潍坊科技学院
Weifang University of Science and Technology

第6章 函数

授课老师：刘国旭

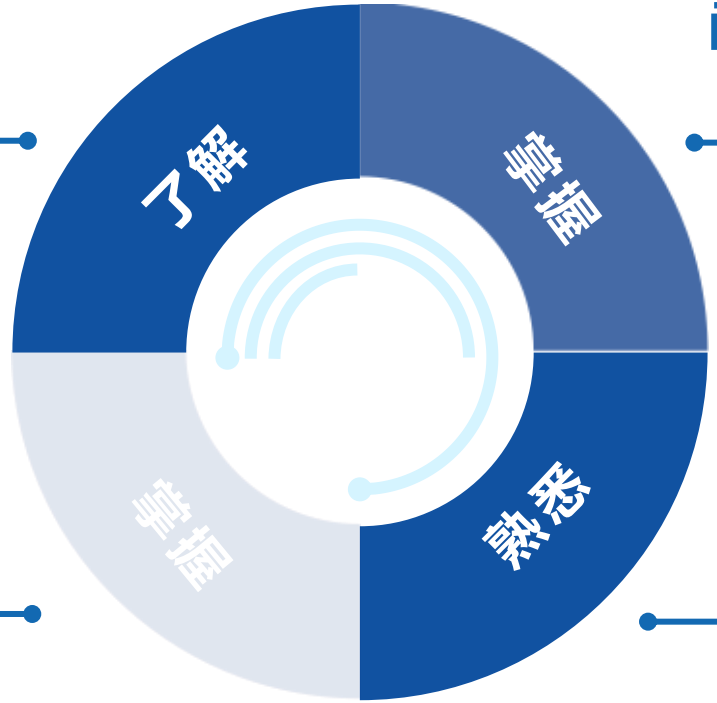
潍坊科技学院

- 函数的定义和调用
- 函数参数的传递
- 函数的返回值
- 变量作用域
- 特殊形式的函数

学习目标

1 了解 函数的概念及优势

2 掌握 函数的定义和使用、函数参数的传递



3 熟悉 变量作用域

4 掌握 局部变量和全局变量、递归函数和匿名函数

目录页

目录页

目录页

6.1 函数概述

函数是组织好的、实现单一功能或相关联功能的代码段。我们可以将函数视为一段有名字的代码，这类代码可以在需要的地方以“函数名()”的形式调用。

- ✓ print()
- ✓ input()

6.1 函数概述

```
# 打印边长为2个星号的正方形
for i in range(2):
    for i in range(2):
        print("*", end=" ")
    print()

# 打印边长为3个星号的正方形
for i in range(3):
    for i in range(3):
        print("*", end=" ")
    print()

# 打印边长为4个星号的正方形
for i in range(4):
    for i in range(4):
        print("*", end=" ")
    print()
```

(a)未使用函数的程序

```
# 打印正方形的函数
def print_triangle(lenth):
    for i in range(lenth):
        for i in range(lenth):
            print("*", end=" ")
        print()
```

```
# 使用函数，打印边长为2个星号的正方形
print_triangle(2)
# 使用函数，打印边长为3个星号的正方形
print_triangle(3)
# 使用函数，打印边长为4个星号的正方形
print_triangle(4)
```

(b)使用函数的程序

- 结构清晰
- 代码精简

6.1 函数概述

程序若希望再打印一个边长为5个星号的正方形，应该如何解决呢?

6.1 函数概述

```
# 打印边长为2个星号的正方形
for i in range(2):
    for i in range(2):
        print("*", end=" ")
    print()

# 打印边长为3个星号的正方形
for i in range(3):
    for i in range(3):
        print("*", end=" ")
    print()

# 打印边长为4个星号的正方形
for i in range(4):
    for i in range(4):
        print("*", end=" ")
    print()

# 打印边长为5个星号的正方形
for i in range(5):
    for i in range(5):
        print("*", end=" ")
    print()
```

冗余代码继续增加

(a)未使用函数的程序

```
# 打印正方形的函数
def print_triangle(lenth):
    for i in range(lenth):
        for i in range(lenth):
            print("*", end=" ")
        print()

# 使用函数，打印边长为2个星号的正方形
print_triangle(2)
# 使用函数，打印边长为3个星号的正方形
print_triangle(3)
# 使用函数，打印边长为4个星号的正方形
print_triangle(4)
# 使用函数，打印边长为5个星号的正方形
print_triangle(5)
```

