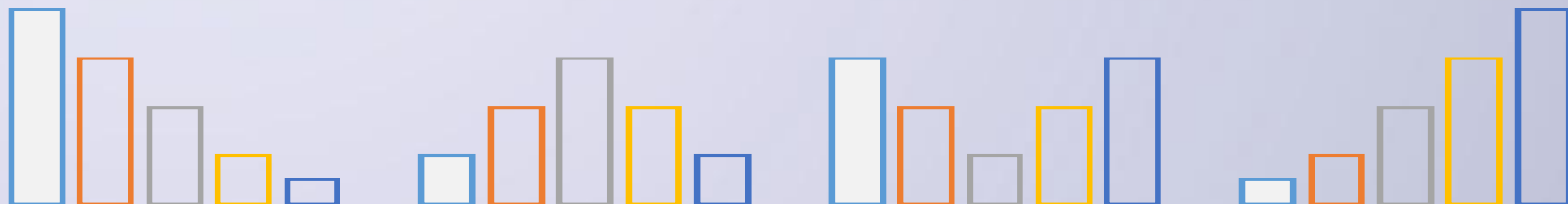


Python语言程序设计

成都信息工程大学区块链产业学院

刘硕

第5章 函数和代码的复用





前情回顾

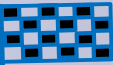
数字类型及操作

- 整数类型的无限范围及4种进制表示
- 浮点数类型的近似无限范围、小尾数及科学计数法
- +、-、*、/、//、%、**、二元增强赋值操作符
- abs()、divmod()、pow()、round()、max()、min()
- int()、float()、complex()



字符串类型及操作

- 正向递增序号、反向递减序号、<字符串>[M:N:K]
- +、*、len()、str()、hex()、oct()、ord()、chr()
- .lower()、.upper()、.split()、.count()、.replace()
- .center()、.strip()、.join() 、.format()格式化

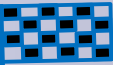


```
#TextProBarV3.py  import time  scale =  
                    50  
print("执行开始".center(scale//2, "-"))  
    start = time.perf_counter()  
  
for i in range(scale+1):  a = '*' * i  
b = '.' * (scale - i)  
c = (i/scale)*100  
dur = time.perf_counter() - start  
print("\r{:^3.0f}%[{}->{}]{:.2f}s".format(c,a,b,dur),end=' ')  
    time.sleep(0.1)  
print("\n"+"执行结束".center(scale//2, '-'))
```



程序的分支结构

- 单分支 *if* 二分支 *if-else* 及紧凑形式
- 多分支 *if-elif-else* 及条件之间关系
- *not and or* *> >= == <= < !=*
- 异常处理 *try-except-else-finally*





File Edit Format Run Options Window Help

```
while True:
    height, weight = eval(input("请输入身高(米)和体重(公斤): "))
    bmi = weight / pow(height, 2)
    print("BMI 数值为: {:.2f}".format(bmi))
    who, nat = "", ""
    if bmi < 18.5:
        who, nat = "偏瘦", "偏瘦"
    elif 18.5 <= bmi < 24:
        who, nat = "正常", "正常"
    elif 24 <= bmi < 25:
        who, nat = "正常", "偏胖"
    elif 25 <= bmi < 28:
        who, nat = "偏胖", "偏胖"
    elif 28 <= bmi < 30:
        who, nat = "偏胖", "肥胖"
    else:
        who, nat = "肥胖", "肥胖"
    name = input("The player is ")
    if who != nat:
        print(name + " cried and blamed that it is unfair!")
    else:
        print(name + " feeled that it is fair.")
    print("BMI 指标为:国际' {0}', 国内' {1}'".format(who, nat)+"\n")
```

===== RESTART: E:/liushuo/currybmi.py =====

请输入身高(米)和体重(公斤): 1.91,90.7

BMI 数值为: 24.86

The player is Curry

Curry cried and blamed that it is unfair!

BMI 指标为:国际' 正常', 国内' 偏胖'

请输入身高(米)和体重(公斤): 2.11,106.6

BMI 数值为: 23.94

The player is Durant

Durant feeled that it is fair.

BMI 指标为:国际' 正常', 国内' 正常'

请输入身高(米)和体重(公斤):

程序的循环结构

- *for...in* 遍历循环: 计数、字符串、列表、文件...
- *while* 无限循环
- *continue* 和 *break* 保留字: 退出当前循环层次
- 循环 *else* 的高级用法: 与 *break* 有关

```
#CalPiV2.py
```

```
from random import random
```

```
from time import perf_counter
```

```
DARTS = 1000*1000
```

```
hits = 0.0
```

```
start = perf_counter()
```

```
for i in range(1, DARTS+1): x, y =  
random(), random()
```

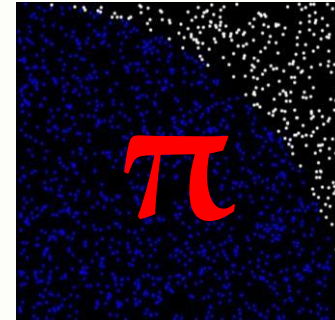
```
dist = pow(x ** 2 + y ** 2, 0.5)
```

```
if dist <= 1.0: hits = hits + 1
```

```
pi = 4 * (hits/DARTS)
```

```
print("圆周率值是: {}".format(pi))
```

```
print("运行时间是: {:.5f}s".format(perf_counter()-start))
```



目录

5.1 函数的基本使用

5.2 形参与实参

5.3 参数类型

5.4 return语句

5.5 变量作用域

5.6 lambda表达式

5.7 案例精选

5.8 高级话题

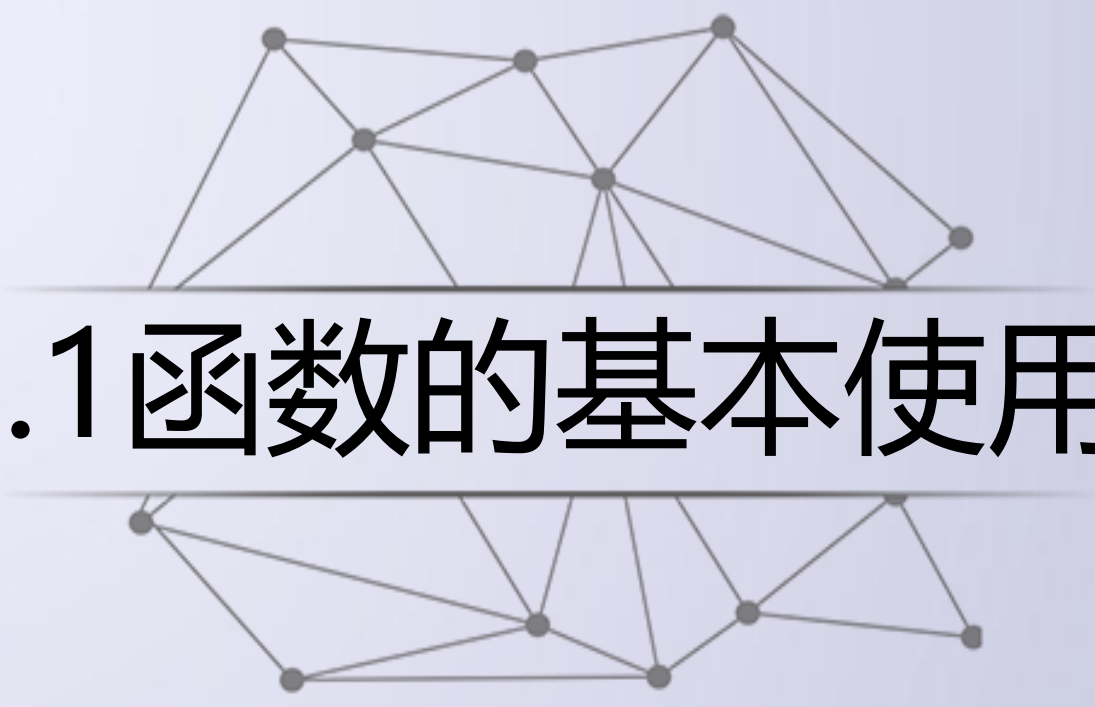
5.9 datetime库的使用

5.10 七段数码管绘制

5.11 代码复用和模块化设计

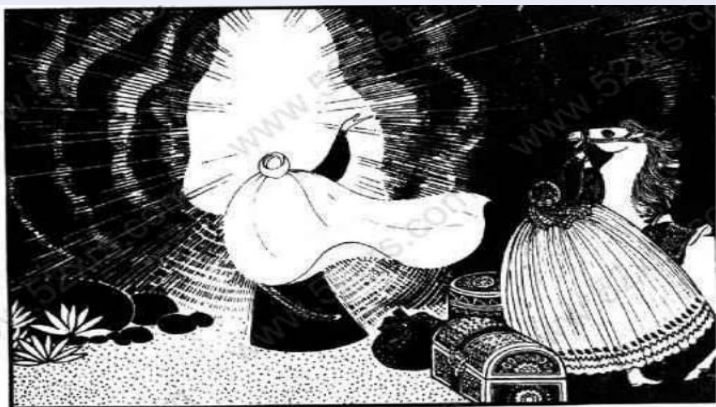
5.12函数的递归

5.13科赫曲线绘制



5.1 函数的基本使用

阿里巴巴和四十大盗-开门函数



6 一个强盗头子走到山洞的一块大石头前面，说：“开门吧，胡麻胡麻！”他说罢，大石头应声打开，四十个强盗随着头子钻进了山洞。



9 阿里巴巴走到山洞前面，说：“开门吧，胡麻胡麻！”大石头果然应声打开了。



20 高西木听罢，摆摆手说：“你必须把到宝库的路告诉我。否则，我上衙门告你，让官家没收你的钱财，并重重地处罚你。”阿里巴巴说：“我没有做贼偷钱，不怕法官。你要到宝库，我指给你路。不过，我怕强盗会害你啊！”

强盗设置的防止遗忘的开门函数：

```
def kaimen(n="胡麻胡麻"):  
    print("开门吧，胡麻胡麻")
```

```
>>> def kaimen(n="胡麻胡麻"):  
...     print('开门吧',n)  
...  
...  
>>> kaimen()  
开门吧 胡麻胡麻  
>>> kaimen("鸡豆鸡豆")  
开门吧 鸡豆鸡豆
```


普通人的开门函数：

```
def kaimen(n):  
    print("开门吧", n)
```

```
>>> def kaimen(n):  
...     print('开门吧',n)  
...  
...  
>>> kaimen("鸡豆鸡豆")  
开门吧 鸡豆鸡豆  
>>> kaimen("扁豆扁豆")  
开门吧 扁豆扁豆
```



25 他越发慌乱，接着喊道：“开门吧，鸡豆鸡豆！开门吧，扁豆扁豆！开门吧，蚕豆蚕豆！”可是，洞门始终不开。这时候，他相信自己非死不可了。



5.1.1 函数的定义

- 函数是一段具有特定功能的、可重用的语句组，用函数名来表示并通过函数名进行完成功能调用。（**迅速打开山洞**）
 - 函数也可以看作是一段具有名字的子程序，可以在需要的地方调用执行，不需要在每个执行地方重复编写这些语句。每次使用函数可以提供不同的参数作为输入，以实现对不同数据的处理；函数执行后，还可以反馈相应的处理结果。
- 函数是一种功能抽象,一般函数表达特定功能

5.1.1 函数的定义

- 将可能需要反复执行的代码封装为函数，并在需要该功能的地方进行调用，不仅可以实现**代码复用**，更重要的是可以保证代码的**一致性**，只需要修改该函数代码则所有调用均受到影响。

没有封装，则修改代码要多处修改

- 设计函数时，应注意尽量显式地调用，而不是使用太多全局变量，这样修改一个函数的变量会影响全局。
- 在实际项目开发中，往往会把一些通用 高度相关的函数封装到一个模块中，并把这个通用模块文件放到顶层文件夹中，这样更方便管理（导入和调用，就可以使用函数）。



5.1.1 函数的定义

Python定义一个函数使用def保留字，语法形式如下：

```
def <函数名>(<参数列表>):
```

```
    <函数体>
```

```
    return <返回值列表>
```

❖ 注意事项

- ✓ 函数形参不需要声明其类型，也不需要指定函数返回值类型
- ✓ 即使该函数不需要接收任何参数，也必须保留一对空的圆括号
- ✓ 括号后面的冒号必不可少
- ✓ 函数体相对于def关键字必须保持一定的空格缩进
- ✓ Python允许嵌套定义函数

5.1.1 函数的定义

函数名 参数
 ↙ ↖
def fact(n) :

s=1

for i *in* range(1, n+1):

s*= i

return s ↖ 返回值

计算 n!

5.1.1 函数的定义

- 在编写函数时，应尽量减少副作用，尽量不要修改参数本身，不要修改除返回值以外的其他内容。

例如提供列表求和，直接返回加和的数值即可

- 应充分利用Python函数式编程的特点，让自己定义的函数尽量符合纯函数式编程的要求，例如保证线程安全、可以并行运行等等。

(函数式编程是一种古老的编程模式，就是用函数（计算）来表示程序，用函数的组合来表达程序组合的思维方式)

闭包

```
>>> def FunX(x):  
...     def FunY(y):  
...         return x*y  
...     return FunY  
...
```

```
>>> i =FunX(8)  
>>> i  
<function FunX.<locals>.FunY at 0x0000018335584550>
```

```
>>> i(10)  
80  
>>> FunX(8)(10)  
80
```

5.1.1 函数的定义

$$y = f(x)$$

- 函数定义时，所指定的参数是一种**占位符**
- 函数定义后，如果不经过**调用**，不会被执行
- 函数定义时，参数是**输入**、函数体是**处理**、结果是**输出 (IPO)**



5.1.1 函数的定义

微实例5.1：生日歌。

过生日时要为朋友唱生日歌，歌词为：

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear <名字>

Happy birthday to you!

编写程序为Mike和Lily输出生日歌。最简单的实现方法是重复使用print()语句



5.1.1 函数的定义

最简单的实现方法是重复使用print()语句，如下：

```
1 print("Happy birthday to you!")
2 print("Happy birthday to you!")
3 print("Happy birthday, dear Mike!")
4 print("Happy birthday to you!")
```



5.1.1 函数的定义

微实例5.1

m5.1HappyBirthday.py

```
1  def happy():
2      print("Happy birthday to you!")
3  def happyB(name):
4      happy()
5      happy()
6      print("Happy birthday, dear {}".format(name))
7  happy()
8  happyB("Mike")
9  print()
10 happyB("Lily")
```

>>>

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear
Mike!
Happy birthday to you!

