Python

极夜酱

目录

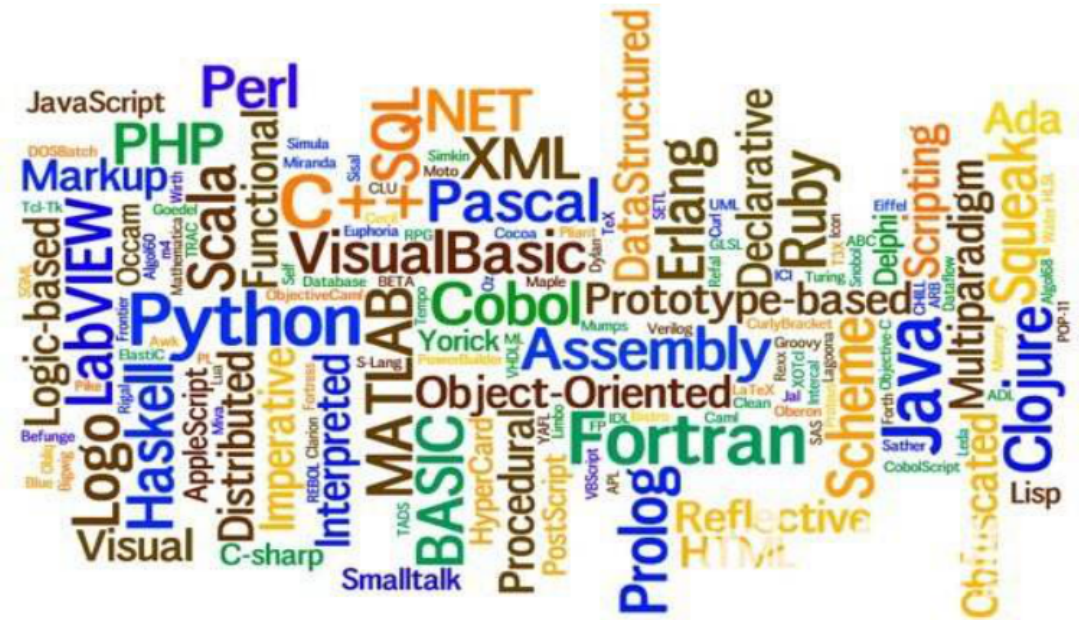
Chapter 1 Hello World!

1.1 Hello World!

1.1.1 编程语言（Programming Language）

程序是为了让计算机去解决某些问题，它由一系列指令构成。但是计算机并不能理解人类的语言，即使是最简单的，例如“计算一下 1+2 是多少”。

计算机采用的是二进制（binary），也就是只能够理解 0 和 1，因此编程语言用于作为人类与计算机之间沟通的桥梁。



通过使用编程语言来描述解决问题的步骤，从而让计算机一步一步去执行。流程图（flow chat）成为了一种程序的图形化表示方式。

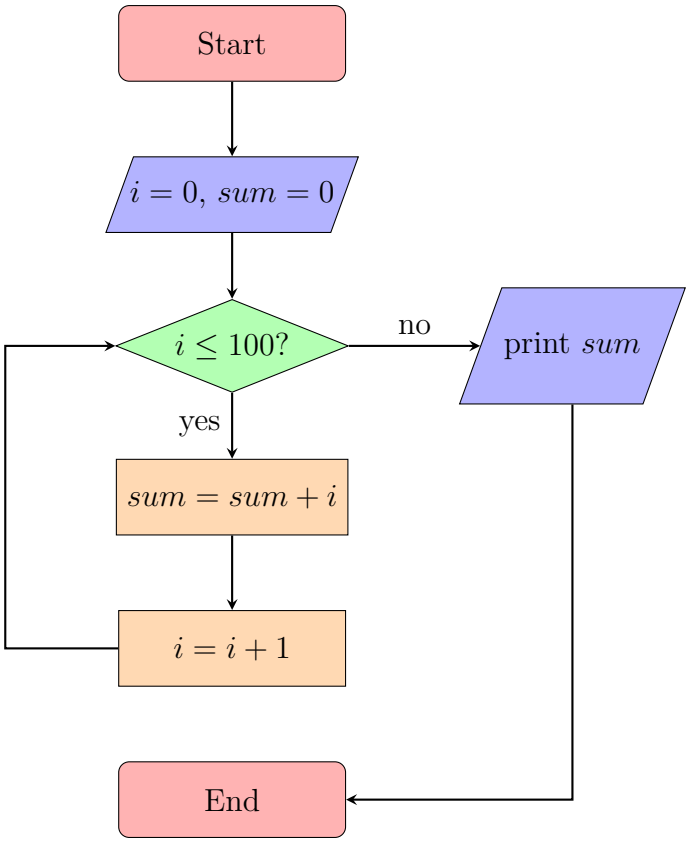


图 1.1: 计算 $\sum_{i=1}^{100} i$ 的流程图

1.1.2 Hello World!

Hello World 是学习编程的第一个程序，它的作用是向屏幕输出"Hello World!"。

Hello World!

```
1 print("Hello World!")
```

运行结果

```
Hello World!
```

不同编程语言的 Hello World 写法大同小异，可以看出编程语言的基本结构是相似的。

C

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

C++

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

Java

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

1.1.3 注释（Comment）

注释就是对代码的解释和说明，它并不会程序所执行。注释能提高程序的可读性，让人更加容易了解代码的功能。

注释一般分为单行注释和多行注释：

1. 单行注释：以 # 开头，该行之后的内容视为注释。
2. 多行注释：以""" 开头，""" 结束，中间的内容视为注释。

注释

```
"""
    Author: Terry
    Date: 2022/11/16
"""
print("Hello World!")       # display "Hello World!"
```

1.2 数据类型

1.2.1 数据类型（Data Types）

在计算机中，每个数据一般都有一个对应的类型，基础数据类型包括：

1. 数值型
    - 整数 int
    - 浮点数 float
    - 复数 complex
2. 文本型
    - 字符串 str
3. 布尔型 bool
4. 数据结构
    - 列表 list
    - 元组 tuple
    - 集合 set
    - 字典 dict