再次调用函数

(b)使用函数的程序

6.1 函数概述

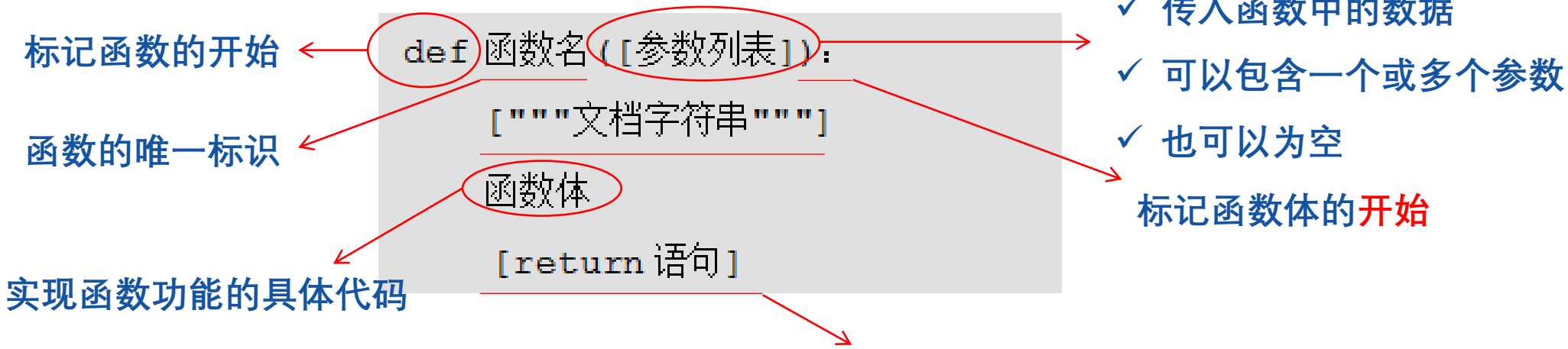
函数式编程具有以下优点：

- 将程序模块化，既减少了冗余代码，又让程序结构更为清晰
- 提高开发人员的编程效率
- 方便后期的维护与扩展

目录页

6.2.1 定义函数

前面使用的print()函数和input()都是Python的内置函数，这些函数由Python定义。开发人员也可以根据自己的需求定义函数，Python中使用关键字def来定义函数，其语法格式如下：



6.2.1 定义函数

例如，定义一个计算两个数之和的函数。

```
def add():
    result = 11 + 22
    print(result)
```

无参函数

```
def add_modify(a, b):
    result = a + b
    print(result)
```

有参函数

6.2.2 调用函数

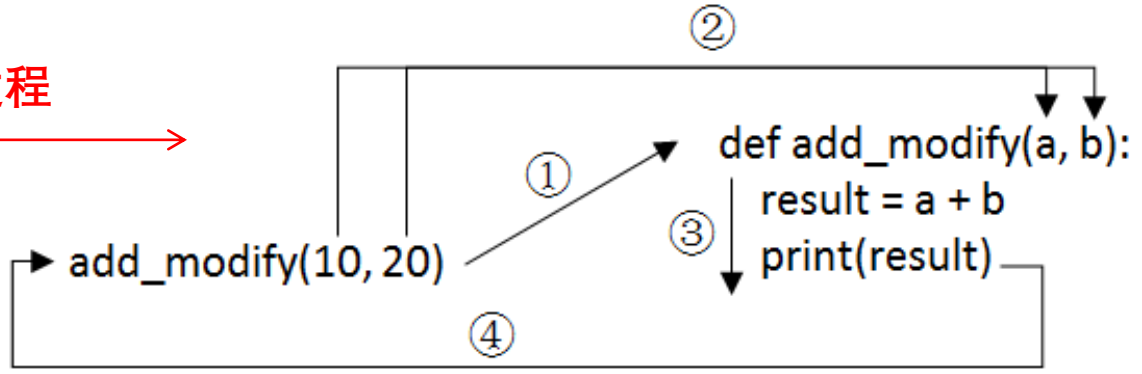
- 函数在定义完成后不会立刻执行，直到被程序调用时才会执行。
- 语法格式如下：

函数名([参数列表])

例如

```
add()
add_modify(10, 20)
```

调用过程



1. 程序在调用函数的位置暂停执行。
2. 将数据传递给函数参数。
3. 执行函数体中的语句。
4. 程序回到暂停处继续执行。

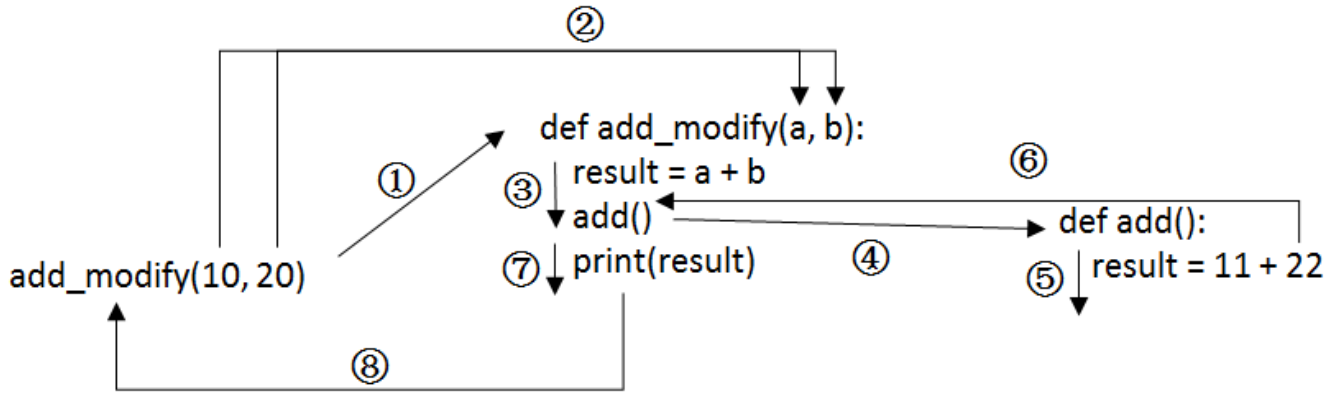
6.2.2 调用函数

函数内部也可以调用其他函数，这被称为函数的嵌套调用。

例如

```
def add_modify(a, b):
    result = a + b
    add()
    print(result)
add_modify(10, 20)
```