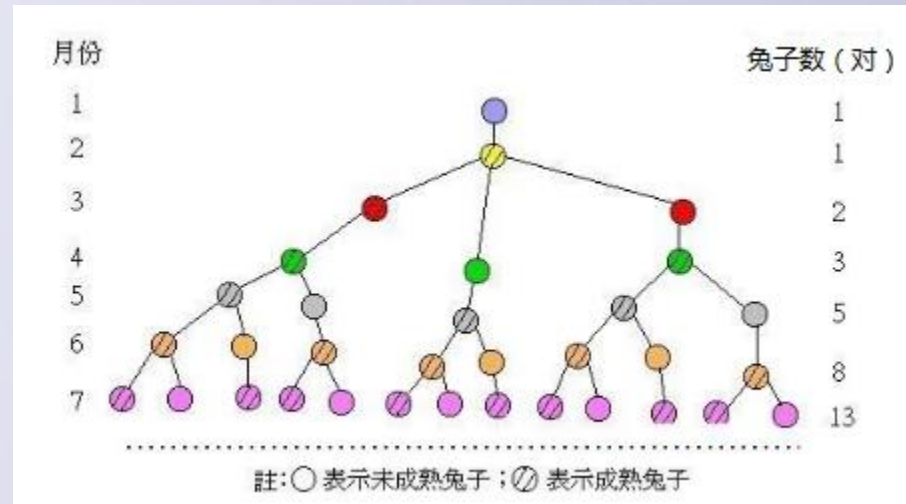
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear
Lily!
Happy birthday to you!
```

5.1.1函数的定义

- 生成斐波那契数列的函数定义和调用

```
def fib(n):  
    a, b = 0, 1  
    while a < n:  
        print(a, end=' ')  
        a, b = b, a+b  
    print()
```

fib(1000)



5.1.1 函数的定义

- 在定义函数时，开头部分的注释并不是必需的，但是如果为函数的定义加上这段注释的话，可以为用户提供友好的提示和使用帮助。

```
>>> def fib(n):  
    '''accept an integer n.  
    return the numbers less than n in Fibonacci sequence.'''  
    a, b = 1, 1  
    while a < n:  
        print(a, end=' ')  
        a, b = b, a+b  
    print()
```

```
>>> fib(  
    (n)  
    accept an integer n.  
    return the numbers less than n in Fibonacci sequence.
```

```
>>> help(fib)  
Help on function fib in module __main__:  
  
fib(n)  
    accept an integer n.
```

```
>>> def fib(n):  
    '''accept an integer n.'''  
    a, b = 1, 1  
    while a < n:  
        print(a, end=' ')  
        a, b = b, a+b  
    print()  
  
>>> fib.__doc__  
'accept an integer n.'
```

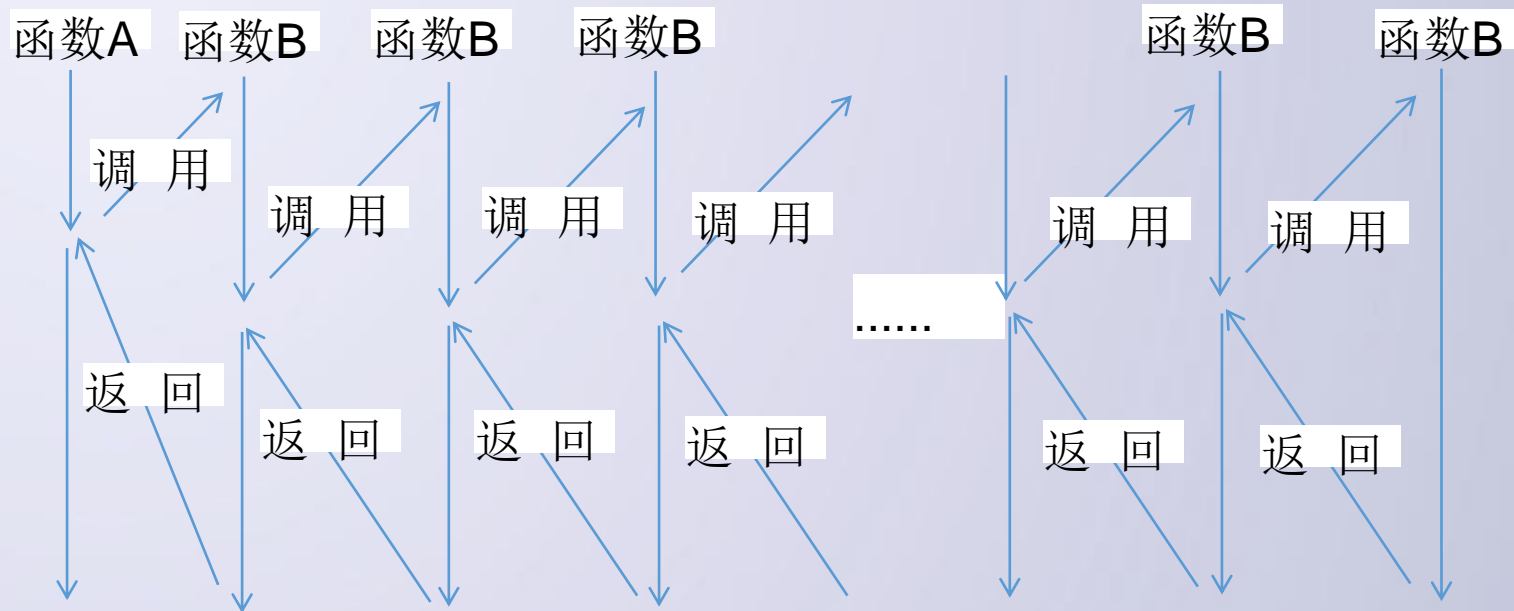

5.1.1 函数的定义

- Python是一种高级动态编程语言，变量类型是随时可以改变的。Python中的函数和自定义对象的成员也是可以随时发生改变的，可以为函数和自定义对象动态增加新成员。

```
>>> def func():  
    print(func.x)                                #查看函数func的成员x  
>>> func()                                       #现在函数func还没有成员x，出错  
AttributeError: 'function' object has no attribute 'x'  
>>> func.x = 3                                  #动态为函数增加新成员  
>>> func()  
3  
>>> func.x                                     #在外部也可以直接访问函数的成员  
3  
>>> del func.x                                 #删除函数成员  
>>> func()                                     #删除之后不可访问  
AttributeError: 'function' object has no attribute 'x'
```

5.1.1函数的定义

- 函数的**递归调用**是函数调用的一种特殊情况，函数调用自己，自己再调用自己，自己再调用自己，...，当某个条件得到满足的时候就不再调用了，然后再一层一层地返回直到该函数的第一次调用。





5.1.2函数调用的过程

程序调用一个函数需要执行以下四个步骤：

- (1) 调用程序在调用处暂停执行；
- (2) 在调用时将实参复制给函数的形参；
- (3) 执行函数体语句；
- (4) 函数调用结束给出返回值，程序回到调用前的暂停处继续执行。

5.1.2函数调用的过程

调用是运行函数代码的方式

```
def fact(n) :
```

```
    s=1
```

函数的定义

```
    for i in range(1,n+1):
```

```
        s*= i
```

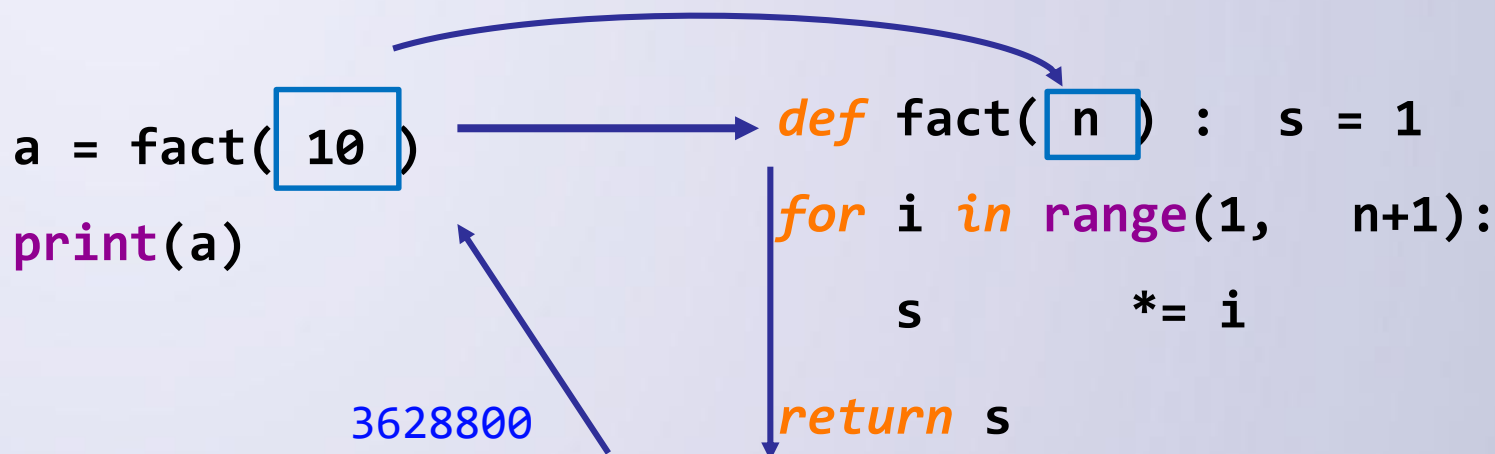
```
    return s
```

```
fact(10)
```

函数的调用

- 调用时要给出实际参数
- 实际参数替换定义中的参数
- 函数调用后得到返回值

5.1.2 函数的调用过程





5.1.2函数调用的过程

`name="Mike"`

```
happyB("Mike") → def happyB(name):  
print()             happy()  
happyB("Lily")      happy()  
                    print("Happy birthday, dear!".format(name))  
                    happy()
```

5.1.2函数调用的过程

`name="Mike"`

```
happyB("Mike")  →  def happyB(name):  
print()           happy() → def happy():  
happyB("Lily")    happy() → print("Happy birthday to you!")  
                  print("Happy birthday, dear!".format(name))  
                  happy()
```

微实例5.1中

happyB()的被调用过程

微实例5.1

m5.1HappyBirthday.py

```
1  def happy():  
2      print("Happy birthday to you!")  
3  def happyB(name):  
4      happy()  
5      happy()  
6      print("Happy birthday, dear {}".format(name))  
7  happy()  
8  happyB("Mike")  
9  print()  
10 happyB("Lily")
```



5.1.2函数调用的过程

```
name="Mike"

happyB("Mike")
print()
happyB("Lily")

def happyB(name):
    happy()
    happy()
    print("Happy birthday, dear!".format(name))
    happy()
```


5.2 形参与实参及参数传递

- 函数定义时括弧内为形参，一个函数可以没有形参，但是括弧必须要有，表示该函数不接受参数。
- 函数调用时向其传递实参，将实参的值或引用传递给形参。
- 在定义函数时，对参数个数并没有限制，如果有多个形参，需要使用逗号进行分割。

Def fire(weapon):



Def fire(weapon):



参数个数

函数可以有参数，也可以没有，但必须保留括号

def <函数名>() :

<函数体>

return<返回值>

def fact() :

print("我也是函数")

可选参数传递

函数定义时可以为某些参数指定默认值，构成可选参数

def <函数名>(<非可选参数>, <可选参数>) :

<函数体>

return <返回值>

可选参数传递

可选参数



```
def fact(n, m=1) :
```

```
    s=1
```

```
    for i in range(1,n+1):
```

```
        s*= i
```

```
    return s//m
```

```
>>> fact(10)
```

```
3628800
```

```
>>> fact(10,5)
```

```
725760
```

计算 $n!//m$

可变参数传递

函数定义时可以设计可变数量参数，即不确定参数总数量


```
def    <函数名>(<参数>, *b) :
```

```
    <函数体>
```

```
    return <返回值>
```

可变参数传递

计算 n!乘数

 可变参数

```
def fact(n,*b):  
    s=1  
    for i in range(1, n+1):  
        s *= i  
    for item in b:  
        s *= item  
    return s
```

```
>>> fact(10,3)
```

```
10886400
```

```
>>> fact(10,3,5,8)
```

```
435456000
```

参数传递的两种方式

函数调用时，参数可以按照位置或名称方式传递

```
def fact(n, m=1) :
```

```
    s = 1
```

```
    for i in range(1, n+1):
```

```
        s *= i
```

```
    return s//m
```

```
>>> fact( 10, 5 )
```

```
725760
```

```
>>> fact( m=5, n=10 )
```

```
725760
```

位置传递

名称传递

位置传递支持可变数量的参数，但是容易忘记实参的含义

名称传递不容易忘记实参的含义，但是不支持支持可变数量的参数



参数的位置和名称传递

Python提供了按照形参名称输入实参的方式，调用如下：

```
result = func(x2=4, y2=5, z2=6, x1=1, y1=2, z1=3)
```

由于调用函数时指定了参数名称，所以参数之间的顺序可以任意调整。

形参与实参

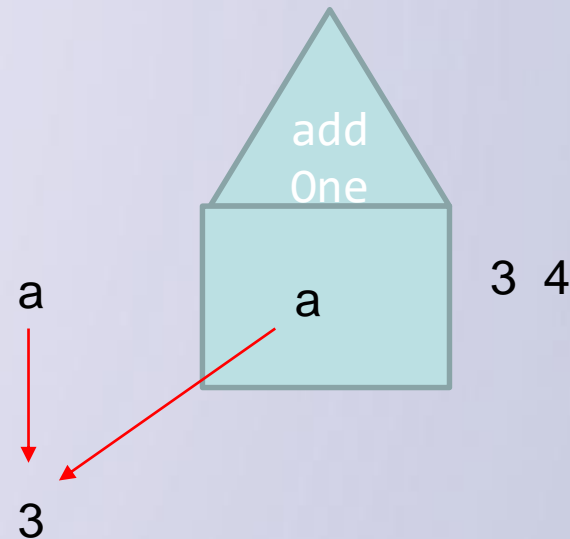
- 编写函数，接受两个整数，并输出其中最大数。

```
def printMax(a, b):  
    if a>b:  
        print(a, 'is the max')  
    else:  
        print(b, 'is the max')
```

形参与实参

- 对于绝大多数情况下，在函数内部直接修改形参的值不会影响实参。例如：

```
>>> def addOne(a):  
    print(a)  
    a += 1  
    print(a)  
>>> a = 3  
>>> addOne(a)  
3  
4  
>>> a  
3
```



形参与实参

- 在有些情况下，可以通过特殊的方式在函数内部修改实参的值，例如下面的代码。

```
>>> def modify(v):                #修改列表元素值
    v[0] = v[0]+1
>>> a = [2]
>>> modify(a)
>>> a
[3]
>>> def modify(v, item):          #为列表增加元素
    v.append(item)
>>> a = [2]
>>> modify(a,3)
>>> a
[2, 3]
```

形参与实参

- 也就是说，如果传递给函数的是可变序列，并且在函数内部使用下标或可变序列自身的方法增加、删除元素或修改元素时，修改后的结果是可以反映到函数之外的，实参也得到相应的修改。

