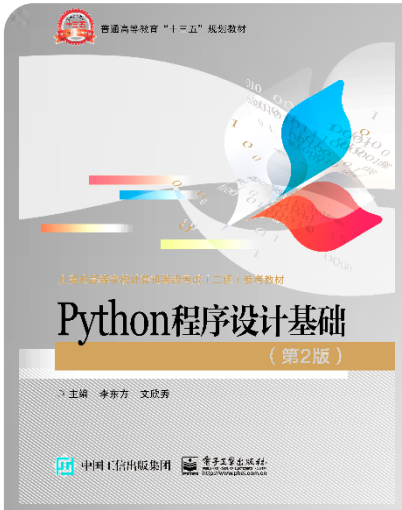
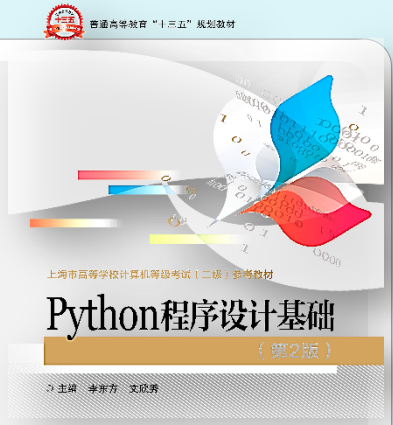


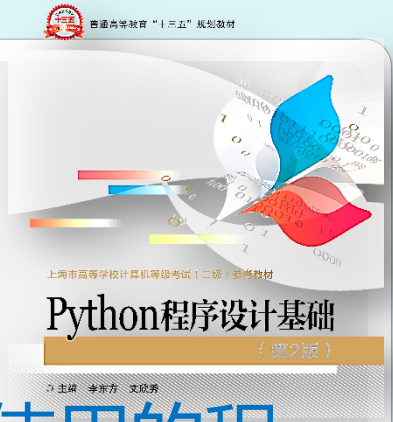
第6章 函数

本章教学目标：

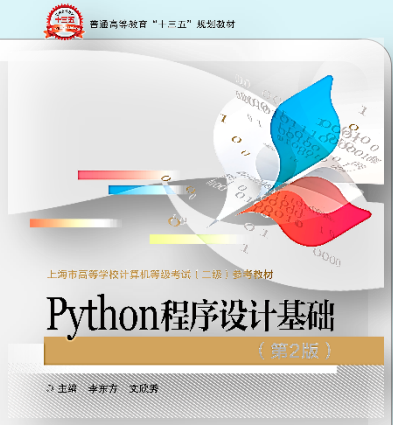
- 掌握函数的声明与调用。
- 理解并掌握函数的参数传递。
- 理解变量的作用域。
- 理解匿名函数的声明和调用。
- 了解函数的递归。
- 了解生成器、装饰器和闭包等函数高级应用。



6.1 函数的定义与调用



- 在程序设计中，常需要将一些经常重复使用的程序代码定义为函数，方便重复调用执行，以提高程序的模块化和代码的重复利用率，这就是自定义函数。



6.1 函数的定义与调用

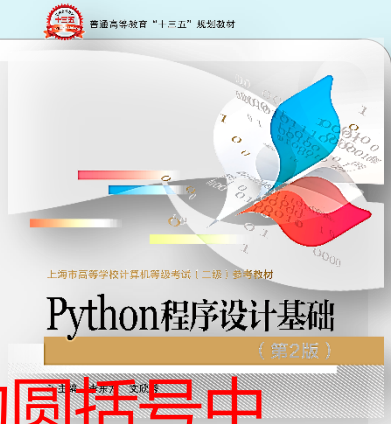
- 函数的声明
  - 函数名、参数和函数体组成
  - 自定义函数用def关键字声明
  - 函数的命名原则与变量命名相同
  - 函数语句使用缩进表示与函数体的隶属关系。

```
def <函数名> ([形式参数列表]):
    <执行语句>
    [return <返回值>]
```

```
def myfunc(x,y):
    return x+y
```