1.2.2 变量（Variable）

变量是用来存储数据的内存空间，每个变量都有一个类型，使用 type() 函数可以查看变量的类型。

```
num = 10;
print(type(num))    # <class 'int'>
salary = 8232.56;
print(type(salary)) # <class 'float'>
```

变量的命名需要符合规范：

1. 由字母、数字和下划线组成，不能以数字开头

2. 不可以使用编程语言中预留的关键字

3. 使用英语单词，顾名思义

关键字是编程语言内置的一些名称，具有特殊的用处和意义，因此不应该作为变量名，防止产生歧义。

False	None	True	and	as
assert	break	class	continue	def
del	elif	else	except	finally
for	from	global	if	import
in	is	lambda	nonlocal	not
or	pass	raise	return	try
while	with	yield		

表 1.1: 关键字

1.3 输入输出函数

1.3.1 print()

print() 的功能是向屏幕输出指定格式的文本，但是有些需要输出的字符在编程语言中具有特殊含义，因此这些特殊的字符，需要经过转义后输出。

转义字符	描述
\\	反斜杠 \
\'	单引号 '
\"	双引号 "
\n	换行
\t	制表符

表 1.2: 转义字符

转义字符

```
1 print("\"Hello\nWorld\"")
```

运行结果

```
"Hello
World"
```

除了直接使用 print() 输出一个变量的值外，还可以在 print() 中使用对应类型的占位符。

数据类型	占位符
int	%d
float	%f
str	%s

表 1.3: 占位符

长方形面积

```
length = 10
width = 5
area = length * width
print("Area = %d * %d = %.2f" % (length, width, area))
```

运行结果

```
Area = 10 * 5 = 50.00
```

另一种输出的方式是使用 f-string，它可以在字符串中直接使用变量的值。

```
print(f"Area = {length} * {width} = {area:.2f}")
```

在默认情况下，print() 函数输出数据后，会以换行作为结束符。如果不希望使用换行作为结束符，则可以在 print() 函数中追加一个 end 参数。

等比数列

```
num1 = 1
num2 = 2
num3 = 4
num4 = 8
print(num1, end=', ')
print(num2, end=', ')
print(num3, end=', ')
print(num4, end='...')
```

运行结果

```
1, 2, 4, 8...
```

1.3.2 input()

有时候一些数据需要从键盘输入，input() 可以读取用户输入，并赋值给相应的变量。

input() 读取到的数据类型是 str，通过转换函数可以将其转换为其它类型。

圆面积

```
1 import math
2
3 r = float(input("Radius: "))
4 area = math.pi * r ** 2
5 print("Area = %.2f" % area)
```

运行结果

```
Radius: 5
Area = 78.54
```

math 模块中定义了一些常用的数学函数，例如 pow(x, y) 可用于计算 x 的 y 次方。

1.4 表达式

1.4.1 算术运算符

整除运算符//用于计算两个数相除的整数部分，例如 $21 // 4 = 5$。

取模（modulo）运算符% 用于计算两个整数相除之后的余数，例如 $22 \% 3 = 1$、$4 \% 7 = 4$。

逆序三位数

```
num = int(input("Enter a 3-digit integer: "))
a = num // 100
b = num // 10 % 10
c = num % 10
print("Reversed:", c * 100 + b * 10 + a)
```

运行结果

```
Enter a 3-digit integer: 520
Reversed: 25
```

1.4.2 复合运算符

使用复合运算符可以使表达式更加简洁。例如 `a = a + b`可以写成 `a += b`, -=、*=、/=、%= 等复合运算符的使用方式同理。

字符串拼接

```
s = "Hello" + "World"
s += "!"
print(s)
```

运行结果

```
HelloWorld!
```

Chapter 2 分支

2.1 逻辑运算符

2.1.1 关系运算符

编程中经常需要使用关系运算符来比较两个数据的大小，比较的结果是一个布尔值（boolean），即 True（非 0）或 False（0）。

在编程中需要注意，一个等号 = 表示赋值运算，而两个等号 == 表示比较运算。

数学符号	关系运算符
$<$	<
$>$	>
$\leq$	<=
$\geq$	>=
$=$	==
$\neq$	!=

2.1.2 逻辑运算符

逻辑运算符用于连接多个关系表达式，其结果也是一个布尔值。

1. 逻辑与 and：当多个条件全部为 True，结果为 True。

条件 1	条件 2	条件 1 and 条件 2
T	T	T
T	F	F
F	T	F
F	F	F

2. 逻辑或 or：多个条件至少有一个为 True 时，结果为 True。

条件 1	条件 2	条件 1 or 条件 2
T	T	T
T	F	T
F	T	T
F	F	F

3. 逻辑非 not：条件为 True 时，结果为 False；条件为 False 时，结果为 True。

条件	not 条件
T	F
F	T

2.2 if

2.2.1 if

if 语句用于判断一个条件是否成立，如果成立则进入语句块，否则不执行。

年龄

```
age = int(input("Enter your age: "))
if 0 < age < 18:
    print("Minor")
```

运行结果

```
Enter your age: 17
Minor
```

2.2.2 if-else

if-else 的结构与 if 类似，只是在 if 语句块中的条件不成立时，执行 else 语句块中的语句。

闰年

```
year = int(input("Enter a year: "))

"""
    A year is a leap year if it is
    1. exactly divisible by 4, and not divisible by 100;
    2. or is exactly divisible by 400
"""
if year % 4 == 0 and year % 100 != 0 or year % 400 == 0:
    print("Leap year")
```

```
else:
    print("Common year")
```