嵌套调用函数add()



多学一招：函数的嵌套定义

函数在定义时可以在其内部嵌套定义另外一个函数，此时嵌套的函数称为外层函数，被嵌套的函数称为内层函数。

示例

```
def add_modify(a, b):
    result = a + b
    print(result)
    def test():
        print("我是内层函数")
add_modify(10, 20)
```

内层函数

注意

- 函数外部无法直接调用内层函数
- 只能通过外层函数间接调用内层函数

多学一招：函数的嵌套定义

示例

```
def add_modify(a, b):
    result = a + b
    print(result)
    def test():
        print("我是内层函数")
    test()
add_modify(10, 20)
```

目录页

6.3 函数参数的传递

我们通常将定义函数时设置的参数称为形式参数（简称为形参），将调用函数时传入的参数称为实际参数（简称为实参）。函数的参数传递是指将实际参数传递给形式参数的过程。

6.3 函数参数的传递

函数参数的传递可以分为位置参数传递、关键字参数传递、默认参数传递、参数的打包与解包以及混合传递。

6.3.1 位置参数的传递

函数在被调用时会将实参按照相应的位置依次传递给形参，也就是说将第一个实参传递给第一个形参，将第二个实参传递给第二个形参，以此类推。

```
def get_max(a, b):
    if a > b:
        print(a,"是较大的值！")
    else:
        print(b,"是较大的值！")
get_max(8, 5)
```

8是较大的值

6.3.2 关键字参数的传递

关键字参数的传递是通过“形参=实参”的格式将实参与形参相关联，将实参按照相应的关键字传递给形参。

```
def connect(ip, port):
    print(f"设备{ip}:{port}连接！")
connect(ip="127.0.0.1", port=8080)
```

6.3.2 关键字参数的传递

无论实参采用位置参数的方式传递，还是关键字参数的方式传递，每个形参都是有名称的，怎么区分用哪种方式传递呢？

符号“/”

6.3.2 关键字参数的传递

Python在3.8版本中新增了仅限位置形参的语法，使用符号“/”来限定部分形参只接收采用位置传递方式的实参。

例如：

```
def func(a, b, /, c):
    print(a, b, c)
```

```
# 错误的调用方式
# func(a=10, 20, 30)
# func(10, b=20, 30)
# 正确的调用方式
func(10, 20, c=30)
```

6.3.3 默认参数的传递

函数在定义时可以指定形参的默认值，如此在被调用时可以选择是否给带有默认值的形参传值，若没有给带有默认值的形参传值，则直接使用该形参的默认值。

定义

```
def connect(ip, port=8080):
    print(f"设备{ip}:{port}连接！")
```

调用

```
connect(ip="127.0.0.1")
connect(ip="127.0.0.1", port=3306)
```

结果

```
设备127.0.0.1:8080连接!
设备127.0.0.1:3306连接!
```

6.3.4 参数的打包与解包

1.打包

如果函数在定义时无法确定需要接收多少个数据，那么可以在定义函数时为形参添加“*”或“**”：

- “*” —— 接收以元组形式打包的多个值
- “**” —— 接收以字典形式打包的多个值

6.3.4 参数的打包与解包

1.打包——“*”

```
def test(*args):
    print(args)
```

定义

```
test(11, 22, 33, 44, 55)
```

调用

```
(11, 22, 33, 44, 55)
```

结果

6.3.4 参数的打包与解包

1.打包——“**”

```
def test(**kwargs):
    print(kwargs)
```

定义

```
test(a=11, b=22, c=33, d=44, e=55)
```

调用

```
{'a': 11, 'b': 22, 'c': 33, 'd': 44, 'e': 55}
```

结果

6.3.4 参数的打包与解包

1.打包——“**”

- 虽然函数中添加“*”或“**”的形参可以是符合命名规范的任意名称，但这里建议使用*args和**kwargs。
- 若函数没有接收到任何数据，参数*args和**kwargs为空，即它们为空元组或空字典。

2.解包

- ✓ 实参是元组 → 可以使用“*”拆分成多个值 → 按位置参数传给形参
- ✓ 实参是字典 → 可以使用“**” 拆分成多个键值对 → 按关键字参数传给形参

2.解包

定义

```
def test(a, b, c, d, e):
    print(a, b, c, d, e)
```

调用

```
nums = (11, 22, 33, 44, 55)
test(*nums)
```

结果

```
11 22 33 44 55
```

2.解包

定义

```
def test(a, b, c, d, e):
    print(a, b, c, d, e)
```

调用

```
nums = {"a":11, "b":22, "c":33, "d":44, "e":55}
test(**nums)
```

结果

```
11 22 33 44 55
```

6.3.5 混合传递

前面介绍的参数传递的方式在定义函数或调用函数时可以混合使用，但是需要遵循一定的规则，具体规则如下。

- ✓ 优先按位置参数传递的方式。
- ✓ 然后按关键字参数传递的方式。
- ✓ 之后按默认参数传递的方式。
- ✓ 最后按打包传递的方式。

6.3.5 混合传递

在定义函数时：

- ✓ **带有默认值的参数必须位于普通参数之后。**
- ✓ 带有“*”标识的参数必须位于带有默认值的参数之后。
- ✓ 带有“**”标识的参数必须位于带有“*”标识的参数之后。

