第5章 函数与模块

CHAPTER 5

随着程序功能的提升，程序开发的难度和程序的复杂度越来越高，为了提高代码的复用性、更好地组织代码结构与逻辑，人们提出了函数这一概念。本章首先对函数的相关知识进行讲解，然后介绍模块的知识内容。

通过本章的学习，大家能够掌握函数的定义和调用方法、理解函数中参数的作用、能够正确地使用 Python 中的内置函数；同时理解 Python 模块概念、掌握模块的语法及正则表达式模块的使用。

25min

5.1 函数的定义和调用

在程序开发中，如果有若干代码的执行逻辑完全相同，就可以考虑将这些代码封装成一个函数，这样不仅可以提高代码的重用性，而且条理会更加清晰，可靠性更高。

Python 提供了很多种内置函数，如 print()、input()、int()、float()等。除此之外，还可以自己创建函数，也就是自定义函数。先来看下面这段代码。

```
print("  *  ")          #字符串中有 5 个字符，"*"号前后各有两个空格
print(" *** ")          #字符串中有 5 个字符，"***"号前后各有一个空格
print("*****")
print(" *** ")
print("  *  ")
```

在上述代码中，使用多个 print()函数输出了一个菱形。如果需要在一个程序的不同位置输出这张图形，则每次都使用 5 行 print()函数输出的做法是不可取的。为了提高编写的效率和代码的重用性，可以把具有独立功能的代码组织成一个小模块，这就是函数。

函数的使用分为定义和调用两部分，下面就重点讲解函数的定义和调用。

5.1.1 函数的定义

函数是组织好的可重复使用的用来实现单一或相关联功能的代码段。函数是带有函数名的一系列语句，在使用之前（或调用前），需要先通过 def 关键字定义，其语法格式如下：

```
def 函数名([参数列表]):
[ """文档字符串"""]
函数体
[return [语句]]
```

函数定义语法格式的规则说明如下。

(1) def关键字：因为def关键字是定义函数的开始标志，所以函数代码块以def开头。后面是函数名和圆括号。

(2) 函数名：函数的唯一标识，其命名规则和变量的命名规则是一样的，即只能由字母、数字和下画线组成，但是不能以数字开头，并且不能和关键字重名。

(3) 函数的参数：必须放在圆括号中，负责接收传入函数中的数据，既可以包含一个或多个参数，也可以为空。

(4) 冒号：函数体的开始标志。函数体代码在冒号的下一行开始编写，并且前面需要缩进。

(5) 文档字符串：由一对三双引号(或三单引号)包裹，用于说明函数的功能，可以省略。

(6) 函数体：实现函数功能的具体代码(可以是一行或多行程序代码)。

(7) return语句：将函数的处理结果返回给调用方，是函数的结束标志。若函数没有返回值，则可以省略return语句，相当于返回None。

【思考与练习5-1】

(1) 定义不带任何参数的函数hello，其功能是输出显示菱形。

(2) 定义带参数的函数add，其功能是计算两个输入整数的和。

5.1.2　函数的调用

1. 函数的调用方法

定义函数后，就相当于有了一段具有特定功能的代码。函数在定义完成后不会立刻执行，要执行这些代码，就需要调用函数。调用函数的方式非常简单，通过“函数名()”即可完成调用，其语法格式如下：

```
函数名([参数列表])
```

在定义【思考与练习5-1】的函数后，运行并输入hello()和add(40,80)，这都是在调用函数，示例代码如下：

```
hello()
add(40,80)
```

2. 查看函数的文档字符串

在前面的函数定义规则中提到，函数体的第1行语句可以选择性地使用文档字符串存

放函数说明。关于文档字符串有以下约定：

（1）第1行应为函数目的的简要描述。

（2）如果有多行，则第2行应为空白行，其目的是将摘要与其他描述从视觉上分隔开。

（3）后面几行应该是一个或多个段落，描述函数的调用约定、副作用等。

文档字符串及其约定其实是可选的而非必需的，没有设置文档字符串并不会造成语法错误。当然，如果用规范的文档字符串为函数增加注释，则可以为程序阅读者提供友好的提示和使用说明，提高函数代码的可读性。

Python中可以用内置函数help()或者通过“函数名.__doc__”来查看函数的文档字符串。

使用help()函数查看函数的文档字符串，其语法格式如下：

```
help(函数名)
```

使用“函数名.__doc__”查看函数的文档字符串，doc前后有两条下画线“__”，其语法格式如下：

```
print(函数名._doc_)
```

注意：如果想要查看函数的文档字符串，则该函数必须已经定义，是可调用函数。

【思考与练习5-2】

显示【思考与练习5-1】中函数的文档字符串。

3. return语句

定义函数中的return语句，用于将函数的处理结果返回给调用方，是函数的结束标志。若函数没有返回值，则可以省略return语句，等函数执行结束后将返回None。另外，单独的return语句(return后面没有任何内容)，也会返回None。None是Python中的特殊类型，代表“无”。

定义mod()函数，其功能是计算圆括号中的两个数的余数。mod(a,b)中的a是被除数，b是除数。当输入的b值为0时，函数没有意义，直接结束返回；当输入的b值不等于0时，则输出显示两个数的余数。使用return语句，根据条件判断，选择性地返回，代码如下：

```
//第5章/mod.py
def mod(a,b):
"""函数功能是计算两个数的余数"""
if b == 0:
return
else:
return a % b
```

分别输入mod(12,7)、mod(12,0)和print(mod(12,0))，观察输出结果。

5.2 函数的参数和返回值

在通常情况下，将在定义函数时设置的参数称为形式参数（简称形参），如 mod(a,b)函数，圆括号里面的 a、b 都是形参。将在调用函数时传入的参数称为实际参数（简称实参），如 mod(12,0)函数，其圆括号里面的数字 12、0 都是实参。“返回值”就是程序中的函数在完成一件事情后，最后返给调用者的结果。

5.2.1 函数的参数传递

函数的参数传递是指将实参传递给形参的过程。函数参数的传递方式可以分为位置参数的传递、关键字参数的传递、默认值参数的传递、不定长参数的传递等。下面将对函数参数的几种传递方式进行详细讲解。

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，即将第 1 个实参传递给第 1 个形参，将第 2 个实参传递给第 2 个形参，以此类推。