运行结果

```
Enter a year: 2020
Leap year
```

2.2.3 if-elif-else

当需要对更多的条件进行判断时，可以使用 if-elif-else 语句。

字符

```
c = input("Enter a character: ")
if c >= 'a' and c <= 'z':
    print("Lowercase")
elif c >= 'A' and c <= 'Z':
    print("Uppercase")
elif c >= '0' and c <= '9':
    print("Digit")
else:
    print("Special character")
```

运行结果

```
Enter a character: T
Uppercase
```

Chapter 3 循环

3.1 while

3.1.1 while

while 循环会对条件进行判断，如果条件成立，就会执行循环体，然后再次判断条件，直到条件不成立。

while 循环的次数由循环变量的变化决定，因此 while 循环一般都包括对循环变量的初值、判断和更新。

```
1 i = 1           # initial value
2 while i <= 5:   # condition
3     print("In loop: i =", i)
4     i += 1      # update
5 print("After loop: i =", i)
```

while 循环的特点是先判断、再执行，因此循环体有可能会执行一次或多次，也有可能一次也不会执行。

平均身高

```
1  NUM_PEOPLE = 5
2
3  total = 0
4
5  i = 1
6  while i <= NUM_PEOPLE:
7      height = float(input("Enter person %d's height: " % i))
8      total += height
9      i += 1
10
```

```
11 average = total / NUM_PEOPLE
12 print("Average height: %.2f" % average)
```

运行结果

```
Enter person 1's height: 160.8
Enter person 2's height: 175.2
Enter person 3's height: 171.2
Enter person 4's height: 181.3
Enter person 5's height: 164
Average height: 170.50
```

整数位数

```
1 num = int(input("Enter an integer: "))
2 n = 0
3
4 while num != 0:
5     num //= 10
6     n += 1
7
8 print("Digits:", n)
```

运行结果

```
Enter an integer: 123
Digits: 3
```

猜数字

```
1 import random
2
3 answer = random.randint(1, 100)
```

```python
cnt = 0

while True:
    num = int(input("Guess a number: "))
    cnt += 1

    if num > answer:
        print("Too high")
    elif num < answer:
        print("Too low")
    else:
        break

print("Correct! You guessed %d times." % cnt)
```

运行结果

```
Guess a number: 50
Too high
Guess a number: 25
Too low
Guess a number: 37
Too low
Guess a number: 43
Too high
Guess a number: 40
Too high
Guess a number: 38
Too low
Guess a number: 39
Correct! You guessed 7 times.
```

3.2 for

3.2.1 for

while 循环将循环变量的初值、条件和更新写在了三个地方，但是这样不容易明显地看出循环变量的变化。

for 循环在一行内就可以清晰地表示出循环的次数，因此对于指定次数的循环一般更多地会采用 for 循环，而对于不确定次数的一般会采用 while 循环。

range() 函数能够生成指定范围的整数序列：

```
for i in range(5):
    print(i, end=' ')       # 0 1 2 3 4

for i in range(10, 15):
    print(i, end=' ')       # 10 11 12 13 14

for i in range(1, 10, 2):
    print(i, end=' ')       # 1 3 5 7 9
```

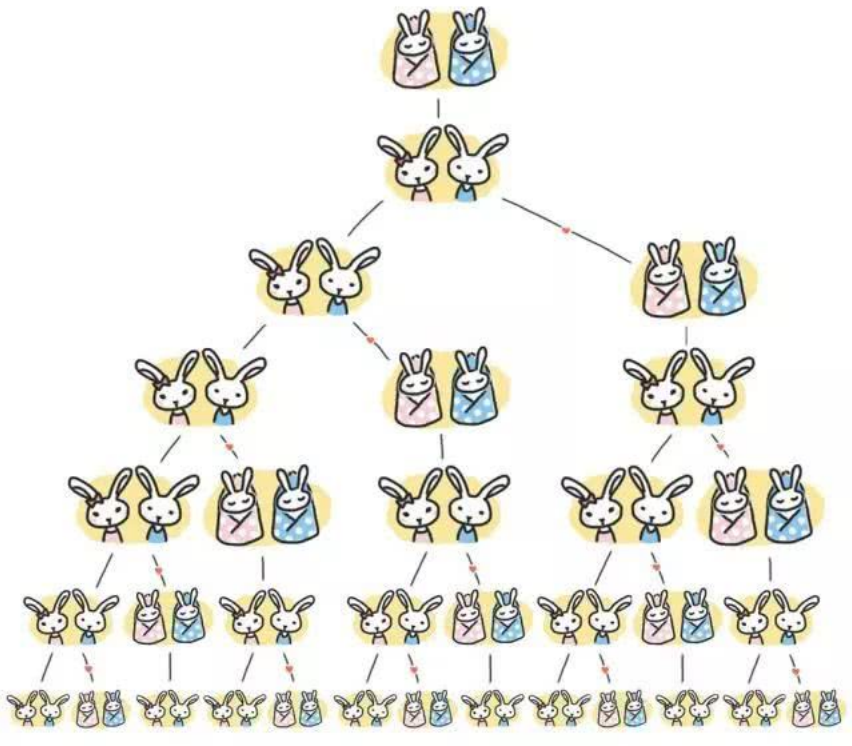
累加

```
sum = 0
for i in range(1, 101):
    sum += i
print("Sum =", sum)
```

运行结果

```
Sum = 5050
```

斐波那契数列



```python
n = int(input("Enter the number of terms: "))

if n == 1:
    print(1)
elif n == 2:
    print(1, 1)
else:
    num1 = 1
    num2 = 1
    print(1, 1, end=' ')
    for i in range(3, n + 1):
        val = num1 + num2
        print(val, end=' ')
        num1 = num2
        num2 = val
    print()
```

运行结果

```
Enter the number of terms: 10
1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

3.2.2 嵌套循环

循环也可以嵌套使用，外层循环每执行一次，内层循环就会执行多次。

```
1 for i in range(2):
2     for j in range(3):
3         print("i = %d, j = %d" % (i, j))
```

运行结果

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
```