占位函数：

```
def emptfunc():
    pass
```



6.1 函数的定义与调用

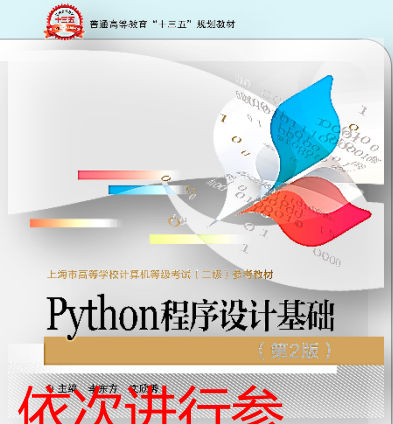
- 函数的调用

在语句中直接使用函数名，并在函数名之后的圆括号中传入参数，多个参数之间以半角逗号隔开

```
def myfunc(x,y):
    return x+y
a,b=2.5,3.6
print('%0.2f+%0.2f=%0.2f'
%(a,b,myfunc(a,b)))
```

在调用函数时，实际传递给函数的参数称为实际参数(argument)，简称实参；调用时，即使不需要传入实际参数，也要带空括号。

在函数定义时，用来接收调用该函数时传入的参数称为形式参数(parameter)，简称形参；即使没有参数，也要带空括号。



6.2 参数的传递

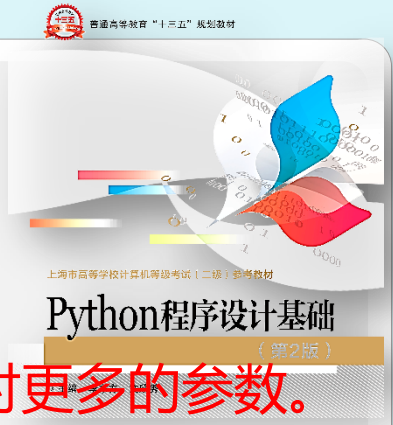
- 参数按位置依次传递

调用函数时，按照函数声明时参数的原有顺序（位置）依次进行参数传递。

- 参数赋值和参数默认值传递

在调用函数时，可在调用函数名后的圆括号内用“**形参变量名=参数值**”的方式传入参数，这种方式不必按照定义函数时原有的参数顺序。在定义函数时，可以同时定义默认参数。调用该函数时，如果没有传递同名形式参数，则会使用默认参数值。

```
def myfunc(x,y=2):
    return x+y
a,b=2.5,3.6
print('%0.2f+默认值=%0.2f' %(a,myfunc(x=a)))
print('%0.2f+%0.2f=%0.2f' %(a,b,myfunc(y=b,x=a)))
```



6.2 参数的传递

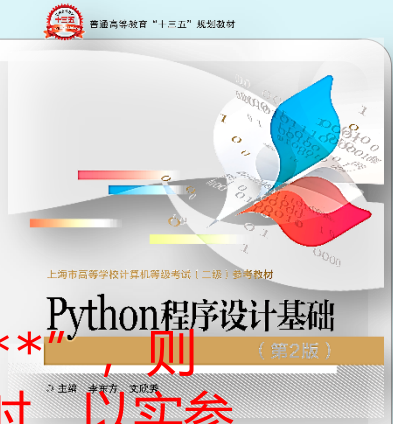
- 元组类型变长参数传递

使用可变长参数可让Python的函数处理比初始声明时更多的参数。在函数声明时，若在某个参数名称前面加一个星号“*”，则表示该参数是一个元组类型可变长参数。在调用该函数时，依次将必须赋值的参数赋值完毕后，将继续依次从调用时所提供的参数元组中接收元素值为可变长参数赋值。

如果在函数调用时没有提供元组类型参数，相当于提供了一个空元组，即不必传递可变长参数。

```
def printse_series(d,*dtup):
    print('必须参数：',d)
    if len(dtup)!=0:
        print('元组参数：',end=' ')
        for i in dtup:
            print(i,end=' ')
```

printse_series(10)
printse_series(10,20,30,40)



6.2 参数的传递

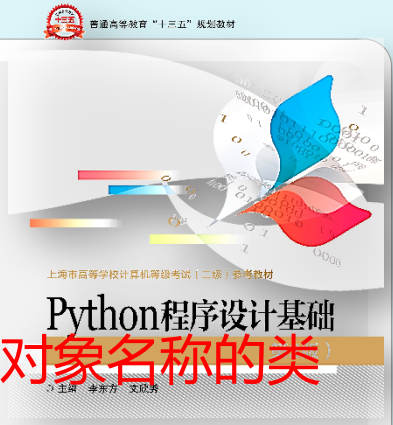
- 字典类型变长参数传递

在函数声明时，若在其某个参数名称前面加两个星号“**”，则表示该参数是一个字典类型可变长参数。在调用该函数时，以实参变量名等于字典值的方式传递参数，由函数自动按字典值接收，实参变量名以字符形式作为字典的键。