定义一个获取两个数之间最大值的 g_max()函数，并调用 g_max()函数，代码如下：

```
//第 5 章/g_max1.py
def g_max(a,b):
"""取两个数中的较大值并显示输出"""
if a > = b:
print("较大值是：",a)
else:
print("较大值是：",b)
g_max(56,78)
```

在上述代码中函数功能是取两个数中的较大值并显示输出。第 1～6 行代码用于定义 g_max()函数，第 1 行代码定义了能接收两个参数的函数，其中，a 为第 1 个形参，用于接收函数传递的第 1 个数值；b 为第 2 个形参，用于接收函数传递的第 2 个数值。第 7 行是函数的调用语句。如果想调用 g_max()函数，则需要给函数的参数传递两个数值。

注意：如果函数定义了多个参数，则在调用函数时，传递的实参要和形参一一对应，即调用函数时输入的参数数量和位置必须与定义时一致。

2. 关键字参数的传递

关键字参数是指通过形参名来确定输入的参数值。在通过该方式指定实参时，不需要与形参的位置完全一致，只需将参数名写正确。这样可以避免用户需要牢记参数位置的麻烦，使函数参数传递更加灵活、方便。

定义一个获取两个数之间最大值的 g_max()函数，并调用 g_max()函数，代码如下：

```
//第 5 章/g_max2.py
def g_max(a,b):
"""取两个数中的较大值并显示输出"""
if a >= b:
print("较大值是: ",a)
else:
print("较大值是: ",b)
g_max(b = 56,a = 78)
```

运行程序,输出如下:

```
较大值是:78
```

调用 g_max()函数,并通过关键字参数指定实参。虽然在指定实参时,顺序和在定义函数时不一致,但是运行结果和预期是一致的。

3. 默认值参数的传递

在调用函数时,如果没有指定某个参数,则将抛出异常。为了解决这个问题,可以为参数设置默认值,即在定义函数时直接指定形参的默认值。这样一来,当没有传入参数时,则直接使用在定义函数时设置的默认值。在定义函数时,指定默认值的形参必须在所有参数的最后,否则将会产生语法错误。

以某大学社团纳新的会员管理为例,因为社团招纳的新会员一般为大一学生,所以在会员管理中可以将会员年级参数的默认值设置为大一学生,代码如下:

```
//第 5 章/new_member1.py
def new_member(name,student_id,age = 18,grade = "大一学生"):
print("姓名:",name)
print("学号:",student_id) print("年龄:",age)
print("年级:",grade)
print("****************************")
return
new_member("毛毛","20210102001",20,"大二学生")
new_member("苗苗","20220101001")
```

在上述代码中第 1～7 行代码用于定义 new_member()函数,该函数的功能是显示输出新会员的姓名、学号、年龄和年级。第 8 行和第 9 行代码是函数的调用语句。第 8 行代码将函数中的 4 个参数全部输入,按输入内容显示会员信息;第 9 行代码只输入了参数“姓名”及“学号”,没有输入后面的两个参数,但由于在设置函数时,分别对后面的两个参数 age 和 grade 设置了默认值“18”和“大一学生”,因此当这两个参数被省略时会按默认值输出。

运行程序,输出如下:

```
姓名 : 毛毛
学号 : 20210102001
年龄 : 20
```

```
年级 : 大二学生
**************************
姓名 : 苗苗
学号 : 20220101001
年龄 : 18
年级 : 大一学生
**************************
```

4. 不定长参数的传递

通常在定义一个函数时，如果无法明确需要的参数个数，则可以在定义函数时使用不定长参数。不定长参数的定义有两种，一种是 * args，另一种是 ** kwargs，前者接收多个实参并将其放在一个元组中，后者则接收任意多个类似关键字参数的字典，基本的语法格式如下：

```
det 函数名 ([formal_args,] * args, ** kwargs):
"""文档字符串"""
函数体
[return[表达式]]
```

在上述语法格式中，formal_args 为形参，* args 和 ** kwargs 为不定长参数。在调用函数时，函数传入的参数个数会优先匹配 formal_args 参数的个数。如果传入的参数个数与 formal_args 参数的个数相同，则不定长参数会返回空的元组或字典；如果传入参数的个数比 formal_args 参数的个数多，则可以分为以下两种情况。

（1）如果传入的参数没有指定名称，则 * args 会以元组的形式存放这些多余的参数。

（2）如果传入的参数指定了名称（如 m＝1），则 ** kwargs 会以字典的形式存放这些被命名的参数（如{m：1}）。

为了更好地理解，通过不定长参数的应用进行演示，代码如下：

```
//第 5 章/test.py
def test(a,b, * c, ** d):
print(a,end = " ")
print(b,end = " ")
print(c,end = " ")
print(d)
test(1,2)
test(1,2,3,4,5,6,7,8)
test(1,2,3,4,5,6,7,8,m = 9,n = 10)
```

在上述代码中第 1～5 行代码用于定义 test()函数，其中有多个参数，包括形参 a 和 b，以及不定长参数 * c 和 ** d。end＝" "表示末尾不换行，加空格显示。第 6～8 行代码是函数的调用语句。在第 6 行函数调用语句的 test()函数中输入了两个参数，其中实参 1 和 2 对应形参 a 和 b；不定长参数 * c 和 ** d 没有实参与之对应，所以不定长参数 * c 和 ** d 为空。在第 7 行函数调用语句的 test()函数中输入了 8 个参数，其中实参 1 和 2 对应形参 a 和

b；实参 3,4,5,6,7,8 对应不定长参数 * c,以元组形式存储；不定长参数 ** d 没有参数与之对应,所以为空。在第 8 行函数调用语句的 test() 函数中输入了 10 个参数,其中实参 1 和 2 对应形参 a 和 b；实参 3、4、5、6、7、8 对应不定长参数 * c,以元组形式存储；最后两个参数 m=9,n=10 指定了名称,不定长参数 ** d 将以字典的形式存放这些被命名的参数。

运行程序,输出如下：

```
1 2 () {}
1 2 (3, 4, 5, 6, 7, 8) {}
1 2 (3, 4, 5, 6, 7, 8) {'m': 9, 'n': 10}
```