6.3.5 混合传递

定义

```
def test(a, b, c=33, *args, **kwargs):
    print(a, b, c, args, kwargs)
```

调用

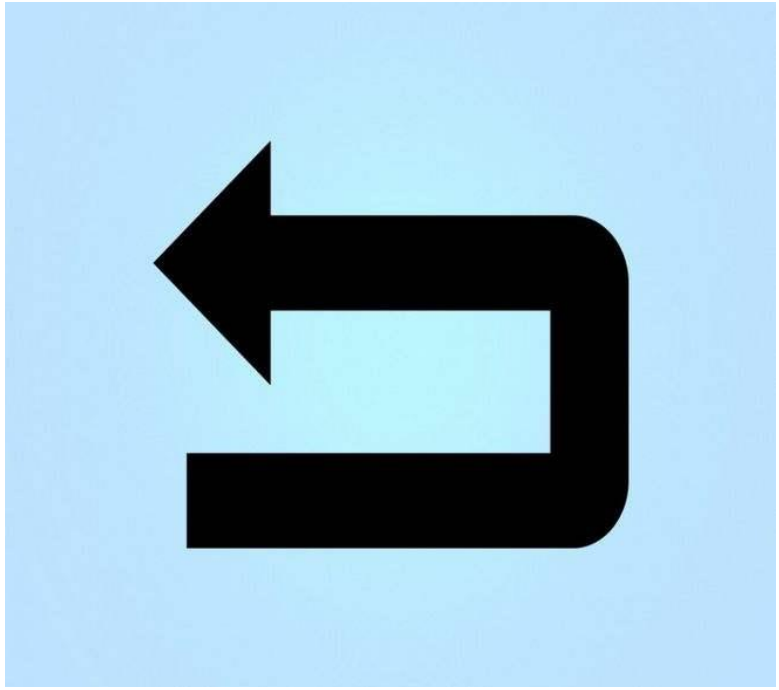
```
test(1, 2)
test(1, 2, 3)
test(1, 2, 3, 4)
test(1, 2, 3, 4, e=5)
```

结果

```
1 2 33 () {}
1 2 3 () {}
1 2 3 (4,) {}
1 2 3 (4,) {'e': 5}
```

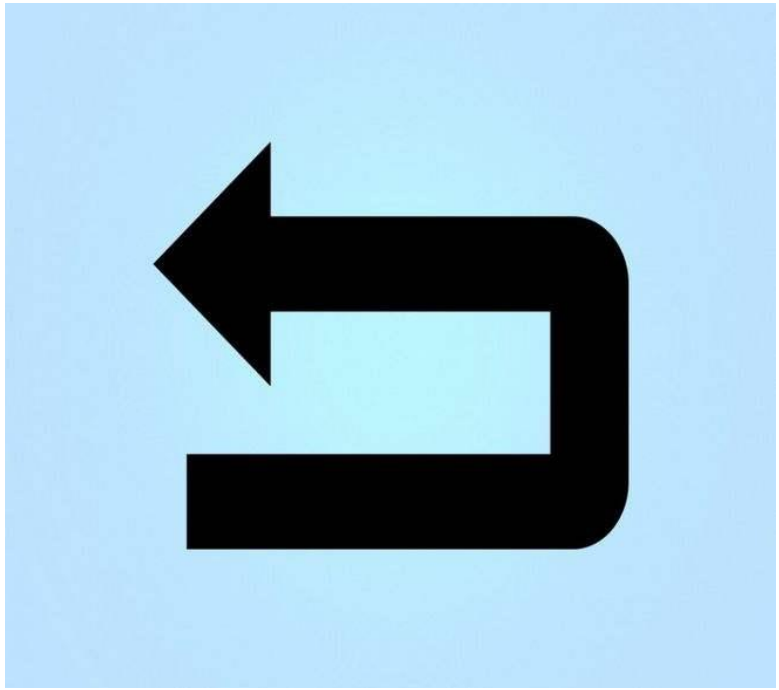
目录页

6.4 函数的返回值



函数中的return语句会在函数结束时将数据返回给程序，同时让程序回到函数被调用的位置继续执行。

6.4 函数的返回值



定义

```
def filter_sensitive_words(words):
    if "山寨" in words:
        new_words = words.replace("山寨", "**")
        return new_words
```