如果在函数调用时没有提供字典类型参数，则相当于提供了一个空字典，即不必传递可变长参数。

```
def printse_series2(d,*dtup,**ddic):
    print('必需参数：',d)
    if len(dtup)!=0:
        print('元组：',end=' ')
        for i in dtup:
            print(i,end=' ')
    if len(ddic)!=0:
        print('\n字典：',ddic)
        for k in ddic:
            print('%s对应%s' %(k,ddic[k]))
```

printse_series2(1,2,3,4,5,6,x=10,y=20,z=30)

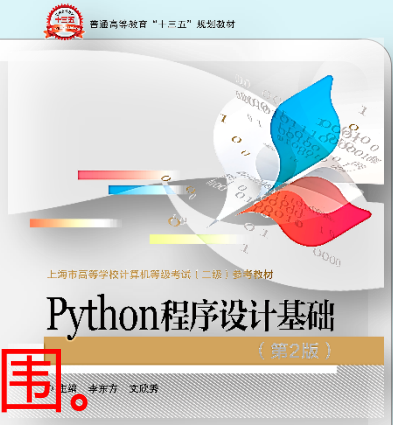


6.2 参数的传递

- 高阶函数

  能够接受将函数对象名称作为参数传入的函数，这里对象名称的类型是函数而不是字符串。

```
def add(x,y):
    return x+y
def subtract(x,y):
    return x-y
def myfunc(x,y,f):     # 形参f的类型为函数对象
    return f(x,y)
a,b=5,2
method=add         # 注意，add不加引号
print('%s：参数1为%d，参数2为%d，结果为%d' %(method,a,b,
                                    myfunc(a,b,method)))
method=subtract
print('%s：参数1为%d，参数2为%d，结果为%d' %(method,a,b,
                                        myfunc(a,b,method)))
```



6.2 参数的传递

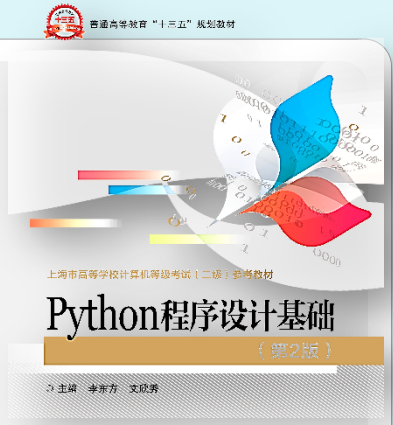
- 函数中变量的作用域
  在程序中能够对该变量进行读/写操作的范围。
  - **L**ocal　函数中定义的变量
  - **E**nclosing　嵌套中父级函数的局部作用域变量
  - **G**lobal　模块级别定义的全局变量
  - **B**uilt-in　内置模块中的变量

程序执行对变量的搜索和读/写时，优先级由近及远，即：*函数中定义的变量>嵌套中父级函数的局部作用域变量>模块级别定义的全局变量>内置模块中的变量*

Python允许出现同名变量。若具有相同命名标识的变量出现在不同的函数体中，则各自代表不同的对象，既不相互干扰，也不能相互访问；若具有相同命名标识的变量在同一个函数体中或具有函数嵌套关系，则不同作用域的变量也各自代表不同的对象，程序执行时按优先级进行访问。