```
>>> def modify(d): #修改字典元素值或为字典增加元素
    d['age'] = 32
>>> a = {'name': 'Python', 'age': 38, 'sex': 'Male'}
>>> a
{'age': 38, 'name': 'Python', 'sex': 'Male'}
>>> modify(a)
>>> a
{'age': 32, 'name': 'Python', 'sex': 'Male'}
```

参数类型

- 在Python中，函数参数有很多种：可以为普通参数、默认值参数、关键参数、可变长度参数等等。
- Python在定义函数时不需要指定形参的类型，完全由调用者传递的实参类型以及Python解释器的理解和推断来决定。
 -
- Python函数定义时也不需要指定函数的类型，这将由函数中的return语句来决定，如果没有return语句或者return没有得到执行，则认为返回空值None。

参数类型

- Python支持对函数参数和返回值类型的标注，`assert`可辅助完成。

```
>>> def test(x:int, y:int) -> int:
    '''x and y must be integers, return an integer x+y'''
    assert isinstance(x, int), 'x must be integer'
    assert isinstance(y, int), 'y must be integer'
    z = x+y
    assert isinstance(z, int), 'must return an integer'
    return z
```

```
>>> test(1, 2)
```

```
3
```

```
>>> test(2, 3.0)
```

#参数类型不符合要求，抛出异常

```
AssertionError: y must be integer
```

参数类型

- 位置参数（**positional arguments**）是比较常用的形式，调用函数时实参和形参的顺序必须严格一致，并且实参和形参的数量必须相同。

```
>>> def demo(a, b, c):  
    print(a, b, c)
```

```
>>> demo(3, 4, 5)                                #按位置传递参数
```

```
3 4 5
```

```
>>> demo(3, 5, 4)
```

```
3 5 4
```

```
>>> demo(1, 2, 3, 4)                                #实参与形参数量必须相同
```

```
TypeError: demo() takes 3 positional arguments but 4 were given
```



可选参数和可变数量参数

在定义函数时，有些参数可以存在默认值

```
>>>def dup(str, times = 2):  
    print(str*times)  
>>>dup("knock~")  
knock~knock~  
>>>dup("knock~", 4)  
knock~knock~knock~knock~
```


默认值参数

- 默认值参数必须出现在函数参数列表的最右端，且任何一个默认值参数右边不能有非默认值参数。

```
>>> def f(a=3,b,c=5):  
    print a,b,c
```

```
SyntaxError: non-default argument follows default argument
```

```
>>> def f(a=3,b):  
    print a,b
```

```
SyntaxError: non-default argument follows default argument
```

```
>>> def f(a,b,c=5):  
    print a,b,c
```

```
>>>
```

默认值参数

- 调用带有默认值参数的函数时，可以不对默认值参数进行赋值，也可以赋值，具有较大的灵活性。

```
>>> def say( message, times =1 ):
    print(message * times)
>>> say('hello')
hello
>>> say('hello',3)
hello hello hello
>>> say('hi',7)
hi hi hi hi hi hi hi
```

默认值参数

- 再例如，下面的函数使用指定分隔符将列表中所有字符串元素连接成一个字符串。

```
>>> def Join(List,sep=None):  
        return (sep or ' ').join(List)  
>>> aList = ['a', 'b', 'c']  
>>> Join(aList)  
'a b c'  
>>> Join(aList, ',')  
'a,b,c'
```

默认值参数

- 注意:
- ✓ 默认值参数只在函数定义时被解释一次
- ✓ 可以使用“函数名.__defaults__”查看所有默认参数的当前值

```
>>> i = 3
>>> def f(n=i):           #参数n的值仅取决于i的当前值
    print(n)
```

```
>>> f()
```

```
3
```

```
>>> i = 5                 #函数定义后修改i的值不影响参数n的默认值
```

```
>>> f()
```

```
3
```

```
>>> i=3
>>> def f(n=i):
...     print(n)
...
>>> f()
3
>>> n=5
>>> f()
3
>>> f.__defaults__
(3,)
```

关键参数

- 关键参数主要指实参，即调用函数时的参数传递方式。
- 通过关键参数，实参顺序可以和形参顺序不一致，但不影响传递结果，避免了用户需要牢记位置参数顺序的麻烦。

```
>>> def demo(a,b,c=5):  
    print(a,b,c)  
>>> demo(3,7)  
3 7 5  
>>> demo(a=7,b=3,c=6)  
7 3 6  
>>> demo(c=8,a=9,b=0)  
9 0 8
```

可变长度参数

- 可变长度参数主要有两种形式：
 - `*parameter`用来接受多个实参并将其放在一个元组中
 - `**parameter`接受关键参数并存放到字典中

可变长度参数

❖ *parameter的用法

```
>>> def demo(*p):  
    print(p)
```

```
>>> demo(1,2,3)
```

```
(1, 2, 3)
```

```
>>> demo(1,2)
```

```
(1, 2)
```

```
>>> demo(1,2,3,4,5,6,7)
```

```
(1, 2, 3, 4, 5, 6, 7)
```



可选参数和可变数量参数

在函数定义时，可以设计可变数量参数，通过参数前增加星号 (*) 实现

```
>>>def vfunc(a, *b):  
    print(type(b))  
    for n in b:  
        a += n  
    return a  
>>>vfunc(1,2,3,4,5)  
<class 'tuple'>  
15
```

```
>>> def vfunc(a, *b):  
...     print(type(b), b)  
...     for n in b:  
...         a += n  
...     return a  
...  
...  
>>> vfunc(1,2,3,4,5)  
<class 'tuple'> (2, 3, 4, 5)  
15
```


可变长度参数

❖ *parameter的用法

```
>>> def demo(**p):  
    for item in p.items():  
        print(item)
```

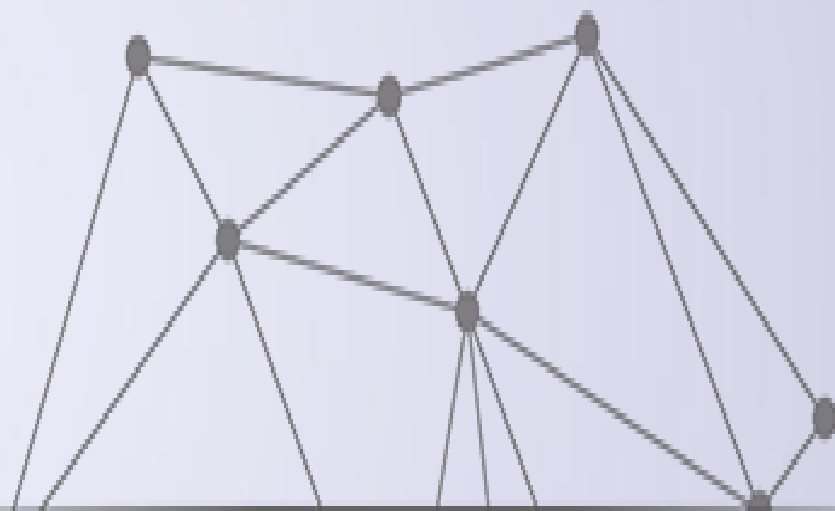
```
>>> demo(x=1,y=2,z=3)  
( 'y', 2)  
( 'x', 1)  
( 'z', 3)
```

可变长度参数

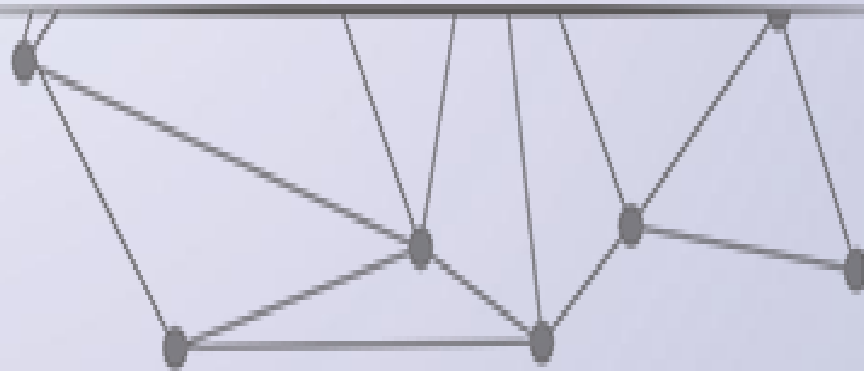
- 几种不同类型的参数可以混合使用，但是不建议这样做

```
>>> def func_4(a,b,c=4,*aa,**bb):  
    print(a,b,c)  
    print(aa)  
    print(bb)
```

```
>>> func_4(1,2,3,4,5,6,7,8,9,xx='1',yy='2',zz=3)  
(1, 2, 3)  
(4, 5, 6, 7, 8, 9)  
{'yy': '2', 'xx': '1', 'zz': 3}  
>>> func_4(1,2,3,4,5,6,7,xx='1',yy='2',zz=3)  
(1, 2, 3)  
(4, 5, 6, 7)  
{'yy': '2', 'xx': '1', 'zz': 3}
```



5.3.4函数的返回值和return 语句





函数的返回值

- return语句用来退出函数并将程序返回到函数被调用的位置继续执行。
- return语句同时可以将0个、1个或多个函数运算完的结果返回给函数被调用处的变量，例如。

```
>>>def func(a, b):  
    return a*b  
>>>s = func("knock~", 2)  
>>>print(s)  
knock~knock~
```

函数的返回值

函数调用时，参数可以按照位置或名称方式传递

```
def fact(n, m=1) :  
    s=1  
    for i in range(1, n+1):  
        s*= i  
    return s//m, n, m
```

```
>>> fact( 10, 5 )
```

元组类型

```
(725760, 10, 5)
```

```
>>> a,b,c = fact(10,5)
```

```
>>> print(a,b,c)
```

```
725760 10 5
```



函数的返回值

函数可以没有return，此时函数并不返回值，如微实例5.1的happy()函数。函数也可以用return返回多个值，多个值以元组类型保存，例如。

```
>>>def func(a, b):  
    return b,a  
>>>s = func("knock~", 2)  
>>>print(s, type(s))  
(2, 'knock~') <class 'tuple'>
```

return语句

- **return**语句用来从一个函数中返回一个值，同时结束函数。
- 如果函数没有**return**语句，或者有**return**语句但是没有执行到，或者只有**return**而没有返回值，Python将认为该函数以**return None**结束。

```
def maximum( x, y ):  
    if x>y:  
        return x  
    else:  
        return y
```

return语句

- 在调用函数或对象方法时，一定要注意有没有返回值，这决定了该函数或方法的使用法。

```
>>> a_list = [1, 2, 3, 4, 9, 5, 7]
>>> print(sorted(a_list))
[1, 2, 3, 4, 5, 7, 9]
>>> print(a_list)
[1, 2, 3, 4, 9, 5, 7]
>>> print(a_list.sort())
None
>>> print(a_list)
[1, 2, 3, 4, 5, 7, 9]
```


全局变量和局部变量

一个程序中的变量包括两类：全局变量和局部变量。

- 全局变量指在函数之外定义的变量，一般没有缩进，在程序执行全过程有效。
- 局部变量指在函数内部使用的变量，仅在函数内部有效，当函数退出时变量将不存在。


```
>>> x=3
>>> def f1():
...     print(x)
...     x=3
...     print(x)
...
>>> f1()
```

```
>>> x=3
>>> def f():
...     print(x)
... 
```

```
>>> def func2():
...     global y
...     y=3
...     print(y)
...
>>> func2()
3
>>> y
3
```

```
>>> def func2():
...     y=3
...     print(y)
...
>>> func2()
3
>>> y
```

```
>>> y
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    y
NameError: name 'y' is not defined
```



变量的返回值

```
>>>n = 1      #n是全局变量
>>>def func(a, b):
        c = a * b      #c是局部变量, a和b作为函数参数也是局部变量
        return c
>>>s = func("knock~", 2)
>>>print(c)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print(c)
NameError: name 'c' is not defined
```


这个例子说明，当函数执行完退出后，其内部变量将被释放。如果函数内部使用了全局变量呢？



变量的返回值

```
>>>n = 1      #n是全局变量
>>>def func(a, b):
        n = b      #这个n是在函数内存中新生成的局部变量，不是全局变量
        return a*b
>>>s = func("knock~", 2)
>>>print(s, n)    #测试一下n值是否改变
knock~knock~ 1
```

- 函数func()内部使用了变量n，并且将变量参数b赋值给变量n，为何全局变量n值没有改变？



变量的返回值

如果希望让func()函数将n当作全局变量，需要在变量n使用前显式声明该变量为全局变量，代码如下。


```
>>>n = 1      #n是全局变量
>>>def func(a, b):
    global n
    n = b      #将局部变量b赋值给全局变量n
    return a*b
>>>s = func("knock~", 2)
>>>print(s, n)  #测试一下n值是否改变
knock~knock~ 2
```



变量的返回值

如果此时的全局变量不是整数n，而是列表类型ls，会怎么样呢？理解如下代码。

```
>>>ls = []      #ls是全局列表变量
>>>def func(a, b):
    ls.append(b)    #将局部变量b增加到全局列表变量ls中
    return a*b
>>>s = func("knock~", 2)
>>>print(s, ls)  #测试一下ls值是否改变
knock~knock~ [2]
```



变量的返回值

如果func()函数内部存在一个真实创建过且名称为ls的列表，则func()将操作该列表而不会修改全局变量，例子如下。