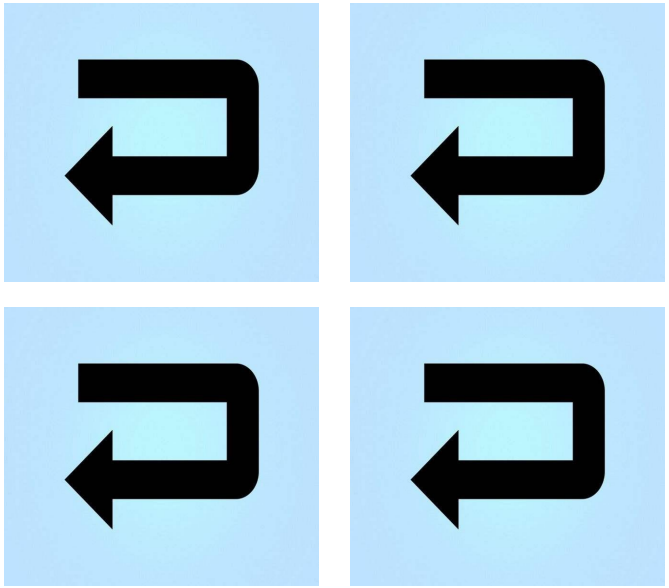
调用

```
result = filter_sensitive_words("这个手机是山寨版吧！")
print(result)
```

结果

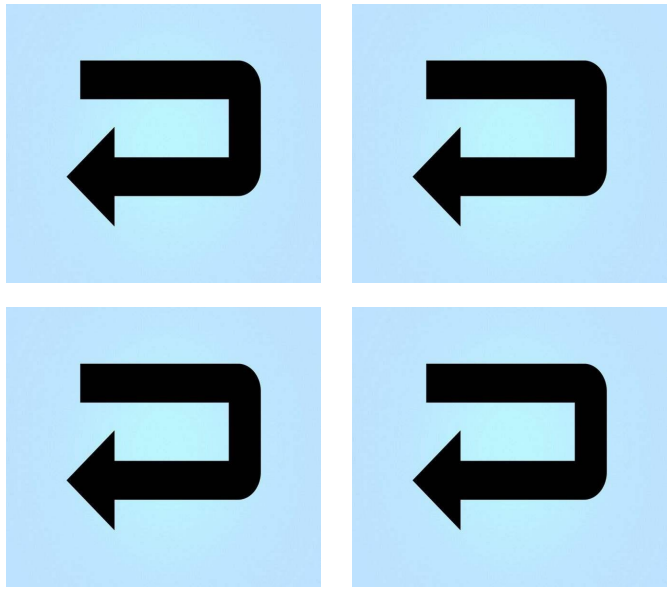
```
这个手机是**版吧!
```

6.4 函数的返回值



如果函数使用return语句返回了多个值，那么这些值将被保存到元组中。

6.4 函数的返回值



定义

```
def move(x, y, step):
    nx = x + step
    ny = y - step
    return nx, ny      # 使用return语句返回多个值
```

调用

```
result = move(100, 100, 60)
print(result)
```

结果

```
(160, 40)
```

目录页

6.5 变量作用域

变量并非在程序的任意位置都可以被访问，其访问权限取决于变量定义的位置，其所处的有效范围称为变量的作用域。

6.5.1 局部变量和全局变量

根据作用域的不同，变量可以划分为局部变量和全局变量。

6.5.1 局部变量和全局变量

1.局部变量

- 函数内部定义的变量，只能在函数内部被使用
- 函数执行结束之后局部变量会被释放，此时无法再进行访问。

6.5.1 局部变量和全局变量

1.局部变量

示例

```
def test_one():
    number = 10                    # 局部变量
    print(number)                  # 函数内部访问局部变量
test_one()
print(number)                      # 函数外部访问局部变量
```

```
10

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-7a06f0043e50> in <module>
      3     print(number)                                  # 函数内部访问局部变量
      4 test_one()
----> 5 print(number)                                      # 函数外部访问局部变量

NameError: name 'number' is not defined
```

6.5.1 局部变量和全局变量

1.局部变量

不同函数内部可以包含同名的局部变量，这些局部变量的关系类似于不同目录下同名文件的关系，它们相互独立，互不影响。

示例

```
def test_one():
    number = 10
    print(number)          # 访问test_one()函数的局部变量number
def test_two():
    number = 20
    print(number)          # 访问test_two()函数的局部变量number
test_one()
test_two()
```

结果

```
10
20
```

2.全局变量

全局变量可以在整个程序的范围内起作用，它不会受到函数范围的影响。

2.全局变量

示例

```
number = 10                          # 全局变量
def test_one():
    print(number)                    # 函数内部访问全局变量
test_one()
print(number)                        # 函数外部访问全局变量
```

结果

```
10
10
```

2.全局变量

全局变量在函数内部只能被访问，而无法直接修改。

示例

```
# 定义全局变量
number = 10
def test_one():
    print(number)
    number += 1
test_one()
print(number)
```

已经声明

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-2-fe5e721058c9> in <module>
      4     print(number)
      5     number += 1
----> 6 test_one()
      7 print(number)

<ipython-input-2-fe5e721058c9> in test_one()
      2 number = 10
      3 def test_one():
----> 4     print(number)
      5     number += 1
      6 test_one()