5.2.2 函数参数标注

函数的返回值及函数的形参都可以不指定类型,但是这往往会导致在阅读程序或函数调用时无法知道参数的类型。Python 提供了“函数参数标注”的手段为形参标注类型。函数标注是关于用户在自定义函数中使用的参数类型的数据信息,以字典的形式存放在函数的“__annotations__”属性中,并且不会影响函数的任何其他部分。在定义函数时,位置参数、默认参数及函数返回值都可以标注类型,其中,形参的标注方式是在形参后加冒号和数据类型,函数返回值的标注方式是在形参列表和 def 语句结尾的冒号之间加上复合符号“->”和数据类型。值得注意的是,函数标注仅标注了参数或返回值的类型,并不会限定参数或返回值的类型。在对函数进行定义和调用时,参数和返回值的类型是可以改变的。

以某大学社团纳新的会员管理为例,在定义函数时添加参数和返回值的标注类型,代码如下：

```
//第 5 章/new_member2.py
def new_member(name:str,id:str,age:str = "18",grade:str = "大一学生") -> str:
print("姓名:",name)
print("学号:",id)
print("年龄:",age)
print("年级:",grade)
print(type(name))
print(type(age))
print("****************************")
return
new_member("毛毛","20210102001",20,"大二学生")  #第 10 行
new_member("苗苗","20220101001")
```

在上述代码中第 1～9 行代码用于定义 new_member()函数,其功能是显示输出新会员的姓名、学号、年龄和年级,并显示输出变量 name 和 age 的数据类型。在定义函数时,形参的标注方式是在形参后加冒号和数据类型,函数返回值的标注方式是在形参列表和 def 语句结尾的冒号之间加上复合符号“->”和数据类型。第 10 行和第 11 行代码是函数的调用语句。第 10 行代码将函数中的 4 个参数全部输入,其中第 3 个参数 age 输入的是数字类型数据 20。程序会按输入内容显示会员信息及变量 name、age 的数据类型。第 11 行代码只输

入了参数“姓名”及“学号”，没有输入后面的两个参数，但由于在设置函数时，分别对后面的两个参数 age 和 grade 设置了默认值“18”和“大一学生”，所以当这两个参数被省略时，按默认值输出。

运行程序，输出如下：

```
姓名 : 毛毛
学号 : 20210102001
年龄 : 20
年级 : 大二学生
<class 'str'>
<class 'int'>
***************************
姓名 : 苗苗
学号 : 20220101001
年龄 : 18
年级 : 大一学生
<class 'str'>
<class 'str'>
***************************
```

因为在调用函数时，new_member("毛毛","20210102001",20,"大二学生")虽然在定义时标注了 age 形参应该是字符型，但是在实际的实参赋值时 age 形参存储的是数字类型数据 20，所以在显示数据类型时我们看到毛毛的 age 数据类型是 int。也就是说，函数标注仅标注了参数或返回值的类型，并不会限定参数或返回值的类型，在对函数进行定义和调用时，参数和返回值的类型都是可以改变的。

5.2.3　函数的返回值

所谓“返回值”，也就是程序中的函数在完成一件事情后，最后给调用者的结果，例如，定义一个累加函数，用于数值累加运算，一旦调用这个函数，函数就会把累加运算的值返给调用者，这个累加运算的值就是函数的返回值。在 Python 语言中，函数的返回值是使用 return 语句来完成的，同时让程序回到函数被调用的位置继续执行。

函数返回值的应用，代码如下：

```
def add(a,b,c,d):
m = a + b + c + d
return m
```

上述代码定义了add()函数，其功能是对参数的值进行累加，并通过 return 语句将结果返回。

运行程序，输入 add(1,2,3,4)，输出如下：

```
10
```

函数中包含 return 语句，意味着这个函数有一个返回值，即返回参数值的累加结果。函数中的 return 语句会在函数结束时将数据返给程序，同时让程序回到函数被调用的位置继续执行。

8min

5.3 函数的递归

递归是一种特殊的函数调用形式，使函数在定义时可以直接或间接地调用其他函数。若函数在定义时直接或者间接地调用了自身，则这个函数被称为递归函数。递归函数通常用于解决结构相似的问题，并采用递归的方式，先将一个复杂的大型问题转换为与原问题结构相似且规模较小的若干子问题，再对最小化的子问题求解，从而得到原问题的解。

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。

递归函数的执行过程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：在遇到终止条件时，沿着递推往回一级一级地把值返回来。一般递归函数的语法格式如下：

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

递归最经典的应用便是阶乘。在数学中，求正整数阶乘(n!)问题可以根据 n 的取值分为以下两种情况。

（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)!。

在利用递归求解阶乘时，n=1 是边界条件，n×(n−1)!是递归公式。

例如用递归方式实现正整数的阶乘，代码如下：

```
//第 5 章/mm.py
def mm(n):
"""用递归方式求 n 的阶乘"
if n == 1:
return 1
else:
return n * mm(n - 1)
print(mm(5))
```

运行程序，输出如下：

```
120
```

5.4　Python 内置函数

内置函数是 Python 预先设置好的函数，不仅能自动加载，而且能直接使用。下面我们分类介绍常用的 Python 内置函数。

5.4.1　数学运算函数

数学运算函数与数学运算相关，一般函数参数是数字类型的，返回值也是数字类型的。常见的数学运算函数见表 5-1。

表 5-1　常见的数学运算函数

函数名	说　　明	示　　例
abs()	返回参数的绝对值	abs(－5.5)返回 5.5
divmod()	返回两个数值的商和余数	divmod(8,5)返回(1,3)
max()	返回所有参数中的最大值	max(3,－4,8,7)返回 8
min()	返回所有参数中的最小值	mix(3,－4,8,7)返回－4
pow()	返回两个参数的幂运算	pow(3,2)返回 9
round()	返回浮点数的四舍五入值	round(3.5654,2)返回 3.57;round(3.5654)返回 4
sum()	返回数字类型序列中的所有元素的和	sum({2,3,4,5})或 sum((2,3,4,5))返回 14