```
>>>ls = []      #ls是全局列表变量
>>>def func(a, b):
    ls = []      #创建了名称为ls的局部列表变量
    ls.append(b)  #将局部变量b增加到全局列表变量ls中
    return a*b
>>>s = func("knock~", 2)
>>>print(s, ls)  #测试一下ls值是否改变
knock~knock~ []
```

变量的返回值

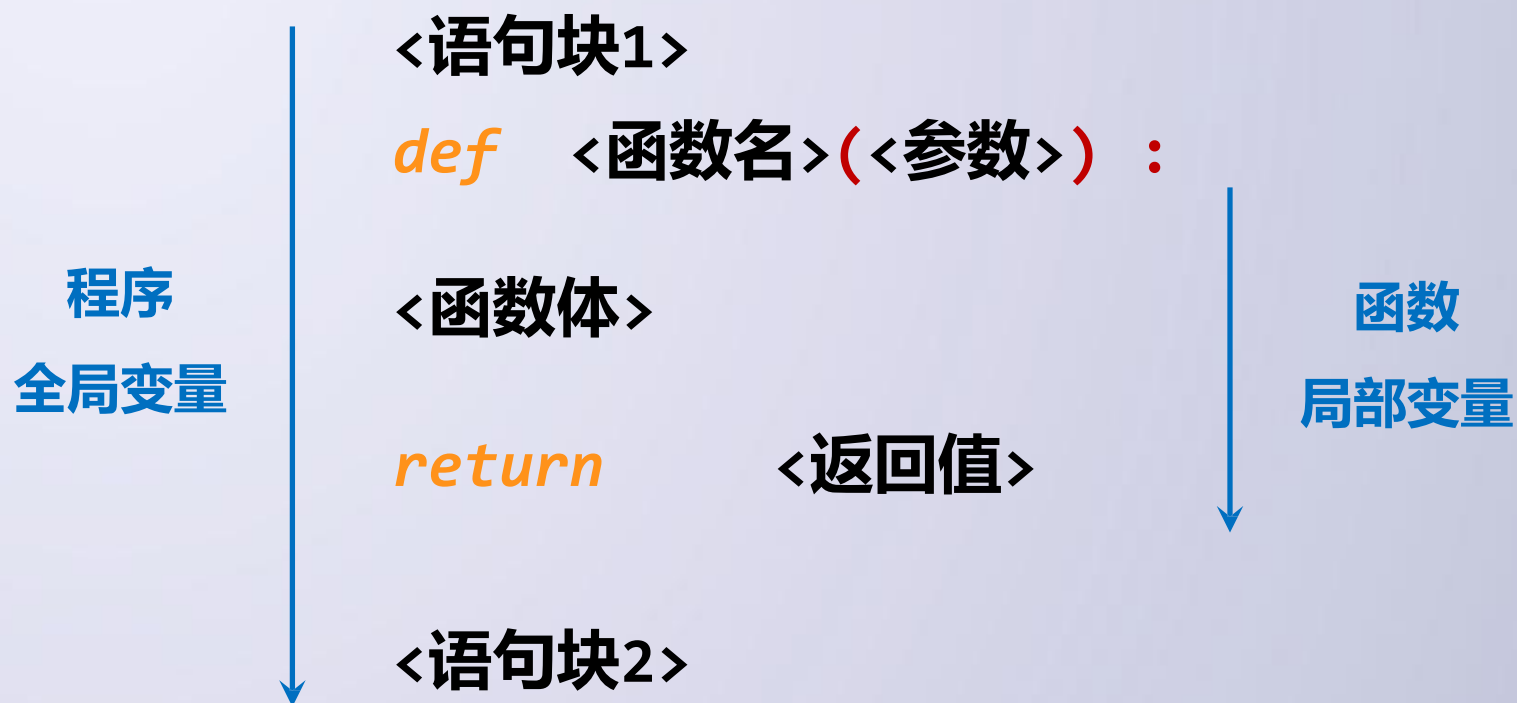
Python函数对变量的作用遵守如下原则：

- 简单数据类型变量无论是否与全局变量重名，仅在函数内部创建和使用，函数退出后变量被释放；
- 简单数据类型变量在用global保留字声明后，作为全局变量；
- 对于组合数据类型的全局变量，如果在函数内部没有被真实创建的同名变量，则函数内部可直接使用并修改全局变量的值；
- 如果函数内部真实创建了组合数据类型变量，无论是否有同名全局变量，函数仅对局部变量进行操作。

变量作用域

- 变量起作用的代码范围称为变量的作用域，不同作用域内变量名可以相同，互不影响。
- 一个变量在函数外部定义和在函数内部定义，其作用域是不同的。
- 在函数内部定义的普通变量只在函数内部起作用，称为局部变量。当函数执行结束后，局部变量自动删除，不再可以使用。
- 局部变量的引用比全局变量速度快，应优先考虑使用。

局部变量和全局变量



局部变量和全局变量

```
n, s = 10, 100
```



n和s是全局变量

```
def fact(n) :
```

```
    s = 1
```



fact()函数中的n和s是局部变量

```
    for i in range(1, n+1):
```

```
        s *= i
```

```
    return s
```

```
print(fact(n), s)
```



n和s是全局变量

运行结果

```
>>>3628800 100
```

局部变量和全局变量

规则1: 局部变量和全局变量是不同变量

- 局部变量是函数内部的占位符，与全局变量可能重名但不同
- 函数运算结束后，局部变量被释放
- 可以使用`global`保留字在函数内部使用全局变量

局部变量和全局变量

```
n, s = 10, 100
```

```
def fact(n) :
```

```
    s = 1
```

```
    for i in range(1, n+1):
```

```
        s *= i
```

```
    return s
```

```
print(fact(n), s)
```

fact()函数中s是局部变量
与全局变量s不同

此处局部变量s是3628800

此处全局变量s是100

运行结果

>>>

3628800 100

局部变量和全局变量

```
n, s = 10, 100
```

```
def fact(n) :
```

```
    global s
```

```
    for i in range(1, n+1):
```

```
        s *= i
```

```
    return s
```

```
print(fact(n), s)
```

fact()函数中使用global保留字声明

此处s是全局变量s

运行结果

>>>

362880000 362880000

此处s指全局变量s

此处全局变量s被函数修改

局部变量和全局变量

规则2: 局部变量为组合数据类型且未创建，等同于全局变量

`ls = ["F", "f"]` ← 通过使用[]真实创建了一个全局变量列表ls

`def func(a) :`
 `ls.append(a)` ← 此处ls是列表类型，未真实创建
 则等同于全局变量

`return`

`func("C")` ← 全局变量ls被修改

`print(ls)`

运行结果

`>>>`

`['F', 'f', 'C']`

局部变量和全局变量

`ls = ["F", "f"]` ← 通过使用[]真实创建了一个全局变量列表ls

def func(a) :

`ls = []`

← 此处ls是列表类型，真实创建

ls是局部变量

`ls.append(a)`

return

`func("C")`

← 局部变量ls被修改

`print(ls)`

运行结果

>>>

['F', 'f']

局部变量和全局变量

使用规则

- **基本数据类型，无论是否重名，局部变量与全局变量不同**
- **可以通过global保留字在函数内部声明全局变量**
- **组合数据类型，如果局部变量未真实创建，则是全局变量**

5.5 变量作用域

- 如果想要在函数内部给一个定义在函数外的变量赋值，那么这个变量**就不能是局部的**，其作用域**必须为全局**的，能够同时作用于函数内外，称为全局变量，可以通过`global`来定义。这分为两种情况：
 - ✓ 一个变量已在函数外定义，如果在函数内需要为这个变量赋值，并要将这个赋值结果反映到函数外，可以在函数内用`global`声明这个变量，将其声明为全局变量。
 - ✓ 在函数内部直接将一个变量声明为全局变量，在函数外没有声明，该函数执行后，将增加为新的全局变量。

5.5 变量作用域

```
>>> def demo():  
    global x  
    x = 3  
    y = 4  
    print(x,y)
```

```
>>> x = 5  
>>> demo()
```

```
3 4
```

```
>>> x
```

```
3
```

```
>>> y
```

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

```
>>> def demo():  
...     #global x  
...     x = 3  
...     y = 4  
...     print(x,y)  
...  
>>> x=5  
>>> demo()  
3 4  
>>> x  
5
```

5.5 变量作用域

```
>>> del x
>>> x
NameError: name 'x' is not defined
>>> demo()
3 4
>>> x
3
>>> y
NameError: name 'y' is not defined
```

5.5 变量作用域

- 注意：在某个作用域内只要有为变量赋值的操作，该变量在这个作用域内就是局部变量，除非使用`global`进行了声明。

```
>>> x = 3
```

```
>>> def f():
```

```
    print(x)
```

```
    x = 5
```

```
    print(x)
```

```
>>> f()
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#10>", line 1, in <module>
```

```
    f()
```

```
  File "<pyshell#9>", line 2, in f
```

```
    print(x)
```

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

#本意是先输出全局变量x的值，但是不允许这样做
#有赋值操作，因此在整个作用域内x都是局部变量

```
>>> x=3
>>> def f():
...     print(x)
...
>>> f()
3
```

5.5 变量作用域

- 如果局部变量与全局变量具有相同的名字，那么该局部变量会在自己的作用域内隐藏同名的全局变量。

```
>>> def demo():  
    x = 3          #创建了局部变量，并自动隐藏了同名的全局变量  
  
>>> x = 5  
>>> x  
5  
>>> demo()  
>>> x              #函数执行不影响外面全局变量的值  
5
```

前情回顾

- 函数的定义和形参实参
- 参数传递
- 不同类型的参数
- 函数的作用域

Python值传递和引用传递

- 形参：方法被调用时需要传递进来的参数，如：`func(int a)`中的`a`，它只有在`func`被调用期间`a`才有意义，也就是会被分配内存空间，在方法`func`执行完成后，`a`就会被销毁释放空间，也就不存在了

```
>>> def func(a):  
...     a = 20  
...     print(a)  
...  
...  
>>> a=10  
>>> func(a)  
20
```

`a=10` 中的 `a` 在被调用之前就已经创建，在调用`func`方法时，被当做参数传入，所以这个`a`是实参。

而 `func(a)` 中的 `a` 只有在 `func` 被调用时它的生命周期才开始，而在`func`调用结束之后，它也随之被释放掉，所以这个`a`是形参。

- 实参：方法被调用时是传入的实际值，它在方法被调用前就已经被初始化并且在方法被调用时传入。

变量的作用域（1）

- 全局变量、局部变量和类型成员变量
- 全局变量：在一个源代码文件中，在函数和类定义之外声明的变量
 - 全局变量的作用域为其定义的模块，从定义的位置起，直到文件结束位置
- 通过import语句导入模块，也可以通过全限定名称“模块名.变量名”访问。或者通过from...import语句导入模块中的变量并访问
- **【例8.19】全局变量定义示例**（global_variable.py）

```
TAX1 = 0.17  #税率常量17%  
TAX2 = 0.2   #税率常量20%  
TAX3 = 0.05  #税率常量5%  
PI = 3.14    #圆周率3.14
```


- **【例8.20】全局变量使用示例**

变量的作用域（2）

```
import global_variable #导入全局变量定义
def tax(x):             #根据税率常量20%计算纳税值
    return x * global_variable.TAX2
#测试代码
a = [1000, 1200, 1500, 2000]
for i in a:             #计算并打印4笔数据的纳税值
    print(i, tax(i))
```

程序运行结果如下：

```
1000 200.0 ↵
1200 240.0 ↵
1500 300.0 ↵
2000 400.0 ↵
```

局部变量

- 在函数体中声明的变量（包括函数参数）称为局部变量，其有效范围（作用域）为函数体
- 如果在一个函数中定义的局部变量（或形式参数变量）与全局变量重名，则局部变量（或形式参数变量）优先，即函数中定义的变量是指局部变量（或形式参数变量），而不是全局变量
- **【例8.21】局部变量定义示例**

```
num = 100    #全局变量
def f():
    num = 105 #局部变量
    print(num) #输出局部变量的值
#测试代码
f()
#print(num)
```

程序运行结果如下

105 ↵

100

全局语句global（1）

- 在函数体中，可以引用全局变量，但如果函数内部的变量名是第一次出现且在赋值语句之前（变量赋值），则解释为定义局部变量
- 【例8.22】函数体错误引用全局变量的示例**

```
m = 100
n = 200
def f():
    print(m+5) #引用全局变量m
    n += 10 #错误，n在赋值语句前面，解释为局部变量（不存在）
#测试代码
f()
```

程序运行结果如下：

105

Traceback (most recent call last):

File "C:\Pythonpa\ch08\f_global.py", line 7, in <module>

f()

File "C:\Pythonpa\ch08\f_global.py", line 5, in f

n += 10 #错误，n 为局部变量

UnboundLocalError: local variable 'n' referenced before assignment

非局部语句nonlocal

- 在函数体中，可以定义嵌套函数，在嵌套函数中，如果要为定义在上级函数体的局部变量赋值，可以使用nonlocal语句，表明变量不是所在块的局部变量，而是在上级函数体中定义的局部变量。nonlocal语句可以指定多个非局部变量。例如nonlocal x, y, z
- 【例8.24】非局部语句nonlocal示例

```
def outer_func():  
    tax_rate = 0.17      #上级函数体中的局部变量  
    print('outer func tax rate =', tax_rate) #输出上级函数体中局部变量的值  
    def innner_func():  
        nonlocal tax_rate  #不是所在块的局部变量，而是在上级函数体中定义的局  
部变量  
        tax_rate = 0.05 #上级函数体中的局部变量重新赋值  
        print('inner func tax rate =', tax_rate) #输出上级函数体中局部变量的值  
    innner_func()          #调用函数  
    print('outer func tax rate =', tax_rate) #输出上级函数体中局部变量的值（已  
更改）  
#测试代码  
outer_func()
```

程序运行结果如下。

```
outer func tax rate = 0.17  
inner func tax rate = 0.05  
outer func tax rate = 0.05
```

输出局部变量和全局变量 (locals和globals函数)

- 使用内置函数globals()和locals(), 可以查看并输出局部变量和全局变量列表
- **【例8.25】**局部变量和全局变量列表示例 (locals_globals.py)

程序运行结果如下：

```
{'a': 1, 'b': 2, 'k': 0, 'x': 'abc', 'y': 'xyz', 'j': 0, 'i': 0}
```

```
{'a': 1, 'b': 2, 'k': 1, 'x': 'abc', 'y': 'xyz', 'j': 1, 'i': 1}
```

```
1. 1, j: 1, k: 1  
__package__: None, __loader__: <class 'frozen_importlib.BuiltinImporter'>, '  
\\globals_and_locals.py', 'a': 1, 'b': 2, 'f': <function f at 0x00000173E205BD00>}
```

```
a=1  
b=2  
def f(a, b):  
    x = 'abc'  
    y = 'xyz'  
    for i in range(2): #i=0~1  
        j = i  
        k = i**2  
        print('局部变量是',locals())  
f(1,2)  
print('全局变量是',globals())
```



5.9datetime库的使用

datetime库概述

以不同格式显示日期和时间是程序中最常用到的功能。Python提供了一个处理时间的标准函数库datetime，它提供了一系列由简单到复杂的时间处理方法。datetime库可以从系统中获得时间，并以用户选择的格式输出。





datetime库概述

datetime库以类的方式（五个类）提供多种日期和时间表达方式：

- `datetime.date`：日期表示类，可以表示年、月、日等（属性）
- `datetime.time`：时间表示类，可以表示小时、分钟、秒、毫秒等
- `datetime.datetime`：日期和时间表示的类，功能覆盖`date`和`time`类
- `datetime.timedelta`：时间间隔有关的类
- `datetime.tzinfo`：与时区有关的信息表示类

datetime.date

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2023, 5, 19)
>>>
```

```
>>> import datetime
>>> datetime.date.today().year
2023
>>> datetime.date.today().timetuple()
<built-in method timetuple of datetime.date object at 0x00000277DAFA6ED0>
>>> datetime.date.today().timetuple()
time.struct_time(tm_year=2023, tm_mon=5, tm_mday=19, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=4, tm_yday=139, tm_isdst=-1)
>>> datetime.date.today().month
5
>>> datetime.date.today().weekday()
4
>>> datetime.date.today().isoweekday()
5
>>> datetime.date.today().isocalendar()
datetime.IsoCalendarDate(year=2023, week=20, weekday=5)
>>> datetime.date.today().isoformat()
'2023-05-19'
>>>
```

`date.timetuple()`: 返回日期对应的`time.struct_time`对象即一个元组;

`date.toordinal()`: 返回日期对应的Gregorian Calendar日期;

`date.weekday()`: 返回`weekday`, 如果是星期一, 返回0; 如果是星期二, 返回1, 以此类推;

`date.isoweekday()`: 返回`weekday`, 如果是星期一, 返回1; 如果是星期二, 返回2, 以此类推;

`date.isocalendar()`: 返回格式如(year, month, day)的元组;

`date.isoformat()`: 返回格式如'YYYY-MM-DD' 的字符串;



datetime库解析

使用datetime.now()获得当前日期和时间对象，使用方法如下：

datetime.now()