6.2 参数的传递

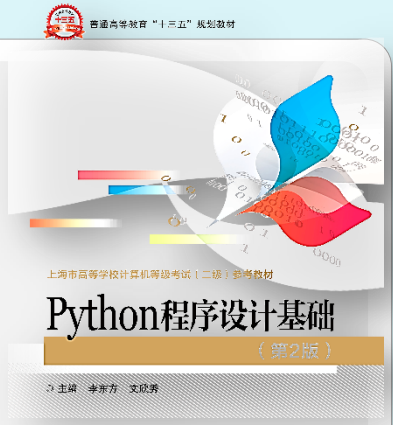
- 【例6-6】 变量作用域测试。



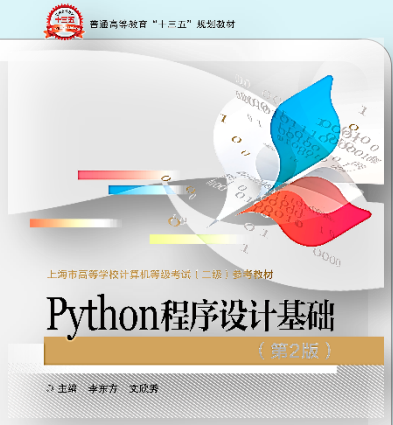

```
x = 0  # global
def outer():
    x = 1  # enclosing
    def inner():
        x = 2  # local
        print('local:  x=',x)
    inner()
    print('enclosing:  x=',x)
outer()
print('global:  x=',x)
```

6.2 参数的传递

- 【例6-7】 全局变量声明测试。




```
sum = 0
def func():
    global sum    # 用global关键字声明对全局变量的改写操作
    print(sum)    # 累加前
    for i in range(5):
        sum+=1
    print(sum)    # 累加后

func()
print(sum)        # 观察执行函数后全局变量发生变化
```



6.3 匿名函数

匿名函数就是没有实际名称的函数。

Python使用lambda来创建匿名函数

在lambda表达式中封装简单的逻辑

<函数对象名>=lambda <形式参数列表>:<表达式>

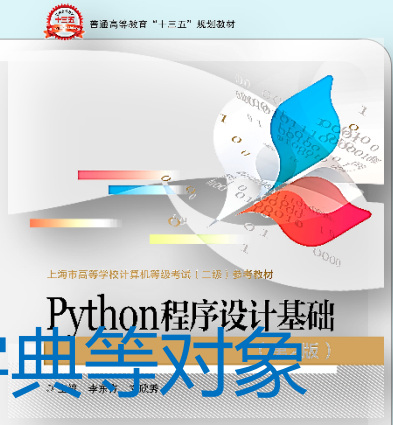
匿名函数适合于处理不再需要在其他位置复用代码的函数逻辑，可以省去函数的定义过程和考虑函数的命名，让代码更加简洁，可读性更好。

```
func=lambda x,y:x+y
a,b=2.5, 3.6
sum=func(a,b)
```

(lambda x,y:x+y)(2.5, 3.6)

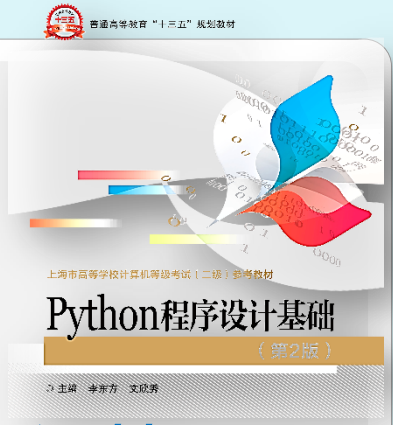
mymax=lambda x,y: x if x>=y else y

sorted函数



- sorted函数对字符串，列表，元组，字典等对象进行排序操作。
- sort是应用在list上的方法，sorted可以对更多的数据类型进行排序操作。
- 即便都是对列表操作，list的sort方法返回的是对已经存在的列表进行操作，而内建函数sorted返回的是一个新的list，而不是在原来的基础上进行的操作。

sorted函数语法

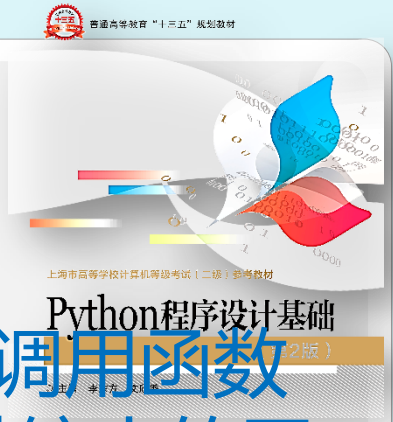


- sorted(iterable[,key[, reverse]])
- iterable -- 序列，如字符串，列表，元组等。
- key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。
- reverse -- 排序规则
- reverse = True 降序， reverse = False 升序（默认）。