UnboundLocalError: local variable 'number' referenced before assignment
```

没有找到

2.全局变量

这是因为函数内部的变量number视为局部变量，而在执行“number+=1”这行代码之前并未声明过局部变量number。

函数内部只能访问全局变量，而无法直接修改全局变量。

多学一招：LEGB原则

LEGB是程序中搜索变量时所遵循的原则，该原则中的每个字母指代一种作用域，具体如下：

L-local

局部作用域

例如，局部变量和形参生效的区域。

E-enclosing

嵌套作用域

例如，嵌套定义的函数中外层函数声明的变量生效的区域。

G-global

全局作用域

例如，全局变量生效的区域。

B-built-in

内置作用域

例如，内置模块声明的变量生效的区域。

Python在搜索变量时会按照“L-E-G-B”这个顺序依次在这四种区域中搜索变量：若搜索到变量则终止搜索，使用搜索到的变量；若搜索完L、E、G、B这四种区域仍无法找到变量，程序将抛出异常。

6.5.2 global和nonlocal关键字

函数内部无法直接修改全局变量或在嵌套函数的外层函数声明的变量，但可以使用global或nonlocal关键字修饰变量以间接修改以上变量。

1.global关键字

使用global关键字可以将局部变量声明为全局变量，其使用方法如下：

global 变量

示例

```
number = 10                  # 定义全局变量
def test_one():
    global number            # 使用global声明变量number为全局变量
    number += 1
    print(number)
test_one()
print(number)
```

结果

```
11
11
```

2.nonlocal关键字

使用nonlocal关键字可以在局部作用域中修改嵌套作用域中定义的变量，其使用方法如下：

nonlocal 变量

示例

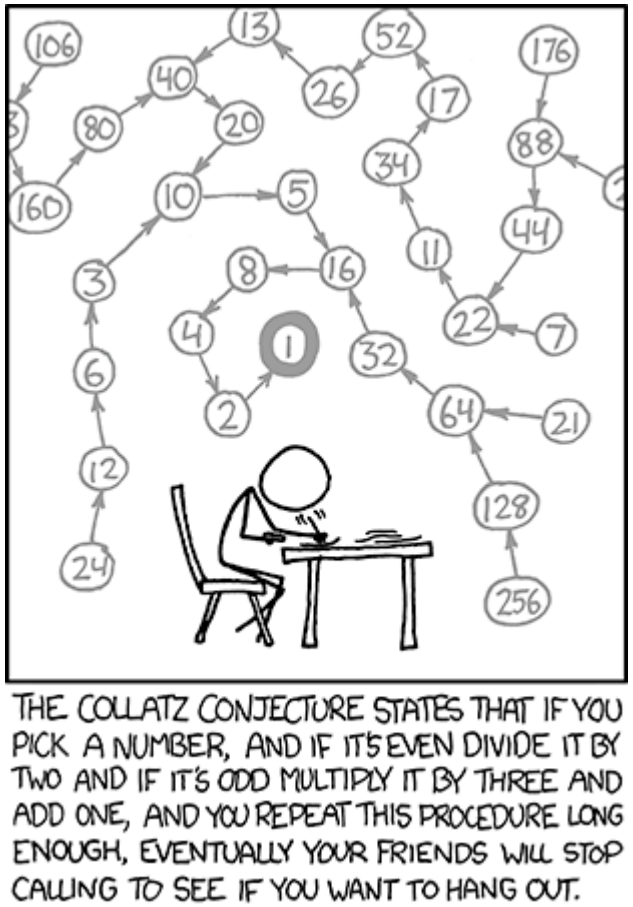
```
def test():
    number = 10
    def test_in():
        nonlocal number
        number = 20
    test_in()
    print(number)
test()
```

结果

```
20
```

目录页

6.6.1 角谷猜想



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

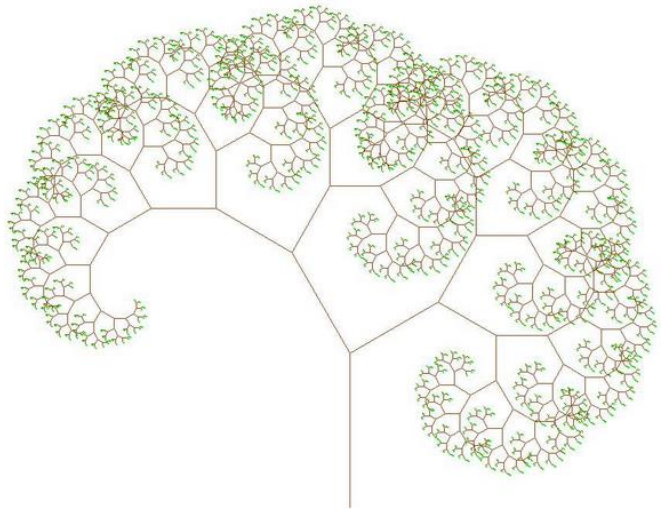
以一个正整数n为例，如果n为偶数，就将它变为n/2，如果除后变为奇数，则将它乘3加1（即3n+1）。不断重复这样的运算，经过有限步后，必然会得到1。本实例要求编写代码，计算用户输入的数据按照以上规律经多少次运算后可变为1。

6.6.2 饮品自动售货机

随着无人新零售经济的崛起，商场、车站、大厦等各种场所都引入了无人饮品自动售货机，方便人们选购自己想要的饮品。购买者选择想要的饮品，通过投币或扫码的方式支付，支付成功后从出货口取出饮品。本实例要求编写代码，利用函数实现具有显示饮品信息、计算总额等功能的程序。

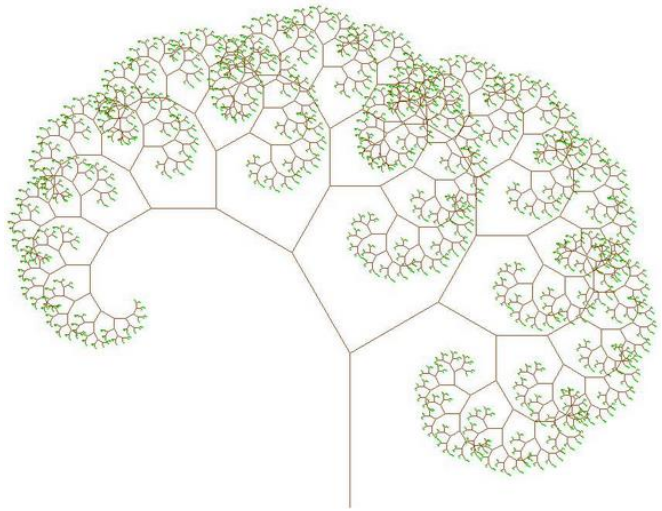
目录页

6.7.1 递归函数



函数在定义时可以直接或间接地调用其他函数。若函数内部调用了自身，则这个函数被称为递归函数。

6.7.1 递归函数



递归函数在定义时需要满足两个基本条件：一个是递归公式，另一个是边界条件。其中:

- 递归公式是求解原问题或相似的子问题的结构；
- 边界条件是最小化的子问题，也是递归终止的条件。

6.7.1 递归函数

递归函数的执行可以分为以下两个阶段：

1.**递推**：递归本次的执行都基于上一次的运算结果。

2.**回溯**：遇到终止条件时，则沿着递推往回一级一级地把值返回来。

递归函数的一般定义**格式**如下所示：