5.4.2　字符串运算函数和方法

字符串运算函数与字符串运算相关，一般函数参数是字符型数据，但返回值类型多样。详细函数见第 4 章字符串常用方法。

5.5　Python 模块

为了编写可维护的代码，通常会把不同功能的代码分别存放到不同的文件中，这样每个文件包含的代码相对较少，这种组织代码的方式被称为模块编程。

5.5.1　模块的概念

在 Python 语言中，由于模块是一个包含变量、语句、函数或类的程序文件，文件的名称就是模块名加“.py”扩展名，所以用户编写程序的过程，也就是编写模块的过程。模块往往体现为多个函数或类的组合，可以被其他应用程序调用。

采用模块编程主要有以下几个优点。

(1) 提高代码的可维护性：在应用系统开发过程中，合理划分程序模块，可以很好地完

成程序功能的定义，有利于代码维护。

（2）提高代码的可重用性：模块是按功能划分的程序，编写好的Python程序以模块的形式保存，只要在其他程序中引用该模块，就可以调用该模块中的函数，从而达到代码重用的目的。程序中使用的模块既可以是用户自定义的模块、Python内置模块，也可以是来自第三方的模块。

（3）有利于避免命名冲突：相同名称的函数和变量可以分别存放在不同的模块中，用户在编写模块时，不需要考虑模块间变量名冲突的问题，但需要注意的是，尽量不要与内置函数名发生冲突。

5.5.2 模块的分类

在Python语言中，模块分为3类：内置模块（标准库）、第三方模块和用户自定义模块。

1. 内置模块（标准库）

内置模块也被称为标准库。此类模块是随Python安装包一起发布的，是Python运行的核心，提供了系统管理、网络通信、文本处理等功能。标准库中有些模块的使用方法和用户自定义模块一样，需要先用import语句引用，才可以使用其中定义的函数，而另一些模块则被包含在Python解释器中，使用时不需要引用，就可以直接使用其中的函数，这部分函数就是前面介绍的内置函数。

2. 第三方模块

第三方模块也被称为第三方库，是在Python发展过程中针对各种领域，如科学计算、Web开发、数据库接口、图形系统等逐步形成的，需要安装才能使用。

3. 用户自定义模块

用户自定义模块，也就是用户自己在项目中定义的模块。在自定义模块中，用户可以添加自定义函数，并根据程序需要进行调用。

5.5.3 模块的使用

1. 导入模块

应用程序要调用一个模块中的变量或函数，需要先导入该模块。导入模块可使import或from语句。

1）import 语句

在Python语言中，如果要引用一些内置的函数，则需要使用import语句来导入某个模块，并且可以使用as关键字为导入的模块指定一个别名。import语句的语法格式如下：

```
import 模块名 1[as 别名 1,模块名 2 as 别名 2,… ,模块名 N as 别名 N]
```

模块导入后，需要通过模块名或模块的别名来调用函数，具体语法格式如下：

```
模块名.函数名
```

为什么必须加上模块名呢？因为可能存在这样一种情况：在多个模块中含有相同名称的函数，此时如果只通过函数名来调用，则解释器无法判断要调用哪个函数。

```
import math                # 使用 import 语句导入 Python 内置模块 math
print(math.sqrt(25))       # 调用 math 模块中的求平方根函数 sqrt()
print(math.pi)             # 调用 math 模块中的常数函数 pi
```

程序的运行结果如下：

```
5.0
3.141592653589793
```

在导入 math 模块时使用 as 关键字为其指定了一个别名 m，因此在调用 math 模块中的函数时必须使用别名进行调用，而不能使用模块名进行调用。

2）from 语句

有时需要用到模块中的某个函数，只需引入该函数，此时可以通过 from 语句导入模块中的指定函数。from 语句的语法格式如下：

```
from 模块名  import 函数名
```

通过 from 语句导入的函数可以直接使用，不需要通过模块名或别名来指明函数所属的模块。当两个模块中含有相同名称的函数时，后面的引入会覆盖前面的引入。也就是说，假如在模块 A 和模块 B 中均有 function()函数，如果引入模块 A 中的 function()函数先于引入模块 B 中的 function()函数，则在调用 function()函数时，执行的是模块 B 中的 function()函数。

如果想一次性引入 math 模块中的所有内容，则可以通过 from math import * 语句来实现，但是不建议这么做，因为该方法只在下面两种情况下建议使用。

(1) 目标模块中的属性非常多，反复输入模块名很不方便。

(2) 在交互式解释器中，这样可以减少输入。

使用 from 语句导入模块，代码如下：

```
# 使用 from 语句导入 math 模块中的求余数函数 fmod 和求绝对值函数 fabs
from math import fmod,fabs
print(fmod(10,3))          # 调用 fmod()函数求 10 除以 3 的余数
print(fabs(-8))            # 调用 fabs()函数求 -8 的绝对值
```

程序的运行结果如下：

```
1.0
8.0
```

2. 创建模块

在 Python 语言中，每个 Python 文件都可以作为一个模块，模块的名称就是文件的名

称。假设创建一个文件 test.py,在文件中定义一个函数 add,用于计算两个数之和。

创建 test.py 文件,定义 add()函数,代码如下:

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

此时,如果想在 main.py 文件中使用 test.py 文件的 add()函数,则可以使用 import test 语句导入 test 文件,并通过 test.add(x,y)语句进行调用。

创建 main.py 文件,导入 test.py 文件,通过键盘输入两个数,调用 test 文件中的 add()函数,求两个数之和,代码如下:

```
import test
x = int(input("x = "))
y = int(input("y = "))
print("x + y = ",test.add(x,y))
```

程序的运行结果如下:

```
x = 10                    #通过键盘输入 x 的值为 10
y = 20                    #通过键盘输入 y 的值为 20
x + y = 30                #输出 x + y 的结果为 30
```

3. 模块搜索路径

在使用 import 语句导入模块时,需要先查找到模块程序的位置,即模块的文件路径,因为这是调用或执行模块的关键。在导入模块时,解释器会进行搜索,以便找到模块所在的位置。搜索按以下顺序进行。

(1) 当前工作目录,即包含 import 语句的代码。

(2) 操作系统的 PYTHONPATH 环境变量中包含的目录。

(3) Python 默认的安装路径。

在导入模块时,不能在 import 或 from 语句中指定模块文件的路径,只能使用 Python 设置的搜索路径。标准模块 sys 的 path 属性可以用来查看搜索路径设置。在交互环境下,可以用以下方式查看搜索路径:

```
import sys
sys.path
```