九九乘法表

1*1=1	1*2=2	1*3=3	1*4=4	1*5=5	1*6=6	1*7=7	1*8=8	1*9=9
2*1=2	2*2=4	2*3=6	2*4=8	2*5=10	2*6=12	2*7=14	2*8=16	2*9=18
3*1=3	3*2=6	3*3=9	3*4=12	3*5=15	3*6=18	3*7=21	3*8=24	3*9=27
4*1=4	4*2=8	4*3=12	4*4=16	4*5=20	4*6=24	4*7=28	4*8=32	4*9=36
5*1=5	5*2=10	5*3=15	5*4=20	5*5=25	5*6=30	5*7=35	5*8=40	5*9=45
6*1=6	6*2=12	6*3=18	6*4=24	6*5=30	6*6=36	6*7=42	6*8=48	6*9=54
7*1=7	7*2=14	7*3=21	7*4=28	7*5=35	7*6=42	7*7=49	7*8=56	7*9=63
8*1=8	8*2=16	8*3=24	8*4=32	8*5=40	8*6=48	8*7=56	8*8=64	8*9=72
9*1=9	9*2=18	9*3=27	9*4=36	9*5=45	9*6=54	9*7=63	9*8=72	9*9=81

```
for i in range(1, 10):
    for j in range(1, 10):
        print("%d*%d=%d\t" % (i, j, i * j), end='')
    print()
```

打印图案

```
*
**
***
****
*****
```

```
for i in range(1, 6):
    for j in range(1, i + 1):
        print("*", end='')
    print()
```

3.3 break or continue?

3.3.1 break

break 可用于跳出当前的 switch 或循环结构。在一些情况下，在循环的中途已经完成了某个目标，没有必要再进行剩余的循环，这时就可以使用 break 跳出循环。

例如在判断一个数 n 是否为素数时，利用循环逐个判断 $2 \sim n-1$ 之间的数是否能整除 n。只要发现其中有一个数能整除 n，就证明 n 不是素数，可以跳出循环，不必再进行剩余的检查。

素数

```
1  import math
2
3  n = int(input("Enter an integer: "))
4
5  is_prime = True
6  for i in range(2, int(math.sqrt(n)) + 1):
7      if n % i == 0:
8          is_prime = False
9          break
10
11 if is_prime:
12     print(n, "is a prime number")
13 else:
14     print(n, "is not a prime number")
```

运行结果

```
Enter an integer: 17
17 is a prime number
```

3.3.2 continue

continue 与 break 使用方法类似，但是它并不是跳出循环，而是跳过本轮循环，直接开始下一轮循环。

正数平方和

```
n = 10
print("Enter %d integers: " % n)

sum_square = 0
for i in range(n):
    num = int(input())
    if num < 0:
        continue
    sum_square += num * num

print("Sum of squares of positive integers:", sum_square)
```

运行结果

```
Enter 10 integers:
5
7
-2
0
4
-4
-9
3
9
5
Sum of squares of positive integers: 205
```

Chapter 4 数据结构

4.1 序列

4.1.1 序列

Python 中序列类型包含字符串、列表、元组、字典。序列的核心意义在于可以进行多个数据的保存。

Python 在整体设计的过程之中强调的是简单化。以多数据的存储为例，在许多的编程语言里面，都是利用了数组实现了数据的存储，但是数据有一个最大的问题在于：长度是固定的，无法进行容量的扩充。

在这样的背景下，很多的开发者就不得不去独立地进行一些动态数组的开发，所以就有了数据结构的概念，而数据存储的内容多了，就需要提升数据的操作性能，C、C++、Java 等等都有这样的问题。

Python 在设计的时候充分地考虑到了这些动态性的设计问题，所以才将这些可能动态修改的内容统一地称为序列，也就是说 Python 中的序列就是一个动态（或静态）的存储，而这些存储的操作结构都是内置的，开发者可以避免数据结构的开发所造成的困难，尤其是面对与非科班的人而言。

列表是对传统数组的一种使用包装，但是与传统数组最大的不同在于，Python 中的列表的内容是允许进行动态修改的，并且 Python 中的列表也可以像传统数组那样机型索引的访问，这样就使得时间复杂度降低了许多。

4.2 列表

4.2.1 列表（List）

列表使用【[]】进行列表的定义，Python 的列表是对于传统数组的更高级的实现。列表可以通过索引的形式进行访问，索引下标是从 0 开始的，到列表长度 - 1 结束。

在使用列表进行数据存储的时候，虽然大部分的情况下都会使用相同的数据类型，但是在列表里面却可以同时有不同的数据类型，这比其它语言里面提供的数据的功能强大。虽然提供有多个数据类型的保存使得程序存储更加灵活，但是从实际的开发角度而言，尽量让数据类型保持一致会比较合理。

在 Python 中列表除了正向索引访问之外，也可以进行反向索引访问。

列表

```
lst = [1, 2, 3]

print(lst[0])
print(lst[1])
print(lst[2])

print(lst[-1])
print(lst[-2])
print(lst[-3])

print(lst[3])       # 越界
```

运行结果

```
1
2
3
3
2
1
IndexError: list index out of range
```

序列数据可以直接使用【*】进行重复定义，或者使用【+】进行与其它序列拼接。

序列重复/拼接

```
lst = [1, 2, 3] * 3
print(lst)      # [1, 2, 3, 1, 2, 3, 1, 2, 3]

lst = [1, 2, 3] + [4, 5, 6]
print(lst)      # [1, 2, 3, 4, 5, 6]
```

运行结果

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

4.2.2 成员运算符

在使用索引访问的时候都需要去考虑索引的设置错误的问题，但是使用了 for 循环之后，不再需要明确的进行索引的设置，避免了 IndexError 异常信息。

如果想要判断某一个数据是否在列表之中存在，最直接的方式就是直接进行列表的迭代操作（其它语言传统形式）。

迭代查找

```
lst = ["C/C++", "Java", "Python", "JavaScript"]
key = "Python"      # 待查找关键词
flag = False        # 初始假设未找到

for item in lst:
    if item == key:
        flag = True
        break

if flag:
    print("数据存在")
else:
    print("数据不存在")
```