```
def函数名([参数列表]):
        if 边界条件:
                rerun 结果
        else:
                return 递归公式
```

递归经典应用

阶乘n!

n! = 1 * 2 * 3 * … * n，可以分为以下两种情况：

1. 当n=1时，所得的结果为1。
2. 当n>1时，所得的结果为n*(n-1)!。

兔子数列

兔子一般在出生两个月之后就有了繁殖能力，每对兔子每月可以繁殖一对小兔子，假如所有的兔子都不会死，试问一年以后一共有多少对兔子?

汉诺塔

有三根杆子，初始A杆有N个圆盘，移动圆盘到C杆，要求每次只能移动一个圆盘、大盘不能叠在小盘上面。问一共需要移动多少次?

归并排序

归并排序是一种基于归并算法的排序方法，该方法采用分治策略：先将待排序的序列划分成若干长度为1的子序列……

递归经典应用

阶乘n!

n! = 1 * 2 * 3 * … * n，可以分为以下两种情况：

1. 当n=1时，所得的结果为1。 边界条件
2. 当n>1时，所得的结果为n*(n-1)!。 递归公式

实现

```
def func(num):
    if num == 1:
        return 1
    else:
        return num * func(num - 1)
num = int(input("请输入一个整数："))
result = func(num)
print("5!=%d"%result)
```

6.7.1 递归函数

```
def func(5):
    if num == 1:
        return 1
    else:
        return 5 * func(4)
```

```
def func(4):
    if num == 1:
        return 1
    else:
        return 4 * func(3)
```

```
def func(3):
    if num == 1:
        return 1
    else:
        return 3 * func(2)
```

```
def func(2):
    if num == 1:
        return 1
    else:
        return 2 * func(1)
```