表格排序

```
>>>students = [('江幸',89, 15), ('方鹏',80, 14), ('陈可', 85, 14)]
#第二个分量是成绩，第三个分量是年龄

>>>print(sorted(students, key=lambda s: s[2]))
# 按年龄从小到大排序
[(' 方鹏', 80, 14), ('陈可', 85, 14), ('江幸', 89, 15)]

>>>print(sorted(students, key=lambda s: s[1], reverse=True))
# 按成绩从大到小降序
[('江幸', 89, 15), ('陈可', 85, 14), ('方鹏', 80, 14)]
```

姓名	分数	年龄
江幸	89	15
方鹏	80	14
陈可	85	14

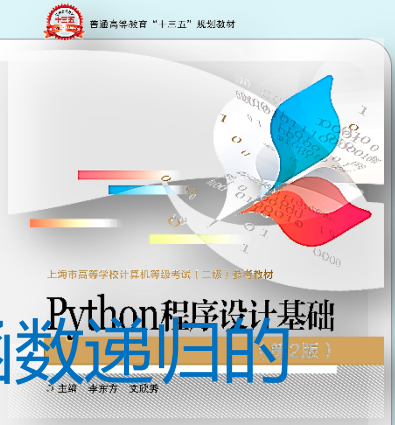


6.4 函数的递归

递归（recursion）是一种直接或者间接调用函数自身的算法，其实质是把问题分解成规模缩小的同类子问题，然后递归调用表示问题的解。

能够设计成递归算法的问题必须满足两个条件：

- 能找到反复执行的过程（调用自身）
- 能找到跳出反复执行过程的条件（递归出口）



6.4 函数的递归

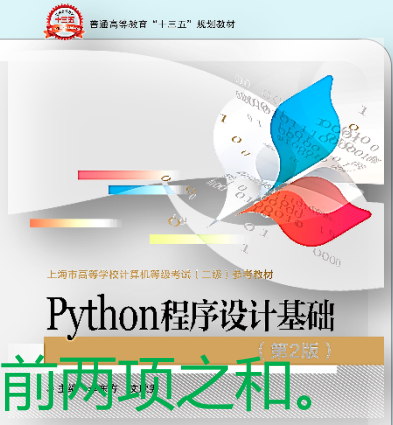
【例6-8】 设n为大于等于1的正整数，用函数递归的方法求阶乘n!

分析：n!可表示为：

$$n! = \begin{cases} 1 & n = 1 \\ n(n-1)! & n > 1 \end{cases}$$

$(n-1)!$可表示为$(n-1)(n-2)!$由此可以设计一个计算阶乘的函数recursive(n)，调用自己并返回n* recursive(n-1)。

```
def recursive(n):
    if n==1:
        return 1
    else:
        return n*recursive(n-1)
a=5
print('%d!=%d' %(a,recursive(a)))
```



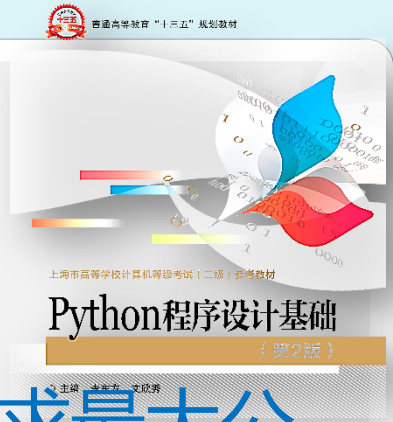
6.4 函数的递归

【例6-9】计算Fibonacci数列第15项的值。

分析：Fibonacci数列除前两项外，每项的值均等于前两项之和。由此可设计函数Fibonacci(i)用递归表达：

Fibonacci(i-1)+Fibonacci(i-2)

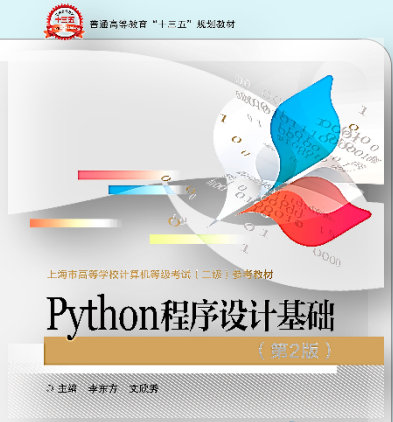
```
def Fibonacci(i):
    if i==0:
        return 0
    elif i==1:
        return 1
    else:
        return Fibonacci(i-1)+Fibonacci(i-2)

n=15
print('Fibonacci数的第%d项为%d' %(n,Fibonacci(n)))
```