运行结果

```
数据存在
```

这种查询最大的问题在于需要进行整个列表数据的迭代，造成的结果就是时间复杂度的攀升 $O(n)$。在 Python 设计过程之中不希望开发者为这些复杂的执行效率犯愁，所以在使用列表判断某些数据是否存在的时候提供了专门的运算符。使用关键字 in（在范围内）、not in（不在范围内）。

成员运算符

```
lst = ["C/C++", "Java", "Python", "JavaScript"]
key = "Python"      # 待查找关键词

if key in lst:
    print("数据存在")
else:
    print("数据不存在")
```

运行结果

数据存在

4.2.3 切片（Slicing）

切片可以截取部分的列表内容。在默认情况下，数据的分片都是依照数值为 1 的间隔进行获取的，如果有需要也可以进行步长的修改。

切片

```
lst = list(range(10))

print("原始列表: ", end='')
print(lst)

print("截取[2:7]: ", end='')
print(lst[2:7])

print("截取[:5]: ", end='')
print(lst[:5])

print("截取[3:]: ", end='')
print(lst[3:])

print("截取[::2]: ", end='')
print(lst[::2])
```

运行结果

```
原始列表: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
截取[2:7]: [2, 3, 4, 5, 6]
截取[:5]: [0, 1, 2, 3, 4]
截取[3:]: [3, 4, 5, 6, 7, 8, 9]
截取[::2]: [0, 2, 4, 6, 8]
```

4.2.4 列表操作函数

Python 中的列表设计非常到位，它可以实现内容动态扩充，可以进行后期的追加或者删除。

函数	功能
append(data)	在列表最后追加新内容
clear()	清除列表数据
copy()	列表拷贝
count()	统计某一个数据在列表中的出现次数
extend(列表)	为一个列表追加另外一个列表
index(data)	从列表查询某个值第一次出现的位置
insert(index, data)	向列表中指定索引位置追加新数据
pop(index)	从列表弹出并删除一个数据
remove(data)	从列表删除数据
reverse()	列表数据反转
sort()	列表数据排序

表 4.1: 列表操作函数

列表内容扩充

```
1 lst = list(range(5))    # [0, 1, 2, 3, 4]
2 lst.append(5)           # [0, 1, 2, 3, 4, 5]
```

```
lst.insert(2, 6)        # [0, 1, 6, 2, 3, 4, 5]
print(lst)
print("列表长度: %d" % len(lst))
```

运行结果

```
[0, 1, 6, 2, 3, 4, 5]
列表长度: 7
```

在使用 remove() 删除的时候，内容存在可以删除，不存在就会抛出 ValueError 异常信息，因此使用 remove() 操作之前一定要通过 in 判断。

如果不知道内容要进行数据的删除，最简单的原始的 Python 支持可以采用 del 关键字实现内容删除，而且使用 del 的时候只需要知道列表数据的索引即可实现。使用 del 关键字的确可以实现索引的删除，但是无法知道被删除了哪些数据。

pop() 可以根据索引删除，而后告诉用户哪些数据被删除了。

删除元素

```
lst = list(range(5))    # [0, 1, 2, 3, 4]
key = 3

if key in lst:
    lst.remove(key)
print(lst)              # [0, 1, 2, 4]

del lst[2]
print(lst)              # [0, 1, 4]

print(lst.pop(1))       # 1
print(lst)              # [0, 4]
```

运行结果

```
[0, 1, 2, 4]
[0, 1, 4]
1
[0, 4]
```

列表反转/排序

```
lst = [7, 4, 0, 3, 6, 8, 9, 2]

print("原始列表: ", end='')
print(lst)

print("反转: ", end='')
lst.reverse()
print(lst)

print("排序（升序）: ", end='')
lst.sort()
print(lst)

print("排序（降序）: ", end='')
lst.sort(reverse=True)
print(lst)
```

运行结果

```
原始列表: [7, 4, 0, 3, 6, 8, 9, 2]
反转: [2, 9, 8, 6, 3, 0, 4, 7]
排序（升序）: [0, 2, 3, 4, 6, 7, 8, 9]
排序（降序）: [9, 8, 7, 6, 4, 3, 2, 0]
```

4.3 元组

4.3.1 元组（Tuple）

列表是可以进行多个数据的保存，并且里面的内容都是可以修改和动态扩充的。与列表不一样，Python 提供了一个不可变的数据的序列——元组，元组需要使用【()】进行定义。元组定义的时候如果只有一个内容，必须要有【,】。

元组

```
1 tup = (0, 1, 2, 3, 4)
2 print(tup)
3 tup[0] = 5
```

运行结果

```
(0, 1, 2, 3, 4)
TypeError: 'tuple' object does not support item assignment
```

tuple() 可以实现列表与元组的转换。

元组与列表的转换

```
1 lst = list(range(5))
2 print(lst)
3 print(type(lst))
4
5 tup = tuple(lst)
6 print(tup)
7 print(type(tup))
```

运行结果

```
[0, 1, 2, 3, 4]
<class 'list'>
(0, 1, 2, 3, 4)
<class 'tuple'>
```

4.3.2 序列统计函数

函数	功能
len()	获取序列的长度
max()	获取序列中的最大值
min()	获取序列中的最小值
sum()	计算序列中的内容总和
any()	序列中有一个为 True 结果为 True，否则为 False
all()	序列中有一个为 False 结果为 False，否则为 True