```
def func(1):
    if num == 1:
        return 1
```

6.7.2 匿名函数

匿名函数是一类无需定义标识符的函数，它与普通函数一样可以在程序的任何位置使用。Python中使用lambda关键字定义匿名函数，它的语法格式如下：

lambda <形式参数列表> :<表达式>

6.7.2 匿名函数

匿名函数与普通函数的主要区别如下：

- 普通函数在定义时有名称，而匿名函数没有名称；
- 普通函数的函数体中包含有多条语句，而匿名函数的函数体只能是一个表达式；
- 普通函数可以实现比较复杂的功能，而匿名函数可实现的功能比较简单；
- 普通函数能被其他程序使用，而匿名函数不能被其他程序使用。

6.7.2 匿名函数

定义好的匿名函数不能直接使用，最好使用一个变量保存它，以便后期可以随时使用这个函数。

示例

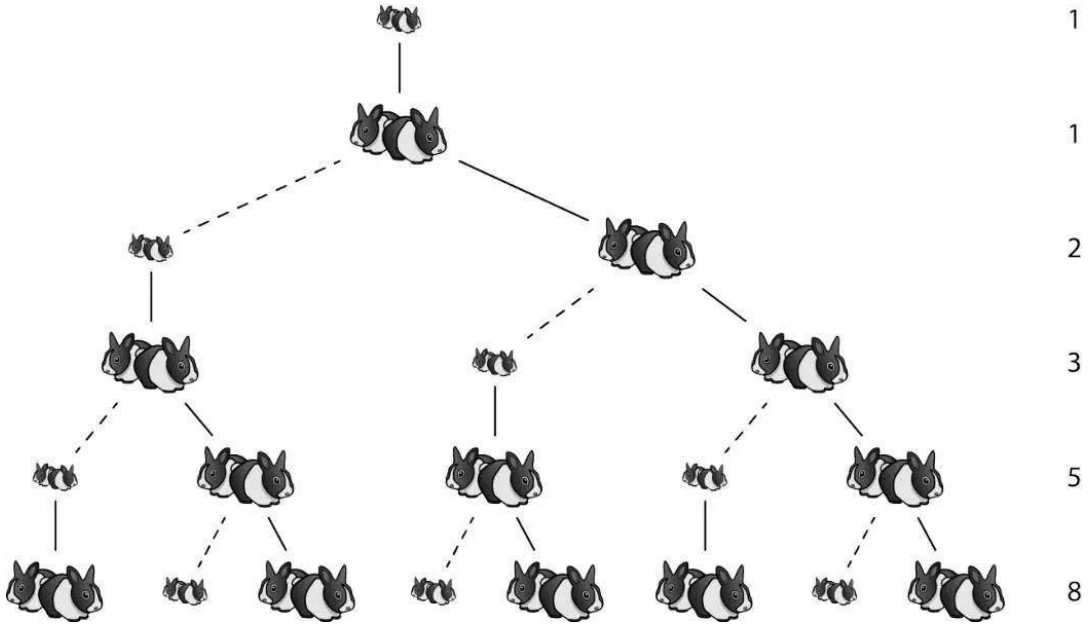
```
# 定义匿名函数，并将它返回的函数对象赋值给变量temp
temp = lambda x : pow(x, 2)
temp(10)
```

目录页

6.8.1 兔子数列

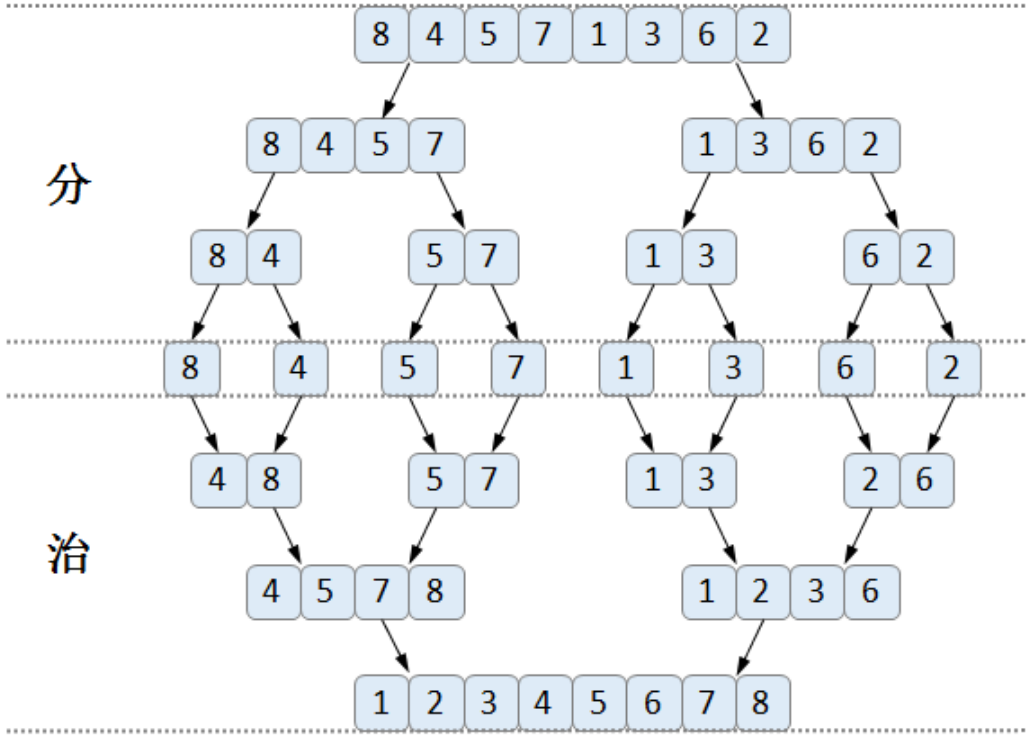
兔子一般在出生两个月之后就有了繁殖能力，每对兔子每月可以繁殖一对小兔子，假如所有的兔子都不会死，试问一年以后一共有多少对兔子？

本实例要求编写代码，利用递归实现根据月份计算兔子总数量的功能。



6.8.2 归并排序

先将待排序的序列划分成若干长度为1的子序列，依次将两个子序列排序后合并成长度为2的子序列；再依次将两个子序列排序后合并成长度为4的子序列，直至合并成最初长度的序列为止，得到一个排序后的序列。本实例要求编写代码，利用递归实现归并排序算法。



目录页

6.9 阶段案例——学生管理系统

本案例要求开发一个具有添加、删除、修改、查询学生信息及退出系统功能的简易版的学生管理系统，系统的功能菜单如图所示。

```
==================================
学生管理系统 V10.0
1.添加学生信息
2.删除学生信息
3.修改学生信息
4.查询所有学生信息
0.退出系统
==================================
```

6.10 本章小结

本章主要讲解了函数的相关知识，包括函数概述、函数的定义和调用、函数参数的传递、函数的返回值、变量作用域、特殊形式的函数，此外本章结合实训案例演示了函数的用法。通过本章的学习，希望读者能深刻地体会到函数的便捷之处，熟练地在实际开发中应用函数。