输出类似以下的结果,显示当前环境的搜索路径:

```
#路径示范,非全部路径
['','C:\\Users\\86155\\AppData\\Local\\Programs\\Python\\Python37\\Lib\\idlelib','C:\\Users
\\86155\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip','C:\\Users\\86155\\
AppData\\Local\\Programs\\Python\\Python37\\DLLs']
```

其中,路径列表的第 1 个元素为空字符串,代表当前目录。导入模块时,解释器会按照

列表顺序进行搜索，直到找到第1个模块。如果模块所在路径不在搜索路径中，则可以调用sys.path中的append()函数来增加模块所在的绝对路径，示例代码如下：

```
sys.path.append("f:\第4章")
sys.path
```

输出结果中显示增加了“f:\第4章”的路径：

```
['','C:\\Users\\86155\\AppData\\Local\\Programs\\Python\\Python37\\Lib\\idlelib','C:\\Users\\86155\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip','C:\\Users\\86155\\AppData\\Local\\Programs\\Python\\Python37\\DLLs', 'f:\\第4章']
```

这种将需要的路径增加到搜索路径中的方法在重新启动解释器时会失效。

4. __name__属性

在Python语言中，每个文件都有两种使用方法，第1种是直接作为独立代码执行，第2种是在执行导入操作时，导入的模块将被执行。如果想要控制Python模块中的某些代码在导入时不执行，而模块独立运行时才执行，则可以使用__name__属性来实现。

__name__属性是Python的内置属性，用于表示当前模块的名称。如果Python文件作为模块被调用，则__name__的属性值为模块文件的主名；如果模块独立运行，则__name__属性值为__main__。

if __name__ == 'main'语句的作用是控制这两种不同情况下执行代码的过程，当__name__的值为main时，文件作为脚本直接执行，而当使用import或from语句导入其他程序时，模块中的代码是不会被执行的。

__name__属性的测试。factorial.py模块文件定义了一个求阶乘的fac()函数，代码如下：

```
//第5章/factorial.py
def fac(x):     #返回x的阶乘
a = 1
for i in range(1,x + 1):
a = a * i
print(x,"阶乘为 ",a)

if _name_ == "_main_":
print("please use me as a module.")
```

当factorial.py模块文件独立运行时，其__name__值为__main__，运行结果如下：

```
please use me as a module.
```

创建一个ex0424.py文件，在该文件中使用import语句导入factorial模块，并调用fac()函数计算x的阶乘，代码如下：

```
import factorial
x = int(input("x = "))
factorial.fac(x)
```

在运行 ex0424.py 文件时，由于 factorial.py 模块作为模块被调用，此时由于__name__的属性值为 factorial，而不是__main__，因此 factorial.py 模块文件中的 print("please use me as a module.")语句没有被执行，运行结果如下：

```
x = 5      #通过键盘输入 x 的值为 5
5 的阶乘为  120
```

5. 包

包是 Python 引入的分层次的文件目录结构，定义了一个由模块、子包及子包下的子包等组成的 Python 应用环境。引入包以后，只要顶层的包名不与其他包的名称冲突，那么所有模块都不会与其他包的名称冲突。

Python 的每个包目录下面都会有名为__init__.py 的特殊文件，该文件可以直接是一个空文件，但必须存在。它表明这个目录不是普通的目录结构，而是一个包，里面包含模块。Python 的包下面还可以有子包，即可有多级目录，以便组成多级层次的包结构。同样地，每个子包文件夹下也都需要一个__init__.py 文件。

从包中导入单独的模块可以使用 import PackageA.SubPackageA.ModuleA 语句，使用时必须用全路径名，也可以使用它的变形语句，即 from PackageA.SubPackageA import ModuleA，在使用时可以直接使用模块名而不用加上包前缀；还可以直接导入模块中的函数或变量，即 from PackageA.SubPackageA.ModuleA import functionA。

具体说明如下：

(1) 当使用 from package import item 语句时，item 既可以是 package 的子模块或子包，也可以是其他定义在包中的名称(如一个函数、类或变量)。首先检查 item 是否定义在包中，如果没找到它，就认为 item 是一个模块并尝试加载它，当加载失败时会抛出一个 ImportError 异常。

(2) 当使用 import item.subitem.subsubitem 语句时，最后一个 item 之前的 item 必须是包，最后一个 item 可以是一个模块或包，但不能是类、函数和变量。

(3) 当使用 from package import * 语句时，如果包的__init__.py 定义了一个名为__all__的列表变量，则它包含模块名称的列表将被导入模块列表；如果没有定义__all__变量，则这条语句不会导入所有的 package 的子模块，而是只保证 package 包被导入。

5.5.4 正则表达式模块

正则表达式本质上是一个特殊的字符序列，可以检查一个字符串是否与某种模式匹配，Python 中通过 re 模块实现正则表达式。该模块提供 Perl 风格的正则表达式匹配模式，可以支持全部的正则表达式功能。

1. re.match()函数

re.match()函数尝试从字符串的开始位置匹配一种模式，如果匹配成功，则 re.match()函数返回一个匹配的对象，否则返回 None，语法格式如下：

```
re.match(pattern, string, flags)
```

其中，pattern 参数为匹配的正则表达式；string 为要匹配的字符串；flags 为可选参数，用于控制正则表达式的匹配方式，如说明是否区分大小写。

使用 re.match()函数，创建 ex0425.py 文件，在导入 re 模块后，对字符串 Hello World 进行匹配，代码如下：

```
//第 5 章/ex0425.py
import re
s = "Hello world!"
print(re.match("world", s))
print(re.match("hello", s))
print(re.match("Hello", s))
print(re.match("hello", s, re.I))   #re.I 表示匹配不区分大小写
```

运行 ex0425.py 文件，程序的运行结果如下：

```
None
None
<re.Match object; span=(0, 5), match='Hello'>
<re.Match object; span=(0, 5), match='Hello'>
```

2. re.search()函数

re.search()函数会扫描整个字符串并返回一个成功的匹配对象。如果匹配失败，则返回 None，语法格式如下：

```
re.search (pattern, string, flags)
```

其中，pattern 参数为匹配的正则表达式，string 为要匹配的字符串，flags 为标志位。使用 re.search()函数，将 match 方法修改为 search 方法，对字符串 Hello World 进行匹配，代码如下：

```
//第 5 章/ex0426.py
import re
s = "Hello world!"
print(re.search("world", s))
print(re.search("hello", s))
print(re.search("Hello", s))
print(re.search("hello", s, re.I))
```