表 4.2: 序列统计函数

序列统计函数

```
1  lst = [4, 0, 1, 3, 2]
2  tup = (8, 5, 7, 9, 6)
3  str = "HelloWorld"
4
5  print("列表lst的长度 = %d" % len(lst))
6  print("元组tup的长度 = %d" % len(tup))
7  print("字符串str的长度 = %d" % len(str))
8
9  print("列表lst最大值: %d" % max(lst))
10 print("元组tup最小值 = %d" % min(tup))
11 print("字符串str最大值 = %c" % max(str))
```

```
12 print("字符串str最小值 = %c" % min(str))
13
14 print("列表lst总和: %d" % sum(lst))
15 print("元组tup总和: %d" % sum(tup))
```

运行结果

```
列表lst的长度 = 5
元组tup的长度 = 5
字符串str的长度 = 10
列表lst最大值: 4
元组tup最小值 = 5
字符串str最大值 = r
字符串str最小值 = H
列表lst总和: 10
元组tup总和: 35
```

4.4 字符串

4.4.1 字符串（String）

字符串是由若干个字符所组成的，使用引号进行一串内容的声明。字符串也可以进行数据的分片操作和序列统计函数。

字符串

```
1 info = "url: www.baidu.com"
2
3 print("字符串长度: %d" % len(info))
4
5 if "url: " in info:
6     print(info[5:])
```

运行结果

```
字符串长度: 18
www.baidu.com
```

4.4.2 字符串格式化

在 Python 默认的语言环境中有字符串格式化的处理支持。为了进一步完善对于字符串格式化的需求，又提供了 format()。在 format() 里面出现的【{}】标记里面可以定义标记名称或者数据的填充序列顺序定义。

字符串格式化

```
1 name = "小灰"
2 age = 16
3 height = 175.6
```

```
4
5 info = "姓名：%s，年龄：%d，身高：%.2f" % (name, age, height)
6 print(info)
7
8 info = "姓名：{}，年龄：{}，身高：{}".format(name, age, height)
9 print(info)
```

运行结果

```
姓名：小灰，年龄：16，身高：175.60
姓名：小灰，年龄：16，身高：175.6
```

4.4.3 字符串数据处理函数

函数	功能
center()	字符串居中显示
find(data)	查找到内容返回索引位置，找不到返回-1
join(data)	字符串连接
split(data [, limit])	字符串拆分
lower()	字符串转小写
upper()	字符串转大写
capitalize()	首字母大写
replace(old, new [, limit])	字符串替换
strip()	删除左右空格

表 4.3: 字符串数据处理函数

字符串大小写转换

字符串大小写转换

```
1 str = "Hello World!"
```

```
2 print(str.upper())
3 print(str.lower())
```

运行结果

```
HELLO WORLD!
hello world!
```

字符串查找 find()

在进行数据内容查询的时候，如果内容存在就返回索引位置，不存在则返回-1。

字符串查找

```
1 str = "Hello World!"
2 print(str.find("Wor"))
3 print(str.find("Python"))
```

运行结果

```
6
-1
```

字符串连接 join()

使用一个特定的字符串连接序列中的若干内容。

字符串连接

```
1 lst = ["www", "baidu", "com"]
2 url = ".".join(lst)
3 print(url)
```

运行结果

```
www.baidu.com
```

字符串拆分 split()

字符串拆分

```
ip = "127.0.0.1"
print("ip信息: %s" % ip.split("."))

date_time = "2021/03/31 17:32:53"
item = date_time.split(" ")
print("date信息: %s" % item[0].split("/"))
print("time信息: %s" % item[1].split(":"))
```

运行结果

```
ip信息: ['127', '0', '0', '1']
date信息: ['2021', '03', '31']
time信息: ['17', '32', '53']
```

字符串替换 replace()

使用一个字符串去替换其它的字符串，在进行字符串替换的时候可以设置替换的次数。replace() 的替换是进行全匹配的替换操作。

字符串替换

```
str = "Hello World, Hello Python!"
print("全部替换: " + str.replace("Hello", "Hey"))
print("部分替换: " + str.replace("Hello", "Hey", 1))
```

运行结果

```
全部替换: Hey World, Hey Python!
部分替换: Hey World, Hello Python!
```

字符串去除前后空格 strip()

在用户键盘数据输入的过程中，有可能用户在输入信息时会添加无用的空格，导致数据判断错误。对于用户输入数据需要进行左右空格的删除。

模拟登陆

```
while True:
    info = input("Username:Password >>> ")
    if len(info) == 0 or info.find(":") == -1:
        print("输入格式错误！ ")
    else:
        data = info.split(":")
        username = data[0]
        password = data[1]
        if username == "admin" and password == "123456":
            print("欢迎【%s】成功登录！ " % username)
            break
        else:
            print("用户名或密码错误！ ")
```

运行结果

```
Username:Password >>> hey:
用户名或密码错误!
Username:Password >>> admin
输入格式错误!
Username:Password >>> admin:123456
欢迎【admin】成功登录!
```

4.5 字典

4.5.1 字典（Dictionary）

字典是一个在开发之中极为重要的类型，字典提供了非常方便地数据内容查找操作。字典的本质就是一个数据查找的序列，与之前的列表或者元组不同的地方在于，其它的结构保存数据的目的都是为了输出使用，而字典是为了查询使用。

字典严格意义上来讲属于一种哈希表的结构，在哈希表进行数据存储的时候往往都需要一个哈希算法进行数据存储位置的计算。

字典是一个二元偶对象的集合，所以里面保存的数据都是成对的，所有的数据内容都按照 key = value 的形式进行存放。考虑到用户使用的方便，key 的类型可以使数字、字符串或者是元组，但是最为常见的还是字符串。

在使用字典进行查询的时候如果指定的 key 不存在，则在代码执行会出现 KeyError 错误提示信息，也就是原生的数据查询方式是有可能会抛出异常的。

字典的本质是根据 key 查找对应的 value，一旦 key 重复的时候，使用新的数据覆盖掉旧的数据。

字典

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}
2 print(info)
3 print(info["age"])
4
5 info["height"] = 180
6 print(info)
```

运行结果