作用： 返回一个datetime类型，表示当前的日期和时间，精确到微秒。

```
>>> from datetime import datetime
>>> today = datetime.now()
>>> today
datetime.datetime(2023, 5, 19, 12, 7, 33, 378773)
```



datetime库解析

使用`datetime.utcnow()`获得当前日期和时间对应的UTC (世界标准时间) 时间对象，使用方法如下：

`datetime.utcnow()`

作用：返回`datetime`类型，表示当前日期和时间的UTC表示，精确到微秒。

```
>>> today = datetime.utcnow()  
>>> today  
datetime.datetime(2023, 5, 19, 4, 8, 8, 13592)
```



datetime库解析

`datetime.now()` 和 `datetime.utcnow()` 都返回一个 `datetime` 类型的对象，也可以直接使用 `datetime()` 构造一个日期和时间对象，使用方法如下：

`datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0)`

作用： 返回一个 `datetime` 类型，表示指定的日期和时间，可以精确到微秒。



datetime库解析

调用datetime()函数直接创建一个datetime对象，表示2030年11月6日22:33，32秒7微秒，执行结果如下：

```
>>> someday = datetime(2030,11,6,22,33,32,7)
>>> someday
datetime.datetime(2030, 11, 6, 22, 33, 32, 7)
```

程序已经有了一个datetime对象，进一步可以利用这个对象的属性显示时间，为了区别datetime库名，采用上例中的someday代替生成的datetime对象



datetime库解析

属性	描述
someday.min	固定返回 datetime 的最小时间对象， datetime(1,1,1,0,0)
someday.max	固定返回datetime的最大时间对象， datetime(9999, 12, 31, 23, 59, 59, 999999)
someday.year	返回someday包含的年份
someday.month	返回someday包含的月份
someday.day	返回someday包含的日期
someday.hour	返回someday包含的小时
someday.minute	返回someday包含的分钟
someday.second	返回someday包含的秒钟
someday.microsecond	返回someday包含的微秒值



datetime库解析

datetime对象有3个常用的时间格式化方法，如表所示

属性	描述
<code>someday.isoformat()</code>	采用ISO 8601标准显示时间
<code>someday.isoweekday()</code>	根据日期计算星期后返回1-7,对应星期一到星期日
<code>someday.strftime(format)</code>	根据格式化字符串format进行格式显示的方法

isoformat()和isoweekday()方法的使用如下：

```
>>> someday = datetime(2030,11,6,22,33,32,7)
>>> someday.isoformat()
'2030-11-06T22:33:32.000007'
>>> someday.isoweekday()
3
```



datetime库解析

strftime()方法是时间格式化最有效的方法，几乎可以以任何通用格式输出时间

```
>>> someday.strftime("%Y-%m-%d %H:%M:%S")  
'2030-11-06 22:33:32'
```




datetime库解析

格式化字符串	日期/时间	值范围和实例
%Y	年份	0001~9999，例如：1900
%m	月份	01~12，例如：10
%B	月名	January~December，例如：April
%b	月名缩写	Jan~Dec，例如：Apr
%d	日期	01 ~ 31，例如：25
%A	星期	Monday~Sunday，例如：Wednesday
%a	星期缩写	Mon~Sun，例如：Wed
%H	小时（24h制）	00 ~ 23，例如：12
%I	小时（12h制）	01 ~ 12，例如：7
%p	上/下午	AM, PM，例如：PM
%M	分钟	00 ~ 59，例如：26
%S	秒	00 ~ 59，例如：26



datetime库解析

strftime()格式化字符串的数字左侧会自动补零，上述格式也可以与print()的格式化函数一起使用

```
>>>from datetime import datetime
>>>now = datetime.now()
>>>now.strftime("%Y-%m-%d")
'2023-05-19'
>>>now.strftime("%A, %d. %B %Y %I:%M%p")
'Friday, 19. May 2023 12:09PM'
>>>print("今天是{0:%Y}年{0:%m}月{0:%d}日".format(now))
今天是2023年05月19日
```

小练习

5.10 请利用datetime库将当前系统时间转换为字符串

```
>>> import datetime
>>> time=datetime.datetime.now()
>>> time
datetime.datetime(2023, 5, 19, 12, 15, 2, 357666)
>>> print("当前的系统时间是{0:%Y}年{0:%m}月{0:%d}日{0:%H}:{0:%M}".format(datetime.datetime.now()))
当前的系统时间是2023年05月19日12:15
```

5.11 请利用datetime库输出5种不同的日期格式

```
>>> print("当前的UTC系统时间是{0:%Y}年{0:%m}月{0:%d}日{0:%H}:{0:%M}".format(datetime.datetime.utcnow()))
当前的UTC系统时间是2023年05月19日04:18
```

```
>>> 当前的UTC系统时间是2023年05月19日04:18
>>> time=datetime.datetime.now()
>>> time
datetime.datetime(2023, 5, 19, 12, 19, 32, 299217)
>>> time.isoformat()
```

```
>>> time.strftime("%Y-%m-%d %H:%M:%S")
'2023-05-19 12:19:32'
```

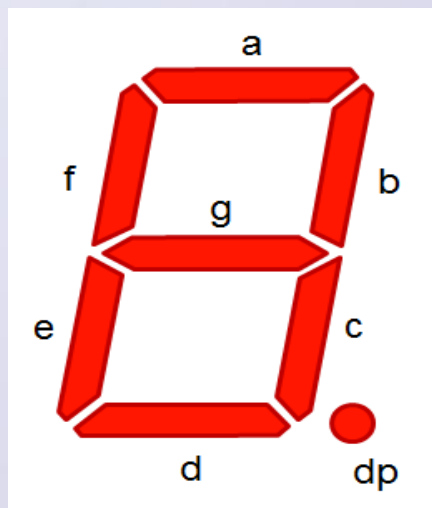
5.12 思考如何利用datetime库对一个程序的运行进行计时



5.10七段数码管绘制

七段数码管绘制

七段数码管（seven-segment indicator）由7段数码管拼接而成，每段有亮或不亮两种情况，改进型的七段数码管还包括一个小数点位置，如图所示。



七段数码管绘制

七段数码管能形成 $2^7=128$ 种不同状态，其中部分状态能够显示易于人们理解的数字或字母含义，因此被广泛使用。下图给出了十六进制中16个字符的七段数码管表示。





七段数码管绘制

每个0到9的数字都有相同的七段数码管样式，因此，可以通过设计函数复用数字的绘制过程。进一步，每个七段数码管包括7个数码管样式，除了数码管位置不同外，绘制风格一致，也可以通过函数复用单个数码段的绘制过程。

七段数码管绘制

Python Turtle Graphics

20230517

File Edit Format Run Options Window Help

```
import turtle, datetime
def drawLine(draw): #绘制单段数码管
    turtle.pendown() if draw else turtle.penup()
    turtle.fd(40)
    turtle.right(90)
def drawDigit(d): #根据数字绘制七段数码管
    drawLine(True) if d in [2, 3, 4, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if d in [0, 1, 3, 4, 5, 6, 7, 8, 9] else drawLine(False)
    drawLine(True) if d in [0, 2, 3, 5, 6, 8] else drawLine(False)
    drawLine(True) if d in [0, 2, 6, 8] else drawLine(False)
    turtle.left(90)
    drawLine(True) if d in [0, 4, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if d in [0, 2, 3, 5, 6, 7, 8, 9] else drawLine(False)
    drawLine(True) if d in [0, 1, 2, 3, 4, 7, 8, 9] else drawLine(False)
    turtle.left(180)
    turtle.penup()
    turtle.fd(20)
def drawDate(time): #获得要输出的数字
    turtle.pencolor("red")
    for i in time:
        drawDigit(int(i)) #注意：通过eval()函数将数字变为整数
def main():
    turtle.setup(800, 350, 200, 200)
    turtle.penup()
    turtle.fd(-300)
    turtle.pensize(5)
    drawDate(datetime.datetime.now().strftime('%Y%m%d'))
    turtle.hideturtle()
main()
```





七段数码管绘制

实例代码7.1

e7.1[DrawSevenSegDisplay.py](#)

```
1  #e7.1DrawSevenSegDisplay.py
2  import turtle, datetime
3  def drawLine(draw):    #绘制单段数码管
4      turtle.pendown() if draw else turtle.penup()
5      turtle.fd(40)
6      turtle.right(90)
7  def drawDigit(d): #根据数字绘制七段数码管
8      drawLine(True) if d in [2,3,4,5,6,8,9] else drawLine(False)
9      drawLine(True) if d in [0,1,3,4,5,6,7,8,9] else drawLine(False)
10     drawLine(True) if d in [0,2,3,5,6,8] else drawLine(False)
11     drawLine(True) if d in [0,2,6,8] else drawLine(False)
12     turtle.left(90)
13     drawLine(True) if d in [0,4,5,6,8,9] else drawLine(False)
14     drawLine(True) if d in [0,2,3,5,6,7,8,9] else drawLine(False)
15     drawLine(True) if d in [0,1,2,3,4,7,8,9] else drawLine(False)
```



七段数码管绘制

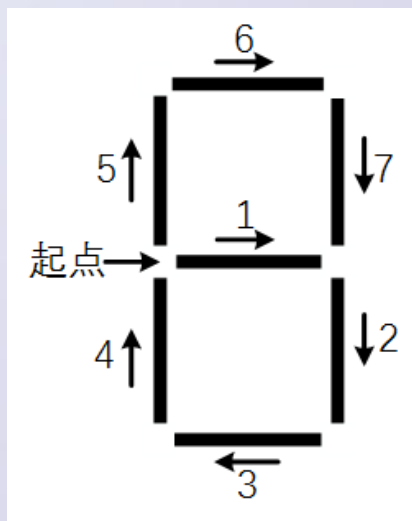
实例代码7.1

e7.1 [DrawSevenSegDisplay.py](#)

```
16         turtle.left(180)
17         turtle.penup()
18         turtle.fd(20)
19     def drawDate(time): #获得要输出的数字
20         turtle.pencolor("red")
21         for i in date:
22             drawDigit(int(i)) #注意：也可通过eval()函数将数字变为整数
23     def main():
24         turtle.setup(800, 350, 200, 200)
25         turtle.penup()
26         turtle.fd(-300)
27         turtle.pensize(5)
28         drawDate(datetime.datetime.now().strftime('%Y%m%d'))
29         turtle.hideturtle()
main()
```

七段数码管绘制

实例代码定义了drawDigit()函数，该函数根据输入的数字d绘制七段数码管，结合七段数码管结构，每个数码管的绘制采用图所示顺序。





七段数码管绘制

绘制起点在数码管中部左侧，无论每段数码管是否被绘制出来，turtle画笔都按顺序“画完”所有7个数码管。对于给定数字d，哪个数码段被绘制出来采用if...else...语句判断。

8

```
drawLine(True) if d in [2,3,4,5,6,8,9] else drawLine(False)
```



七段数码管绘制



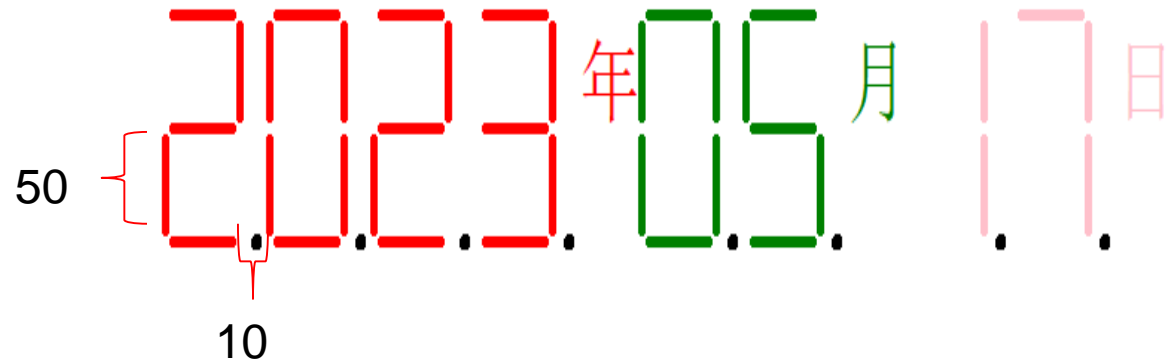
```
1 #e7.2DrawSevenSegDisplay.py
2 import turtle, datetime
3 def drawGap(): #绘制数码管间隔
4     turtle.penup()
5     turtle.fd(5)
6 def drawLine(draw): #绘制单段数码管
7     drawGap()
8     turtle.pendown() if draw else turtle.penup()
9     turtle.fd(40)
10    drawGap()
11    turtle.right(90)
12 def drawDigit(d): #根据数字绘制七段数码管
13     drawLine(True) if d in [2,3,4,5,6,8,9] else drawLine(False)
14     drawLine(True) if d in [0,1,3,4,5,6,7,8,9] else drawLine(False)
15     drawLine(True) if d in [0,2,3,5,6,8] else drawLine(False)
16     drawLine(True) if d in [0,2,6,8] else drawLine(False)
17     turtle.left(90)
18     drawLine(True) if d in [0,4,5,6,8,9] else drawLine(False)
19     drawLine(True) if d in [0,2,3,5,6,7,8,9] else drawLine(False)
20     drawLine(True) if d in [0,1,2,3,4,7,8,9] else drawLine(False)
21     turtle.left(180)
22     turtle.penup()
23     turtle.fd(20)
24
```

```
25 def drawDate(date):
26     turtle.pencolor("red")
27     for i in date:
28         if i == '-':
29             turtle.write('年', font=("Arial", 18, "normal"))
30             turtle.pencolor("green")
31             turtle.fd(40)
32         elif i == '=':
33             turtle.write('月', font=("Arial", 18, "normal"))
34             turtle.pencolor("blue")
35             turtle.fd(40)
36         elif i == '+':
37             turtle.write('日', font=("Arial", 18, "normal"))
38         else:
39             drawDigit(eval(i))
40 def main():
41     turtle.setup(800, 350, 200, 200)
42     turtle.penup()
43     turtle.fd(-350)
44     turtle.pensize(5)
45     drawDate(datetime.datetime.now().strftime('%Y-%m=%d+'))
46     turtle.hideturtle()
main()
```

