    >>> factorial(0)
    1
    """
    result = 1
    for i in range(1, n + 1):
    result *= i
    return result
```

doctest是python的内置功能，允许开发者将测试用例嵌入到docstring中，格式符合从终端中复制过来的形式，>>>后面是调用测试例子，下面一行是预期的输出，执行测试可以在终端中以命令启动：

```
python –m doctest fact.py –v
```

（假设代码存于fact.py文件中）

■函数中定义函数

```
def main():
    print("hello world")
    def say_goodbye():
        print("goodbye!")
    say_goodbye()
```

- 不建议在一个函数中定义另一个函数：
  - 每次调用外部函数，内部函数都被重新定义一次，增加额外开销
  - 使代码层级变深，且依赖外部函数变量，增加阅读与维护难度
  - 内部函数无法直接在外部测试，需要调用外部函数间接验证，增加复杂度

■函数的总结

- 函数是一种数据对象
  - 函数拥有类型
  - 函数可以作为数据被绑定给一个名字
  - 函数可以作为其他函数的参数
  - 函数可以作为值被返回给其他函数
- 要注意作用域
  - 主程序在全局作用域中运行
  - 函数调用时在一个新的临时作用域中运行
- 恰当地定义函数可以提升代码的可读性，建立多角色代码访问场景

Reading and QA Time

See you next week !