```
{'name': '小灰', 'age': 16, 'height': 175.6}
16
{'name': '小灰', 'age': 16, 'height': 180}
```

既然字典是一个序列，那么字典可以使用 in 进行判断指定 key 是否存在，通过这样的机制可以避免由于 key 不存在所带来的 KeyError 异常的抛出。

同样，字典也可以使用 for 循环迭代输出，但是迭代的知识字典中全部的 key。

字典迭代输出

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}
2 for key in info:
3     print("%s: %s" % (key, info[key]))
```

运行结果

```
name: 小灰
age: 16
height: 175.6
```

在进行字典数据迭代输出的时候，最好的做法是每一次迭代直接返回当前完整的 key:value 的映射项，此时的处理就需要依赖 items() 来完成。

items() 迭代输出

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}
2 for key, value in info.items():
3     print("%s: %s" % (key, value))
```

运行结果

```
name: 小灰
age: 16
height: 175.6
```

4.5.2 字典操作函数

函数	功能
clear()	清空字典数据
update({k:v, ...})	更新字典数据
get(key[, defaultvalue])	根据 key 获取数据
pop(key)	弹出字典中指定的 key 数据
popitem()	从字典中弹出一组映射项
keys()	返回字典中全部 key 数据
values()	返回字典中全部的 value 数据

表 4.4: 字典操作函数

字典数据更新 update()

Python 中的字典是可以进行存储数据的动态扩充的，update() 除了拥有表面上的更新之外，也拥有数据的扩充操作。

字典数据更新 update()

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}
2 info.update({"city": "Shanghai", "job": "programmer"})
3 print(info)
```

运行结果

```
{'name': '小灰', 'age': 16, 'height': 175.6,
'city': 'Shanghai', 'job': 'programmer'}
```

字典数据删除 pop()

pop() 可以根据 key 进行弹出操作。

字典数据删除

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}
2 print(info.pop("age"))
3 print(info)
```

运行结果

```
16
{'name': '小灰', 'height': 175.6}
```

字典数据获取 get()

可以通过 key 获取数据，如果 key 不存在则返回 None。

字典数据获取

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}
2 print(info.get("name"))
3 print(info.get("job"))
```

运行结果

```
小灰
None
```

Chapter 5 函数

5.1 函数

5.1.1 函数（Function）

函数执行一个特定的任务，Python 提供了大量内置函数，例如 print() 用来输出字符串、len() 用来计算序列长度等。

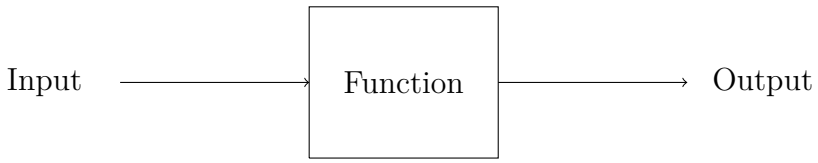


图 5.1: 函数

当调用函数时，程序控制权会转移给被调用的函数，当函数执行结束后，函数会把程序序控制权交还给其调用者。

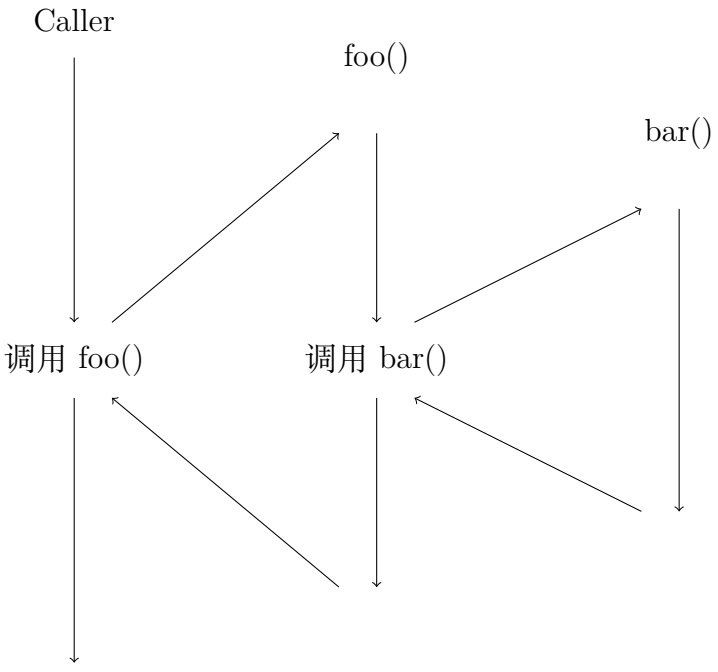


图 5.2: 函数调用

使用 def 关键字可以定义函数：

```
def func_name([param_list]):
    # code
```

5.1.2 函数设计方法

为什么不把所有的代码全部写在一起，还需要自定义函数呢?

使用函数有以下好处:

1. 避免代码复制，代码复制是程序质量不良的表现
2. 便于代码维护
3. 避免重复造轮子，提高开发效率

在设计函数的时候需要考虑以下的几点要素:

1. 确定函数的功能
2. 确定函数的参数
    - 是否需要参数
    - 参数个数
    - 参数类型
3. 确定函数的返回值
    - 是否需要返回值
    - 返回值类型

函数实现返回最大值

```
def get_max(num1, num2):
    # if num1 > num2:
    #     return num1
    # else:
```

```
    #     return num2

    return num1 if num1 > num2 else num2

print(get_max(4, 12))
print(get_max(54, 33))
print(get_max(0, -12))
print(get_max(-999, -774))
```

运行结果

```
12
54
0
-774
```

函数实现累加和

```
def get_sum(start, end):
    total = 0
    for i in range(start, end+1):
        total += i
    return total

print("1-100的累加和 = %d" % get_sum(1, 100))
print("1024-2048的累加和 = %d" % get_sum(1024, 2048))
```