七段数码管绘制-增加小数点

```
def drawdigit(digit):
    drawline(True) if digit in [2, 3, 4, 5, 6, 8, 9] else drawline(False)
    drawline(True) if digit in [0, 1, 3, 4, 5, 6, 7, 8, 9] else drawline(False)
    drawline(True) if digit in [0, 2, 3, 5, 6, 8, 9] else drawline(False)
    drawline(True) if digit in [0, 2, 6, 8] else drawline(False)
    turtle.left(90)
    drawline(True) if digit in [0, 4, 5, 6, 8, 9] else drawline(False)
    drawline(True) if digit in [0, 2, 3, 5, 6, 8, 9] else drawline(False)
    drawline(True) if digit in [0, 1, 2, 3, 4, 7, 8, 9] else drawline(False)
    turtle.left(180)
    turtle.penup()
    turtle.fd(20)
    z = 1
def drawpoint():
    #绘制小数点
    turtle.penup()
    turtle.bk(10)
    turtle.seth(-90)
    turtle.fd(50)
    turtle.pendown()
    turtle.begin_fill()
    turtle.color("black")
    turtle.circle(1, 360)
    turtle.end_fill()
    turtle.penup()
    turtle.bk(50)
    turtle.seth(0)
    turtle.fd(10)
    if z <= 4:
        turtle.pencolor("red")
    elif 4 < z <= 6:
        turtle.pencolor("green")
    else:
        turtle.pencolor("pink")
def drawdate(date):
    turtle.pencolor("red")
    for i in date:
        if i == '.':
            turtle.write("年", font=("Arial", 30, "normal"))
            turtle.pencolor("green")
            turtle.fd(40)
            #print(z)
        elif i == '=':
            turtle.write("月", font=("Arial", 30, "normal"))
            turtle.pencolor("pink")
            turtle.fd(40)
            #print(z)
        elif i == '+':
            turtle.write("日", font=("Arial", 30, "normal"))
            #print(z)
        else:
            drawdigit(eval(i))
            drawpoint()
            global z
            z += 1
def main():
    turtle.setup(800, 350, 200, 200)
    turtle.penup()
    turtle.fd(-300)
    turtle.pensize(5)
    drawdate(time.strftime("%Y-%m=%d+", time.gmtime()))
    turtle.hideturtle()
    turtle.done()
main()
```

```
import datetime
print(datetime.datetime.now().strftime('%Y-%m=%d+'))
2023-05=17+
```



七段数码管绘制-倒计时功能



```
3
<class 'str'>
2
<class 'str'>
1
<class 'str'>
0
-----倒计时结束-----
```

===== RESTART: E:/liushuo/qiduanshumaguandaojishi.py ==

请设置倒计时时间: 10
请设置绘制速度大小: 4

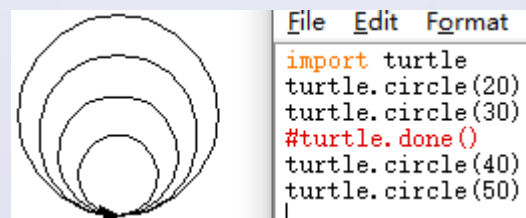
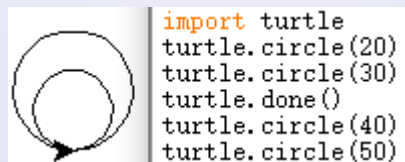
```
<class 'str'>
1
0
<class 'str'>
9
<class 'str'>
8
<class 'str'>
7
<class 'str'>
6
<class 'str'>
5
<class 'str'>
4
<class 'str'>
3
<class 'str'>
2
```

File Edit Format Run Options Window Help

```
#QiDuanShuMaGuan.py
import turtle
def drawGap(): #设置每条线之间的间隔
    turtle.penup() #画笔抬起
    turtle.fd(5)
def drawLine(draw): #绘制单段数码管
    drawGap()
    turtle.pendown() if draw else turtle.penup() #根据参数draw判断画笔是放下还是抬起
    turtle.fd(40)
    drawGap()
    turtle.right(90)
def drawDights(dight): #根据数字绘制七段数码管
    drawLine(True) if dight in [2, 3, 4, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if dight in [0, 1, 3, 4, 5, 6, 7, 8, 9] else drawLine(False)
    drawLine(True) if dight in [0, 2, 3, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if dight in [0, 2, 6, 8] else drawLine(False)
    turtle.left(90)
    drawLine(True) if dight in [0, 4, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if dight in [0, 2, 3, 5, 6, 7, 8, 9] else drawLine(False)
    drawLine(True) if dight in [0, 1, 2, 3, 4, 7, 8, 9] else drawLine(False)
    turtle.left(180)
    turtle.penup()
    turtle.fd(15) #每个绘制的七段数码管之间的距离
def drawDate(time):
    turtle.pencolor("red")
    for i in reversed(range(time+1)):
        num = str(i)
        print(type(num))
        for n in num:
            print(n)
            drawDights(int(n))
        turtle.clear() #清空已经绘制的七段数码管，为下次绘制做好准备
        s = len(num)
        turtle.fd(-65*s) #回退到起始位置
def main():
    turtle.setup(500, 350, 20, 20) #设置窗口大小以及相对屏幕的位置
    turtle.hideturtle() #隐藏画笔
    temp = input("请设置倒计时时间:")
    sp = input("请设置绘制速度大小:")
    turtle.speed(eval(sp)) #设置绘制速度
    turtle.penup()
    turtle.fd(-200) #画笔默认在窗口中心，此处使画笔回退200个像素点
    turtle.pensize(5)
    drawDate(eval(temp))
    print("倒计时结束".center(20, "-"))
    turtle.done() #程序运行后不会立即退出
    turtle.pendown()
    turtle.fd(40)
main()
```

七段数码管绘制-倒计时功能

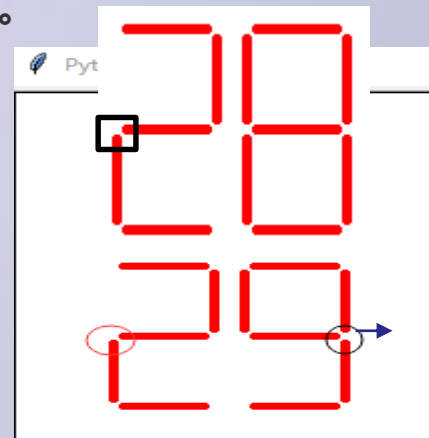
```
>>> i=[1,2,3]
>>> j=reversed(i)
>>> list(j)
[3, 2, 1]
>>> type(j)
<class 'list_reverseiterator'>
>>> j
<list_reverseiterator object at 0x000001F9A8DF3D00>
...|
```



`turtle.clear()`
参数: None
Returns: Nothing

```
def drawGap(): #设置每条线之间的间隔
    turtle.penup() #画笔抬起
    turtle.fd(5)
def drawLine(draw): #绘制单段数码管
    drawGap()
    turtle.pendown() if draw else turtle.penup()
    turtle.fd(40)
    drawGap()
    turtle.right(90)
def drawDigits(digit): #根据数字绘制七段数码管
    drawLine(True) if digit in [2, 3, 4, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if digit in [0, 1, 3, 4, 5, 6, 7, 8, 9] else drawLine(False)
    drawLine(True) if digit in [0, 2, 3, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if digit in [0, 2, 6, 8] else drawLine(False)
    turtle.left(90)
    drawLine(True) if digit in [0, 4, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if digit in [0, 2, 3, 5, 6, 7, 8, 9] else drawLine(False)
    drawLine(True) if digit in [0, 1, 2, 3, 4, 7, 8, 9] else drawLine(False)
    turtle.left(180)
    turtle.penup()
    turtle.fd(15) #每个绘制的七段数码管之间的距离
```

- `turtle.speed()`
- `turtle` 的速度介于0到10之间。
- 若输入的数字大于10或小于0.5, 则速度设置为0。
- 速度字符串通过以下方式映射到速度值:
 - 'fastest' : 0
 - 'fast' : 10
 - 'normal' : 6
 - 'slow' : 3
 - 'slowest' : 1
- 从1到10的速度会强制加快线条绘制和 `turtle` 转向的动画速度。



$\text{min} = (a < b) ? a : b;$

```
def drawDigits(dight):    #根据数字绘制七段数码管
    drawLine(True) if dight in [2, 3, 4, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if dight in [0, 1, 3, 4, 5, 6, 7, 8, 9] else drawLine(False)
    drawLine(True) if dight in [0, 2, 3, 5, 6, 8, 9] else drawLine(False)
    drawLine(True) if dight in [0, 2, 6, 8] else drawLine(False)
    turtle.left(90)
```

```
#e7.2DrawSevenSegDisplay.py
import turtle, datetime
def drawGap(): #绘制数码管间隔
    turtle.penup()
    turtle.fd(5)
def drawLine(draw):    #绘制单段数码管
    drawGap()
    turtle.pendown() if draw else turtle.penup()
    turtle.fd(40)
    drawGap()
    turtle.right(90)
def drawDigit(d): #根据数字绘制七段数码管
    drawLine(True) if d in [2,3,4,5,6,8,9] else drawLine(False)
    drawLine(True) if d in [0,1,3,4,5,6,7,8,9] else drawLine(False)
    drawLine(True) if d in [0,2,3,5,6,8,9] else drawLine(False)
    drawLine(True) if d in [0,2,6,8] else drawLine(False)
    turtle.left(90)
    drawLine(True) if d in [0,4,5,6,8,9] else drawLine(False)
    drawLine(True) if d in [0,2,3,5,6,7,8,9] else drawLine(False)
    drawLine(True) if d in [0,1,2,3,4,7,8,9] else drawLine(False)
    turtle.left(180)
    turtle.penup()
    turtle.fd(20)
```

```
def drawDate(date):
    turtle.pencolor("red")
    for i in date:
        if i == '-':
            turtle.write('年',font=("Arial", 18, "normal"))
            turtle.pencolor("green")
            turtle.fd(40)
        elif i == '=':
            turtle.write('月',font=("Arial", 18, "normal"))
            turtle.pencolor("blue")
            turtle.fd(40)
        elif i == '+':
            turtle.write('日',font=("Arial", 18, "normal"))
        else:
            drawDigit(eval(i))
def main():
    turtle.setup(800, 350, 200, 200)
    turtle.penup()
    turtle.fd(-350)
    turtle.pensize(5)
    drawDate(datetime.datetime.now().strftime('%Y-%m=%d+'))
    turtle.hideturtle()
main()
```

紧耦合-互相之间调用比较紧密

松耦合



5.11 代码的复用和模块化设计

代码复用

把代码当成资源进行抽象

- 代码资源化：程序代码是一种用来表达计算的“资源”
- 代码抽象化：使用函数等方法对代码赋予更高级别的定义
- 代码复用：同一份代码在需要时可以被重复使用

代码复用

函数 和 对象 是代码复用的两种主要形式

函数：将代码命名在代码层面建立了初步抽象

对象：属性和方法

$\langle a \rangle . \langle b \rangle$ 和 $\langle a \rangle . \langle b \rangle ()$

在函数之上再次组织进行抽象



抽象级别

模块化设计

分而治之

- 通过函数或对象封装将程序划分为模块及模块间的表达
- 具体包括：主程序、子程序和子程序间关系
- 分而治之：一种分而治之、分层抽象、体系化的设计思想



代码的复用和模块化设计

函数是程序的一种基本抽象方式，它将一系列代码组织起来通过命名供其他程序使用。函数封装的直接好处是代码复用，任何其他代码只要输入参数即可调用函数，从而避免相同功能代码在被调用处重复编写。代码复用产生了另一个好处，当更新函数功能时，所有被调用处的功能都被更新。



代码的复用和模块化设计

当程序的长度在百行以上，如果不划分模块就算是最好的程序员也很难理解程序含义程序的可读性就已经很糟糕了。解决这一问题的最好方法是将一个程序分割成短小的程序段，每一段程序完成一个小的功能。无论面向过程和面向对象编程，对程序合理划分功能模块并基于模块设计程序是一种常用方法，被称为“模块化设计”。



代码的复用和模块化设计

模块化设计一般有两个基本要求：

- 紧耦合：尽可能合理划分功能块，功能块内部耦合紧密；
- 松耦合：模块间关系尽可能简单，功能块之间耦合度低。

使用函数只是模块化设计的必要非充分条件，根据计算需求合理划分函数十分重要。一般来说，完成特定功能或被经常复用的一组语句应该采用函数来封装，并尽可能减少函数间参数和返回值的数量。



5.12函数的递归

递归的定义

函数作为一种代码封装，可以被其他程序调用，当然，也可以被函数内部代码调用。这种函数定义中调用函数自身的方式称为递归。就像一个人站在装满镜子的房间中，看到的影像就是递归的结果。递归在数学和计算机应用上非常强大，能够非常简洁的解决重要问题。



山鲁佐德：从前有一个国王，特别暴虐，每天都要找一个姑娘成亲，并且很快就会把她杀掉，后来有一个姑娘自愿嫁给国王，每天给他讲故事，

第一天的故事是这样的：从前有一个国王，特别暴虐，每天都要找一个姑娘成亲，并且很快就会把她杀掉，后来有一个姑娘自愿嫁给国王，每天给他讲故事，

第一天的故事是这样的：从前有一个国王，特别暴虐，每天都要找一个姑娘成亲，并且很快就会把她杀掉，后来有一个姑娘自愿嫁给国王，每天给他讲故事，第一天的故事是这样的....



递归的定义

数学上有个经典的递归例子叫阶乘，阶乘通常定义为：

$$n! = n(n-1)(n-2)\dots(1)$$

这个关系给出了另一种方式表达阶乘的方式：

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & otherwise \end{cases}$$



递归的定义

阶乘的例子揭示了递归的2个关键特征：

- (1) 存在一个或多个基例，基例不需要再次递归，它是确定的表达式；
- (2) 所有递归链要以一个或多个基例结尾。

递归-由未知推已知，不断向前 函数调用自身，有时间和空间的消耗 有去有回

循环-由已知推出未知，不断向后，有去无回

斐波那契数列

$$F(n) = F(n-1) + F(n-2)$$

— 函数 + 分支结构

```
def f(n):
```

```
    if n == 1 or n == 2 :
```

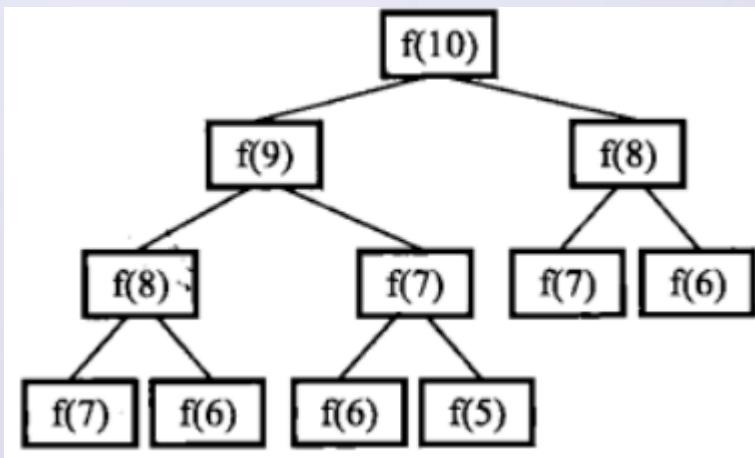
```
        return 1
```

```
    else :
```