运行 ex0426.py 文件，程序的运行结果如下：

```
<re.Match object; span=(6, 11), match='world'>
None
<re.Match object; span=(0, 5), match='Hello'>
<re.Match object; span=(0, 5), match='Hello'>
```

从结果可以发现 re.match()函数与 re.search()函数的区别：re.match()函数仅匹配字符串开头，如果字符串开头不符合正则表达式，则匹配失败，返回 None，而 re.search()函数会匹配整个字符串，直到找到一个匹配对象，如果整个字符串都没有匹配对象，则返回 None。

3. 匹配模式

模式字符串(pattern)可以使用特殊的语法来表示一个正则表达式。常用的模式字符串见表 5-2。

表 5-2 常用的模式字符串

模 式	说 明
.	匹配除"\r\n"之外的任何单个字符。如果要匹配包括"\r\n"在内的任何字符，则可使用像"[\s\S]"的模式
\	将下一个字符标记为特殊字符，例如，"\\n"匹配\n，"\n"匹配换行符。序列"\\"匹配"\"，而"\("则匹配"("，相当于"转义字符"的概念
[xyz]	字符集合。匹配所包含的任意一个字符，例如，"[abc]"可以匹配"plain"中的"a"
[^xyz]	负值字符集合。匹配未包含的任意字符，例如，"[^abc]"可以匹配"plain"中的"plin"
[a-z]	字符范围。匹配指定范围内的任意字符，例如，"[a-z]"可以匹配"a"到"z"的任意小写字母字符 注意：只有连字符在字符组内部且出现在两个字符之间时，才能表示字符的范围；如果出现在字符组的开头，则只能表示连字符本身
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符，例如，"[^a-z]"可以匹配任何不在"a"到"z"的任意字符
\d	匹配一个数字字符。相当于[0-9]，例如，"a\dc"可以匹配"a2c"
\D	匹配一个非数字字符。相当于[^0-9]，例如，"a\dc"可以匹配"abc"
\s	匹配任何不可见字符，包括空格、制表符、换页符等。相当于[\f\n\r\t\v]
\S	匹配任何可见字符。相当于[^\f\n\r\t\v]
\w	匹配包括下画线的任何单词字符。类似但并非"[A-Za-z0-9_]"，这里的单词字符使用 Unicode 字符集
\W	匹配任何非单词字符。相当于"[^A-Za-z0-9_]"
*	匹配前面的子表达式任意次，例如，zo*既能匹配"z"，也能匹配"zo"及"zoo"
+	匹配前面的子表达式一次或多次(大于或等于 1 次)，例如，"zo+"能匹配"zo"及"zoo"，但不能匹配"z"。+相当于{1,}
?	匹配前面的子表达式零次或一次，例如，"do(es)?"可以匹配"do"或"does"中的"do"。? 等价于{0,1}
{n}	n 是一个非负整数。确定匹配 n 次，例如，"o{2}"不能匹配"Bob"中的"o"，但是能匹配"food"中的两个 o

续表

模 式	说 明
{n,}	n 是一个非负整数。至少匹配 n 次,例如,"o{2,}"不能匹配"Bob"中的"o",但能匹配"foooood"中的所有 o。"o{1,}"相当于"o+"。"o{0,}"则相当于"o*"
{n,m}	m 和 n 均为非负整数,其中 n<=m。最少匹配 n 次且最多匹配 m 次,例如,"o{1,3}"将匹配"fooooood"中的前 3 个 o。"o{0,1}"相当于"o?"。需要注意在逗号和两个数之间不能有空格
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性,则^也匹配"\n"或"\r"之后的位置
$	匹配输入字符串的结束位置。如果设置了 RegExp 对象的 Multiline 属性,则 $ 也匹配"\n"或"\r"之前的位置
\b	匹配一个单词边界,也就是指单词和空格间的位置(正则表达式的"匹配"有两种概念,一种是匹配字符,另一种是匹配位置,这里的\b 就是匹配位置的),例如,"er\b"可以匹配"never"中的"er",但不能匹配"verb"中的"er"
\B	匹配非单词边界。"er\B"能匹配"verb"中的"er",但不能匹配"never"中的"er"
\A	仅匹配字符串开头,例如,"\Aabc"可以匹配"abcdef"
\Z	仅匹配字符串末尾,例如,"abc\Z"可以匹配"defabc"

4. 可选标志

正则表达式可以包含一些可选标志(flags)来控制匹配模式。多个标志可以通过按位或表示,例如,"re.I|re.M"表示同时设置 I 和 M 两个标志。常用可选标志见表 5-3。

表 5-3 常用可选标志

标 志	说 明
re.I	使匹配对大小写不敏感
re.L	做本地化识别(locale-aware)匹配
re.M	多行匹配,影响 ^和 $
re.S	使匹配包括换行符在内的所有字符
re.U	根据 Unicode 字符集解析字符。这个标志影响\w、\W、\b、\B
re.X	该标志通过给予更加灵活的格式,以便用户将正则表达式写得更易于理解

5.6 实训任务 1——斐波那契数列

1. 任务需求

已知斐波那契数列"0 1 1 2 3 5 8 13 21 34 …",请输入数列项数 N,打印该数列前 N 项,例如,输入 5,打印结果为"0 1 1 2 3"。

2. 任务分析

根据斐波那契数列规律(第 1 项是 0,第 2 项是 1,从第 3 项开始,每项都等于前两项的和),可以定义函数计算出斐波那契数列第 N 项的数值,如果要打印前 N 项的数值,则只需循环调用。

3. 任务实施

步骤 1：定义计算斐波那契数列的函数。

创建 Python 文件 fib.py，并定义 fib(n)函数，用于计算斐波那契数列第 N 项的数值，代码如下：

```
//第 5 章/fib.py
def fib(n):
if n == 1:
return 0
elif n == 2:
return 1
n1 = 0
n2 = 1
n3 = 0
for i in range(2, n):
#下一项的数值 = 前两项之和
n3 = n1 + n2
#更新 n1 和 n2 项
n1 = n2
n2 = n3
return n3
```