6.4 函数的递归

【例6-10】 辗转相除法（欧几里德法）求最大公约数

```
def gcd(a,b):
    if b==0:
        return a
    else:
        return gcd(b,a%b)
a=162
b=189
print('%d与%d的最大公约数是%d'%(a,b,gcd(a,b)))
```

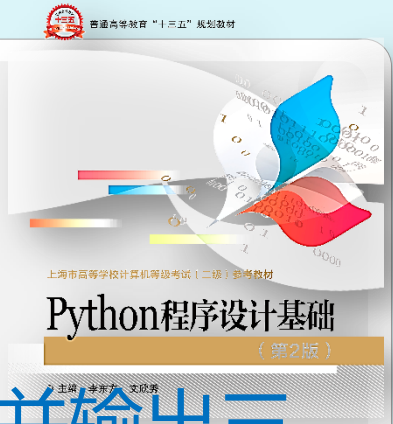


6.5 函数的高级应用

- 生成器（generator）
  - 能够按照解析表达式逐次产生出数据集合中数据项元素的函数。
  - 生成器函数与普通函数的差别主要在于：

生成器函数体中用yield关键字生成数据项，生成器函数用循环遍历时，可以用__next__()方法获取yield生成的数据项

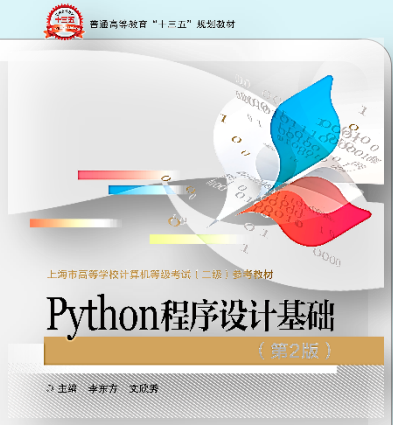
生成器函数与普通函数的执行流程不同。普通函数是顺序执行的，遇到return语句或最后一行语句就返回。而生成器函数在每次调用__next__()方法的时候才执行，遇到yield语句返回，再次执行时不是从头开始，而是从上次返回的yield语句处继续执行。



6.5 函数的高级应用

【例6-11】 用生成器产生连续偶数序列并输出三次测试运行结果。

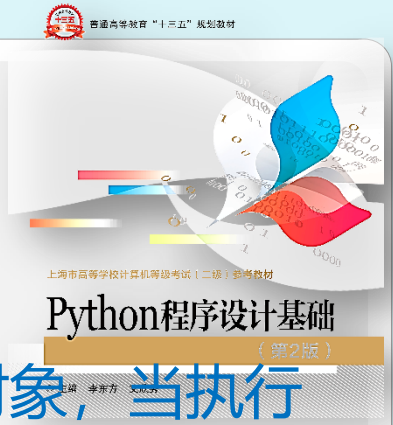
```
def generator_even():
    for i in range(1,11):
        print('第%d步' %i)
        yield i*2
g=generator_even()
print(g.__next__())
print(g.__next__())
print(g.__next__())
```



6.5 函数的高级应用

- 装饰器与闭包
  - 装饰器(decorator)
    - 可以在不必改动原有函数的前提下增加功能，经常被用于事务处理、日志记录、验证权限、调试测试等有需求的场景。

```
def decorator(f):
    def new_f(x,y):
        print('参数1为%d，参数2为%d' %(x,y))
        return f(x,y)
    return new_f
@decorator
def add(x,y):
    return x+y
print(add(2,3))
```



6.5 函数的高级应用

- 闭包(closure)
  - 将函数的语句和执行环境打包在一起得到的对象，当执行嵌套函数时，闭包将获取内部函数所需的整个环境，嵌套函数可以使用外层函数中的变量而不需要通过参数引入。

```
def outer(x):
    def inner(y):
        return x+y
    return inner

f=outer(5)
print(f(20))
```