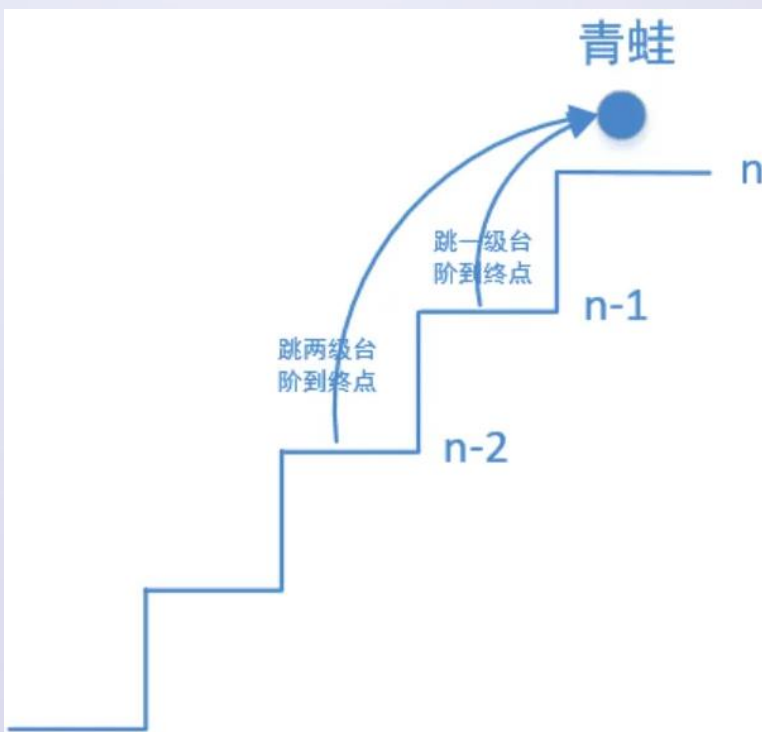
```
        return f(n-1) + f(n-2)
```

— 递归链条

— 递归基例



- 一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法。



设青蛙跳上n级台阶有 $f(n)$ 总方法， $f(n)$ 总方法分为两种：

1. 最后一次跳了一级台阶，这类方法共有 $f(n-1)$ 种；
2. 最后一次跳了两级台阶，这类方法共有 $f(n-2)$ 种。

```
>>> def taijie(n):  
...     if n==1 or n==2:  
...         return n  
...     return taijie(n-1) + taijie(n-2)  
...  
>>> taijie(10)  
89
```




递归的使用方法

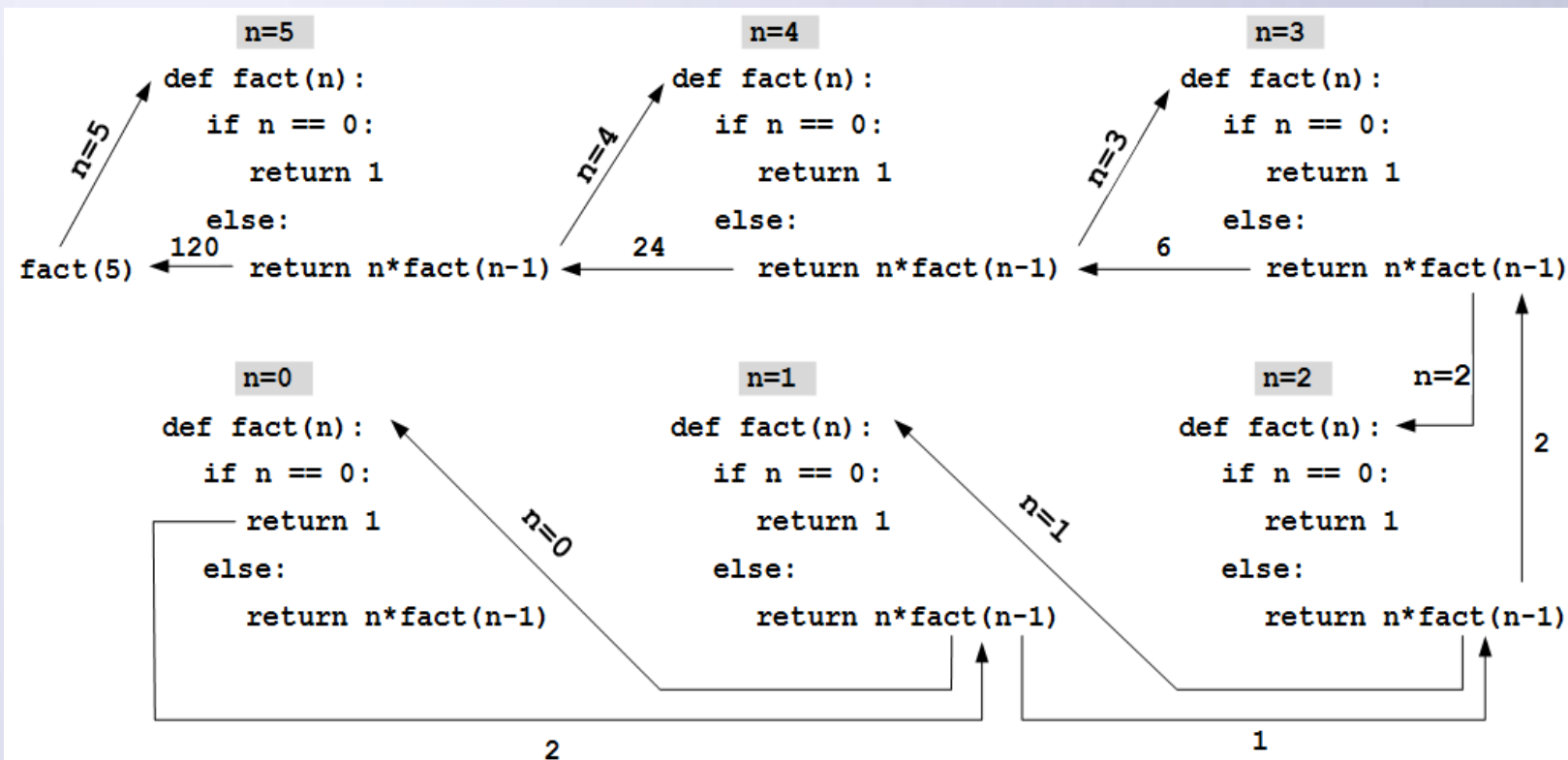
微实例5.21：阶乘的计算。

根据用户输入的整数 n ，计算并输出 n 的阶乘值。

微实例5.21 m5.1CalFactorial.py

```
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6 num = eval(input("请输入一个整数: "))
7 print(fact(abs(int(num))))
```

递归的使用方法





递归的使用方法

微实例5.32：字符串反转。

对于用户输入的字符串s，输出反转后的字符串。

解决这个问题基本思想是把字符串看作一个递归对象。

```
1 def reverse(s):  
2     return reverse(s[1:]) + s[0]
```

递归的使用方法

观察这个函数的工作过程。s[0]是首字符，s[1:]是剩余字符串，将它们反向连接，可以得到反转字符串。执行这个程序，结果如下

```
>>>def reverse(s):  
    return reverse(s[1:]) + s[0]  
>>>reverse("ABC")  
...
```

```
>>> def reverse(s):  
...     return reverse(s[1:]) + s[0]  
...  
>>> reverse("ABC")  
Traceback (most recent call last):  
  File "<pyshell#9>", line 1, in <module>  
    reverse("ABC")  
  File "<pyshell#8>", line 2, in reverse  
    return reverse(s[1:]) + s[0]  
  File "<pyshell#8>", line 2, in reverse  
    return reverse(s[1:]) + s[0]  
  File "<pyshell#8>", line 2, in reverse  
    return reverse(s[1:]) + s[0]  
  [Previous line repeated 1022 more times]  
RecursionError: maximum recursion depth exceeded
```

```
>>> def reverse(s):  
...     if len(s)>=1:  
...         return reverse(s[1:]) + s[0]  
...     return s
```

```
>>> reverse("ABC")  
'CBA'
```

递归函数需要注意的问题

- (1) 必须设置终止条件
 - 缺少终止条件的递归函数，将会导致无限递归函数调用，其最终结果是系统会耗尽内存
- (2) 必须保证收敛
 - 否则，也会导致无限递归函数调用
- (3) 必须保证内存和运算消耗控制在一定范围

递归函数的应用：最大公约数

- **【例8.28】** 使用递归函数计算最大公约数 (`gcd.py`)

(1) 终止条件: `gcd(p, q) = p` #当 `q == 0` 时

(2) 递归步骤: `gcd(q, p % q)` #当 `q > 0` 时

```
import sys
def gcd(p, q): #使用递归函数计算p和q的最大公约数
    if q == 0: return p #如果q=0, 返回p
    return gcd(q, p % q) #否则, 递归调用gcd(q, p % q)
#测试代码
p = int(sys.argv[1]) #p=命令行第一个参数
q = int(sys.argv[2]) #q=命令行第二个参数
print(gcd(p, q)) #计算并输出p和q的最大公约数
```

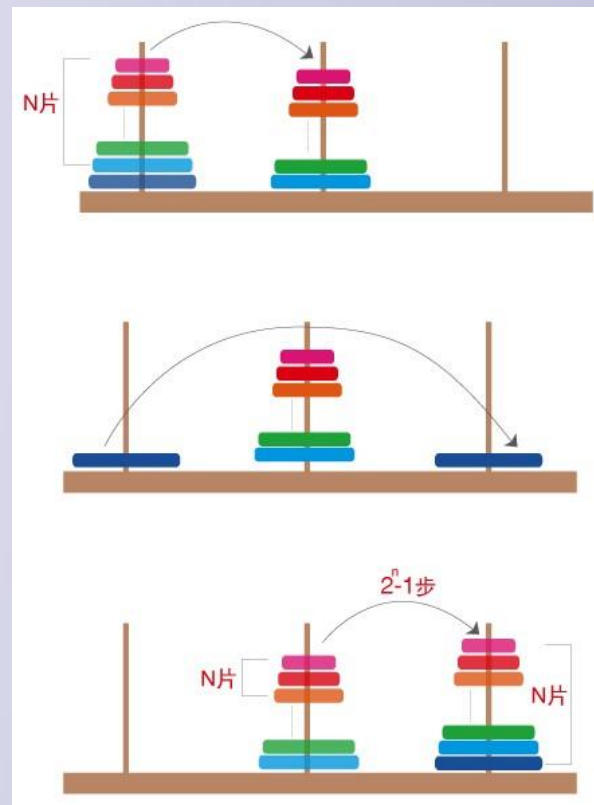
```
e:\liushuo>python 最大公约数.py 12 3
3
e:\liushuo>python 最大公约数.py 69 75
3
```

递归函数的应用：汉诺塔

- 大梵天创造世界的时候，在世界中心贝拿勒斯的圣庙里做了三根金剛石柱，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。称之为汉诺塔
- 大梵天命令婆罗门把圆盘从一根柱子上按大小顺序重新摆放在另一根柱子上。并且规定，在三根柱子之间一次只能移动一个圆盘，且小圆盘上不能放置大圆盘。这个游戏称之为汉诺塔益智游戏
- 汉诺塔益智游戏问题很容易使用递归函数实现。假设柱子编号为a、b、c，定义函数hanoi(n, a, b, c)表示把n个圆盘从柱子a移到柱子c（可以经由柱子b），则有：
 - （1）终止条件。当 $n==1$ 时，hanoi(n, a, b, c)为终止条件。即如果柱子a上只有一个圆盘，则可以直接将其移动到柱子c上
 - （2）递归步骤。hanoi(n, a, b, c)可以分解为三个步骤：hanoi(n-1, a, c, b)、hanoi(1, a, b, c)和hanoi(n-1, b, a, c)。如果柱子a上有n个圆盘，可以看成柱子a上有一个圆盘（底盘）和(n-1)个圆盘，首先需要把柱子a上面的(n-1)个圆盘移动到柱子b，即调用hanoi(n-1, a, c, b)；然后，把柱子a上的最后一个圆盘移动到柱子c，即调用hanoi(1, a, b, c)；再将柱子b上的(n-1)个圆盘移动到柱子c，即调用hanoi(n-1, b, a, c)
 - 每次递归，n严格递减，故逐渐收敛于1



汉诺塔



算法 Hanoi (A, C, n) // n个盘子A到C

1. if $n=1$ then move (A, C) // 1个盘子A到C
2. else Hanoi (A, B, $n-1$)
3. move (A, C)
4. Hanoi (B, C, $n-1$)

设 n 个盘子的移动次数为 $T(n)$

$$T(n) = 2 T(n-1) + 1,$$

$$T(1) = 1,$$

$$T(n) = 2T(n-1)+1=2(2T(n-2) + 1)+1=2(2(2T(n-3)+1)+1)+1...=2^{n-1}$$

$$4T(n-2)+3$$

$$8T(n-3)+7$$

$$2^{n-1}T(n-(n-1))+(2^{n-1}-1)$$

- 【例8.29】使用递归函数实现汉诺塔问题

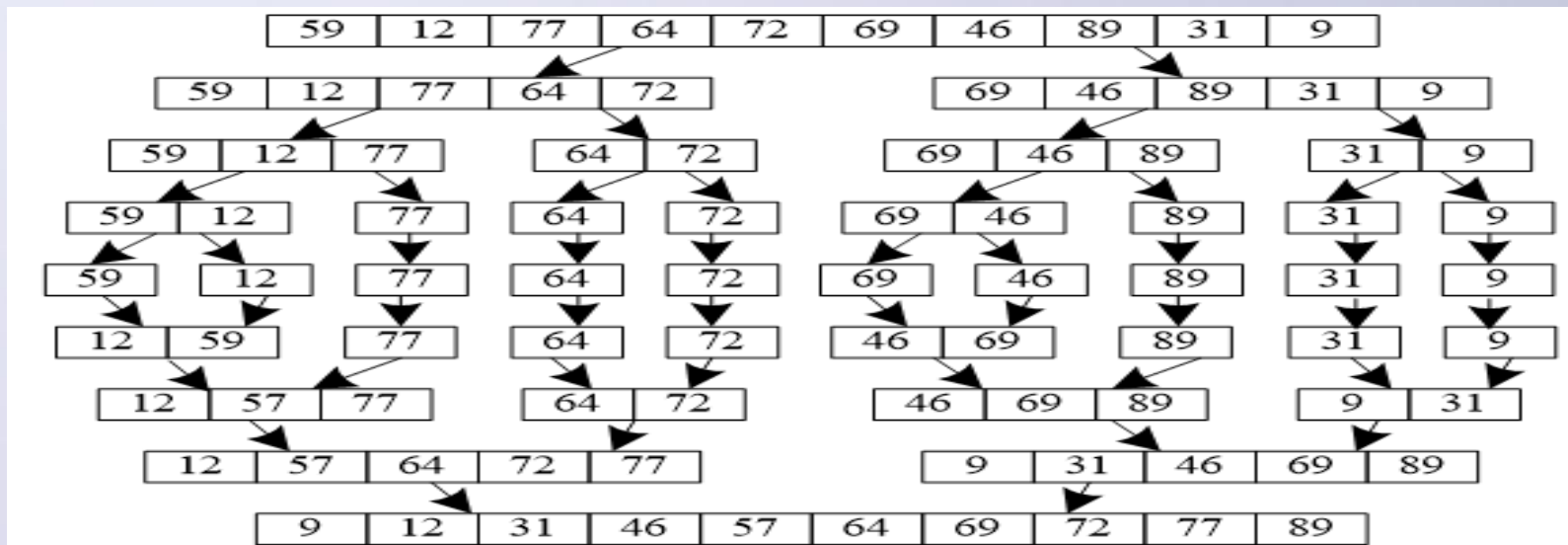
```
#将n个从小到大依次排列的圆盘从柱子a移动到柱子c上，柱子b作为中间缓冲
def hanoi(n, a, b, c):
    if n==1: print(a, '->', c) #只有一个圆盘，直接将圆盘从柱子a移动到柱子c上
    else:
        hanoi(n-1, a, c, b) #先将n-1个圆盘从柱子a移动到柱子b上（采用递归方式）
        hanoi(1, a, b, c)   #然后将最大的圆盘从柱子a移动到柱子c上
        hanoi(n-1, b, a, c) #再将n-1个圆盘从柱子b移动到柱子c上（采用递归方式）
#测试代码
hanoi(4, 'A', 'B', 'C')
```