输入数列的项数 N，通过 for 循环依次调用 fib()函数，求出每项的数值，并打印输出，代码如下：

```
n = input("打印斐波那契数列前 N 项,请输入 N: ")
n = int(n)
for item in range(1,n+1):
print(fib(item), end=" ")
print()
```

步骤 2：运行程序。

运行上述代码，根据提示输入具体的项数后，可以打印出对应的斐波那契数列，如图 5-1 所示。

图 5-1 斐波那契数列

步骤 3：使用递归函数计算斐波那契数列。

使用函数递归思想，改进计算斐波那契数列的函数，从第 3 项开始，可以通过自递归，即

直接调用函数自身 fib(n－1) ＋ fib(n－2)进行计算，这样可以简化代码实现过程，代码如下：

```
//第 5 章/fib2.py
def fib(n):
if n == 1:
return 0
elif n == 2:
return 1
return fib(n - 1) + fib(n - 2)
```

使用上述代码替换步骤 1 中的 fib()函数，可以得到相同的执行结果，但是当数据项较大时（例如，输入 N 为 40)会发现代码执行要花费很长时间，所以使用递归函数虽然简化了代码实现，但运行效率会受到影响。

5.7　实训任务 2——人脸检测与识别模块

1. 任务需求

人脸识别是基于人的脸部特征信息进行身份识别的一种生物识别技术，可以对含有人脸的图像进行自动检测，进而对图像中的脸部特征进行识别，通常也被称为人像识别、面部识别。

本任务将使用 baidu-aip 模块对包含人脸的 JPG 格式图片进行检测和识别，识别结果包含年龄、颜值、面部表情、脸型、性别、是否戴了眼镜等相关信息。

2. 任务分析

人脸识别算法的相关内容较为复杂，可以借助百度 AI 开放平台解决。该平台提供了一套关于人工智能的编程接口(baidu-aip)，让我们不需要了解算法的细节就可以实现人脸检测与识别功能。

3. 任务实施

步骤 1：安装百度 AI 平台的人脸识别模块。

在浏览器上搜索“百度 AI 开放平台”并进入其官方网站，首先在首页导航的“开放能力”下拉列表中先选择“人脸与人体”选项，然后选择右侧的“人脸识别云服务”选项，如图 5-2 所示。

在“人脸识别云服务”界面中，单击“立即使用”按钮，在弹出的登录界面中使用百度账号登录，如果没有对应账号，则可以单击“立即注册”文字链接进行注册，如图 5-3 所示。

第 1 次注册登录后，首先到“用户中心”完成个人实名认证，实名认证后可领取 2QPS 的测试资源，然后在“控制台总览”界面中选择“创建应用”选项，如图 5-4 所示。

根据提示指定应用名称(face_detect)、应用描述等信息。创建完成后，可以在“应用详情”界面中看到该应用的详情信息，其中 AppID、API Key 和 Secret Key 在将来使用百度提供的 API 编写程序时会用到，如图 5-5 所示。

图 5-2　百度 AI 开放平台

图 5-3　登录百度 AI 开放平台

在线安装 baidu-aip 和 chardet 模块。之后，由于在 Python 程序中调用人脸识别的接口函数就是由 baidu-aip 模块提供的，运行该模块还要依赖编码识别的 chardet 模块，所以既可以在控制终端使用在线包管理工具 pip 安装，也可以在 PyCharm 的图形界面中安装，如图 5-6 所示。