程序运行结果如下

```
A->B
A->C
B->C
A->B
C->A
C->B
A->B
A->C
B->C
B->A
C->A
B->C
A->B
A->C
B->C
```

归并排序法

- 归并排序法基于分而治之（Divide and Conquer）的思想。算法的操作步骤如下。
- （1）将包含N个元素的列表分成两个含N/2元素的子列表。
- （2）对两个子列表递归调用归并排序（最后可以将整个列表分解成N个子列表）。
- （2）合并两个已排序好的子列表。



【例】归并排序算法的实现 (mergeSort.py)

素


```
def merge(left, right): #合并两个列表
    merged = []
    i, j = 0, 0 #i和j分别作为left和right的下标
    left_len, right_len = len(left), len(right) #获取左右子列表长度
    while i < left_len and j < right_len: #循环归并左右子列表元素
        if left[i] <= right[j]:
            merged.append(left[i]) #归并左子列表元素
            i += 1
        else:
            merged.append(right[j]) #归并右子列表元素
            j += 1
    merged.extend(left[i:]) #归并左子列表剩余元素
    merged.extend(right[j:]) #归并右子列表剩余元素
    #print(left, right, merged) #跟踪调试
    return merged #返回归并好的列表
```

```
def mergeSort(a): #归并排序
    if len(a) <= 1: #空或者只有1个元素，直接返回列表
        return a
    mid = len(a) // 2 #列表中间位置
    left = mergeSort(a[:mid]) #归并排序左子列表
    right = mergeSort(a[mid:]) #归并排序右子列表
    return merge(left, right) #合并排好序的左右两个子列表


def main():
    a = [59, 12, 77, 64, 72, 69, 46, 89, 31, 9]
    a1 = mergeSort(a)
    print(a1)
if __name__ == '__main__': main()
```

程序运行结果如下。 ↵

[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]



5.13科赫曲线绘制



科赫曲线绘制

自然界有很多图形很规则，符合一定的数学规律，例如，蜜蜂蜂窝是天然的等边六角形等。科赫(Koch)曲线在众多经典数学曲线中非常著名，由瑞典数学家冯·科赫(H·V·Koch)于1904年提出，由于其形状类似雪花，也被称为雪花曲线。



科赫曲线绘制

科赫曲线的基本概念和绘制方法如下：

正整数 n 代表科赫曲线的阶数，表示生成科赫曲线过程的操作次数。科赫曲线初始化阶数为0，表示一个长度为 L 的直线。对于直线 L ，将其等分为三段，中间一段用边长为 $L/3$ 的等边三角形的两个边替代，得到1阶科赫曲线，它包含四条线段。进一步对每条线段重复同样的操作后得到2阶科赫曲线。继续重复同样的操作 n 次可以得到 n 阶科赫曲线。



科赫曲线绘制

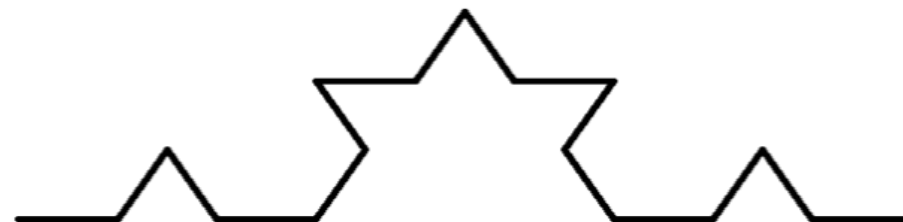
0阶科赫曲线



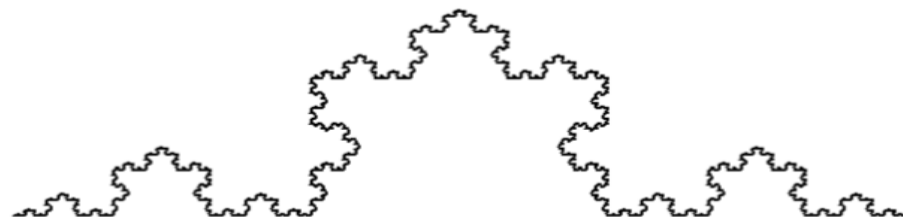
1阶科赫曲线



2阶科赫曲线



5阶科赫曲线



科赫雪花小包裹(上)

科赫曲线的绘制

—递归思想：函数+分支

—递归链条：线段的组合

—递归基例：初始线段

```
#KochDrawV1.py
```

```
import turtle
```

```
def koch(size, n):
```

```
    if n == 0:
```

```
        turtle.fd(size)
```

```
    else:
```

```
        for angle in [0, 60, -120, 60]:
```

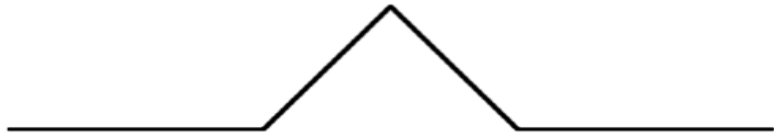
```
            turtle.left(angle)
```

```
            koch(size/3, n-1)
```

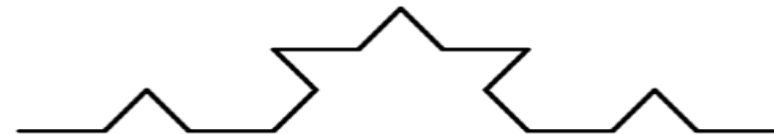
0阶科赫曲线



1阶科赫曲线



2阶科赫曲线




```
#KochDrawV1.py
```

```
import turtle
```

```
def koch(size, n):
```

```
    if n == 0:
```

```
        turtle.fd(size)
```

```
    else:
```

```
        for angle in [0, 60, -120, 60]:
```

```
            turtle.left(angle)
```

```
            koch(size/3, n-1)
```

```
def main():
```

```
    turtle.setup(800,400)
```

```
    turtle.penup()
```

```
    turtle.goto(-300, -50)# 绝对坐标
```

```
    turtle.pendown()
```

```
    turtle.pensize(2)
```

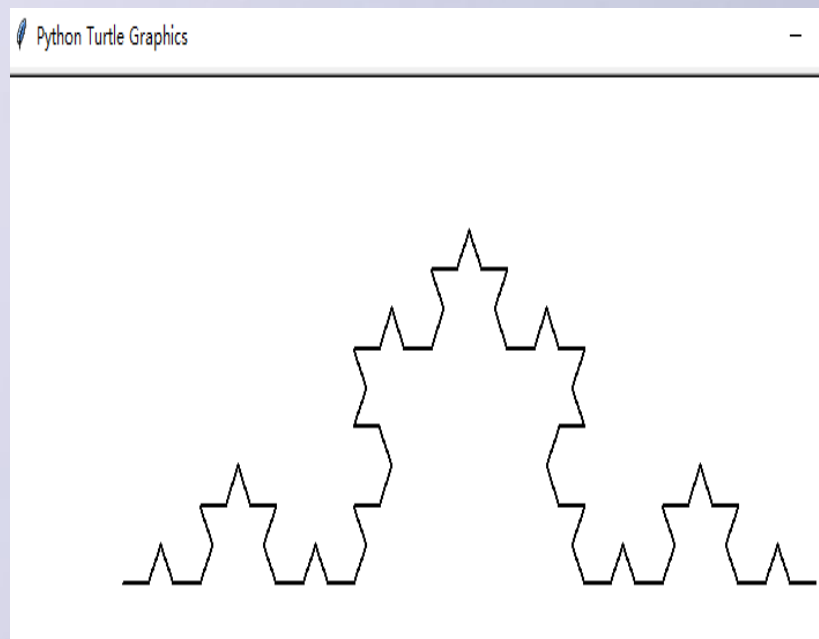
```
    koch(600, 3)      # 3阶科赫曲线, 阶数
```

```
    turtle.hideturtle()
```

```
main()
```

科赫雪花小包裹(上)

科赫曲线的绘制



```
#KochDrawV2.py
```

```
import turtle,time
```

```
def koch(size, n):
```

```
...(略)
```

```
def main():
```

```
    turtle.setup(600,600)
```

```
    turtle.penup()
```

```
    turtle.goto(-200, 100)
```

```
    turtle.pendown()
```

```
    turtle.pensize(2)
```

```
    level = 3 # 3阶科赫雪花, 阶数
```

```
    koch(400, level)
```

```
    turtle.right(120)
```

```
    time.sleep(5)
```

```
    koch(400, level)
```

```
    turtle.right(120)
```

```
    time.sleep(5)
```

```
    koch(400, level)
```

```
    turtle.hideturtle()
```

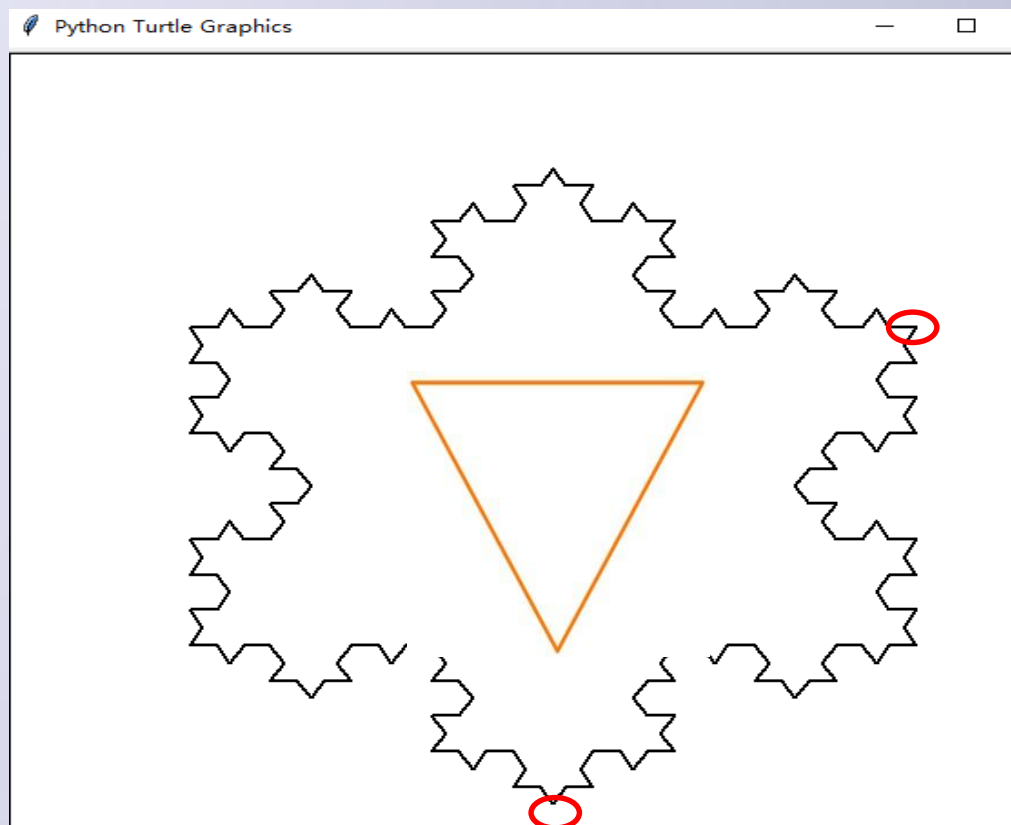
```
main()
```

科赫雪花小包裹(上)

科赫曲线的绘制



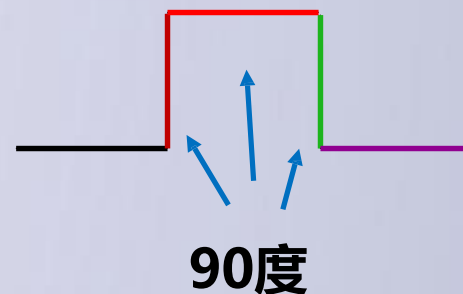
科赫雪花的绘制



举一反三

绘制条件的扩展

- 修改分形几何绘制阶数
- 修改科赫曲线的基本定义及旋转角度
- 修改绘制科赫雪花的基础框架图形



举一反三

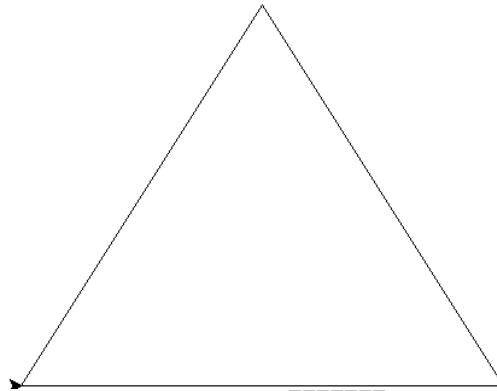
分形几何千千万

—康托尔集、谢尔宾斯基三角形、门格海绵...

—龙形曲线、空间填充曲线、科赫曲线
...

—函数递归的深入应用...

```
import turtle
def draw_angle(a,b,c):
    ax,ay = a
    bx,by = b
    cx,cy = c
    turtle.penup()
    turtle.goto(ax,ay)
    turtle.pendown()
    turtle.goto(bx,by)
    turtle.goto(cx,cy)
    turtle.goto(ax,ay)
    turtle.penup()
```



```
def get_midpoint(a, b):
    ax, ay = a
    bx, by = b
    return (ax + bx) / 2, (ay + by) / 2
```

=====
 =====
 =====
 (-100.0, 50.0)

```
def draw_sierpinski(triangle, depth):
```

```
    a, b, c = triangle
    draw_angle(a, b, c)
```

```
    if depth == 0:
```

```
        return
```

```
    else:
```

```
        d = get_midpoint(a, b)
```

```
        e = get_midpoint(b, c)
```

```
        f = get_midpoint(c, a)
```

```
        draw_sierpinski([a, d, f], depth-1)
```

```
        draw_sierpinski([d, b, e], depth-1)
```

```
        draw_sierpinski([f, e, c], depth-1)
```

```
if __name__ == '__main__':
```

```
    triangle = [[-200, -100], [0, 200], [200, -100]]
```

```
    draw_sierpinski(triangle, 4)
```

```
    turtle.done()
```

```
def get_midpoint(a, b):
    ax, ay = a
    bx, by = b
    return (ax + bx) / 2, (ay + by) / 2
d = get_midpoint([-200, -100], [0, 200])
print (d)
```