图 5-4　“应用详情”界面

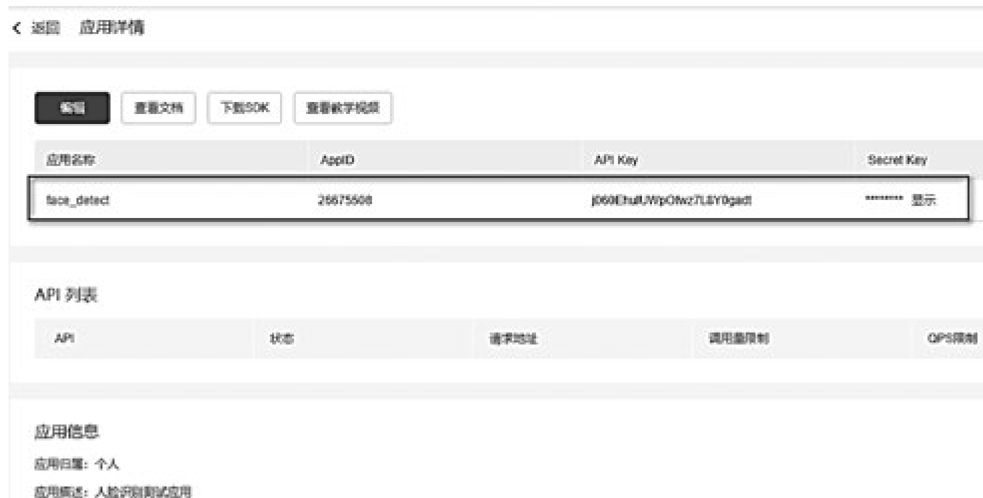

图 5-5　“应用详情”界面

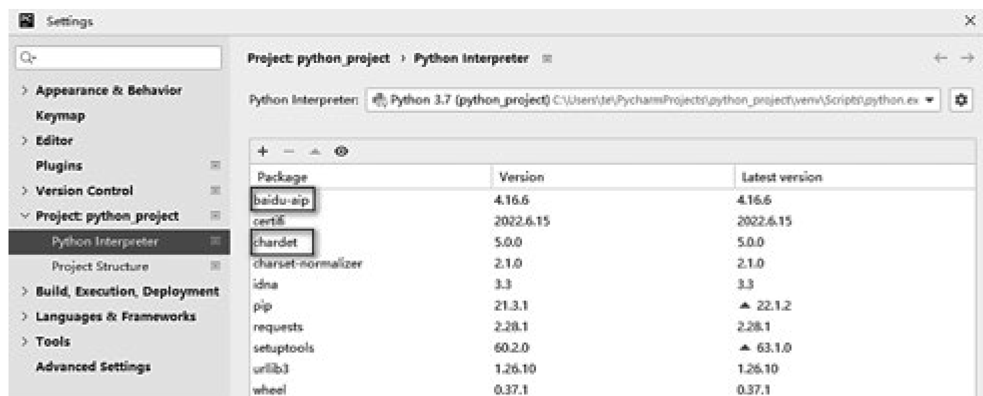

图 5-6　安装 baidu-aip 和 chardet 模块

步骤 2：连接百度人脸识别库。

在 PyCharm 中新建 Python 文件 face_detect_image.py，在该文件中导入 aip 包中与人脸识别相关的接口。设置 AppID、API Key 和 Secret Key，这些信息可以在百度 AI 开放平台的“应用详情”界面获取，直接复制相关信息即可。调用 AipFace 接口和人脸识别应用建立连接，代码如下：

```
#设置 APP_ID、API_KEY、SECRECT_KEY
APP_ID = "26675****"
API_KEY = "j060EhulUWpOfwz7L8Y0****"
SECRECT_KEY = "1sTVm3SPhzfau7QC017GiRSL3zAP****"
#连接百度人脸识别库
client = AipFace(APP_ID, API_KEY, SECRECT_KEY)
```

步骤 3：配置人脸识别选项。

配置人脸识别选项和要识别的图片，首先通过字典的“键”(face_field)定义需要识别的参数，然后准备测试的图片(images/test.jpg)，将该图片放在项目同级目录下，并确定里面包含人脸的图像信息，代码如下：

```
#配置选项
options = {
#人脸检测选项:年龄、颜值、面部表情、脸型、性别、是否戴了眼镜、人种、脸的形状
"face_field":"age,beauty,expression,faceshape,gender,glasses,race,facetype"
}
#指定要识别的图片
filename = './images/test.jpg'
```

步骤 4：打开图片进行人脸识别。

打开测试图片，检测里面的人脸图像并进行识别(注意：图片要使用 Base64 进行编码)，识别完成后会以 JSON 格式返回识别结果，代码如下：

```
#打开图片,对图片进行识别并打印结果
with open(filename, 'rb') as f:
#将读取到的内容经过 base64 模块中的 b64encode 编码成字符串
image = str(base64.b64encode(f.read()), 'utf-8')
#类型为 BASE64
imageType = 'BASE64'
#调用人脸识别接口
data = client.detect(image, imageType, options)
print(data)
```

步骤 5：解析人脸识别结果。

默认的识别结果是 JSON 格式的，可以被看作 Python 中的字典，其中会包含一些冗余信息，如果希望获取人脸识别结果，则只需关注 result 中的 face_list。该列表中的元素为字典类型，可以通过具体的 key 获取相应的识别结果，代码如下：

```
#获取人脸识别结果
face = data['result']['face_list'][0]
#对结果进行解析
print('age:', face['age'])
print('beauty:', face['beauty'])
print('expression:', face['expression']['type'])
print('face_shape:', face['face_shape']['type'])
print('gender:', face['gender']['type'])
print('glasses:', face['glasses']['type'])
print('race:', face['race']['type'])
```

步骤 6：运行程序。

运行程序进行人脸识别测试，可以看到打印的结果，其中 age 表示年龄，beauty 表示颜值，expression 表示面部表情，face_shape 表示脸型，gender 表示性别，glasses 表示是否戴了眼镜，race 表示人种，具体数值如图 5-7 所示。

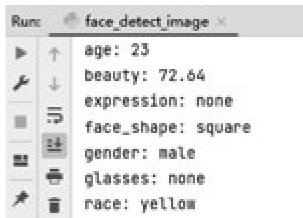

图 5-7 人脸识别结果

本章总结

本章重点介绍了 Python 中函数的定义和调用，并详细说明了函数参数的传递方式，以及 Python 中内置函数的使用。另外，本章还扩展介绍了模块的概念、分类和使用。

函数是组织好的、可重复使用的，用来实现单一或相关联功能的代码段。它能够提高应用的模块化和代码的重复利用率。在程序中可以通过调用函数来提高代码的复用性，从而提高编程效率及程序的可读性。

函数按参数传递可以分为位置参数、关键字参数、默认值参数、不定长参数。

递归是一种特殊的函数调用形式，函数在定义时可以直接或间接地调用其他函数。若函数在定义时直接或间接地调用了自身，则这个函数被称为递归函数。

模块是一个包含了一系列函数的 Python 程序文件，若将一系列的模块文件放在同一个文件夹中，则构成了包。包是可以对模块进行层次化管理的有效工具，大大地提高了代码的可维护性和重用性。

用户既可以编写自己定义的函数、模块和包，也可以使用 Python 提供的各种包。Python 提供的包也被称为内置函数库，更重要的是，除了 Python 的内置函数库，还有实现各种功能的第三方函数库。当需要这些功能时，只需将它们的代码导入自己的程序。这种基于大量第三方函数库的编程方式，正是 Python 语言的魅力所在。

re 模块提供了正则表达式功能，常用的函数包括 match()和 search()。

习题

作业与练习

1. 单选题

(1) 在下列选项中,关于函数的描述正确的是(　　)。

A. 函数用于创建对象
B. 函数可以重新执行得更快
C. 函数是一段代码,用于执行特定的任务
D. 以上说法都正确

(2) 在下列选项中,参数定义不合法的是(　　)。

A. def myfunc(* args):
B. def myfunc(arg1=1):
C. def myfunc(* args,a=1):
D. def myfunc(a=1, ** args):

(3) 在下列选项中,关于导入模块的方式,错误的是(　　)。

A. import math
B. from fib import fibonacci
C. from math import *
D. from * import fib

2. 填空题

(1) 使用(　　)语句可以返回函数值并退出函数。

(2) 如果函数中没有 return 语句或者 return 语句不带任何返回值,则该函数的返回值为(　　)。

(3) 在 Python 语言中,若定义了 f1(p, ** p2):print(type(p2)),则 f1(1,a=2)的运行结果是(　　)。

(4) 在 Python 语言中,若定义了 f1(a,b,c):print(a+b),则 nums=(1,2,3);f1(* nums)的运行结果是(　　)。

3. 编程题

(1) 编写一个函数当输入的 n 为偶数时,调用函数求 $\frac{1}{2}+\frac{1}{4}+\cdots+\frac{1}{n}$;当输入的 n 为奇数时,调用函数求 $\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$。

(2) 由键盘输入任意两个整数 x、y,编写一个函数,用来求 x 的 y 次方。