运行结果

```
1-100的累加和 = 5050
1024-2048的累加和 = 1574400
```

函数实现输出 i 行 j 列由自定义字符组成的图案

```python
def print_chars(row, col, c):
    for i in range(row):
        for j in range(col):
            print(c, end='')
        print()

print_chars(5, 10, '?')
```

运行结果

```
??????????
??????????
??????????
??????????
??????????
```

5.2 主函数

5.2.1 主函数

Python 是为数不多的直接定义完源代码就可以执行的编程语言，很多编程语言对于程序的执行都有非常严格的标准，例如 C、C++、Java 等都有主函数（主方法）来标记程序的起点。

在现实的开发之中主函数是很有必要的，可以区分出其它的结构，在模块中更需要主函数的使用。

在 Python 中如果想实现主函数的定义，必须借助于全局变量 ___name___ 的返回内容，而后采用自定义的函数形式实现，返回的 ___main___ 是一个字符串，这个内容是会改变的，跟程序身处的结构有关。

在很多的编程语言都将主函数通过 main 这个标识符来定义，所以定义的 main() 是符合一般的习惯的。在进行复杂开发的时候，强烈建议使用主函数作为程序的起点。

主函数

```
1 def main():
2     pass
3
4 if __name__ == "__main__":
5     main()
```

5.3 变量作用域

5.3.1 变量作用域

一个变量会根据其自身所处的位置由不同的作用范围，不同范围内使用的变量采用就近取用的原则。

Python 的变量作用域采用 LEGB 原则：

- Local：函数内部变量名称。
- Enclosing Function Locals：外部嵌套函数变量名称。
- Global：函数所在模块或程序文件的变量名称。
- Builtin：内置模块的变量名称。

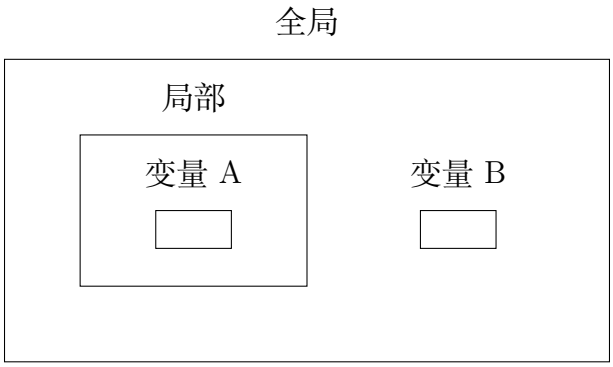


图 5.3: 变量作用域

在一个 Python 源文件之中可以存在有若干个函数（或者类），每一个函数里面有可能定义自己的变量，但是这个变量不是公共的。局部变量只能被一个函数所使用的，其它的函数都无法进行操作。

局部变量

```
1 def foo():
2     num = 123
3
```

```
4 def main():
5     print(num)
6
7 if __name__ == "__main__":
8     main()
```

运行结果

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

如果希望在函数中访问全局的变量，可以通过 global 关键字实现。

全局变量

```
1  num = 123      # 全局变量
2
3  def foo():
4      num = 321   # 局部变量
5      print("foo(): %d" % num)    # 321
6
7  def bar():
8      global num  # 调用全局变量
9      num = 456
10     print("bar(): %d" % num)    # 456
11
12 def main():
13     foo()
14     print(num)  # 123
15     bar()
16     print(num)  # 456
17
18 if __name__ == "__main__":
19     main()
```

运行结果

```
foo(): 321
123
bar(): 456
456
```

5.4 函数参数

5.4.1 参数默认值

在进行函数参数定义的时候，也可以设置默认值。当参数没有传递的时候就利用默认值来进行参数内容的填充，如果在参数上定义了默认值，那么该参数一定要放在参数列表的最后。

参数默认值

```
1  def set_date(year=1970, month=1, day=1):
2      print("%04d-%02d-%02d" % (year, month, day))
3
4  def main():
5      set_date(2021, 4, 1)
6      set_date(2021, 3)
7      set_date(2021)
8      set_date()
9
10 if __name__ == "__main__":
11     main()
```

运行结果

```
2021-04-01
2021-03-01
2021-01-01
1970-01-01
```

5.4.2 可变参数

在 Python 中提供有可变参数形式，所有可变参数都使用元组进行接收。

可变参数

```python
def calculate(operator, *nums):
    """
        定义一个可变参数的数学计算
        通过传入的运算符和运算数进行计算
        Args:
            operator (str): 运算符
            nums (tuple): 运算数（可变参数）
        Return:
            计算结果
    """
    if operator == '+':
        result = 0
        for num in nums:
            result += num
    elif operator == '*':
        result = 1
        for num in nums:
            result *= num
    return result

def main():
    print("累加: %d" % calculate('+', 1, 2, 3, 4, 5))
    print("阶乘: %d" % calculate('*', 1, 2, 3, 4, 5))

if __name__ == "__main__":
    main()
```

运行结果

```
累加: 15
阶乘: 120
```

在进行参数传递的时候也可以使用【**】来标记关键字参数（字典）。

关键字参数

```
def print_scores(name, **scores):
    print(name)
    for key, value in scores.items():
        print("\t|- %s: %s" % (key, value))

def main():
    print_scores("小灰", Python=100, Java=95)
    print_scores("小白", 数据结构=78, 算法=82)

if __name__ == "__main__":
    main()
```

运行结果

```
小灰
    |- Python: 100
    |- Java: 95
小白
    |- 数据结构: 78
    |- 算法: 82
```

5.5 lambda 表达式

5.5.1 lambda 表达式

lambda 指的是函数式编程，函数式编程最简单的理解就是没有名字的函数。lambda 定义简单，只使用一次。

```
lambda param1, param2, ...: statement
```

一般而言，lambda 函数都比较短，但是一般的普通函数内容都会比较多。

lambda 函数

```
def main():
    add = lambda x, y: x + y
    print(add(10, 20))

if __name__ == "__main__":
    main()
```

运行结果

```
30
```

5.6 递归

5.6.1 递归（Recursion）

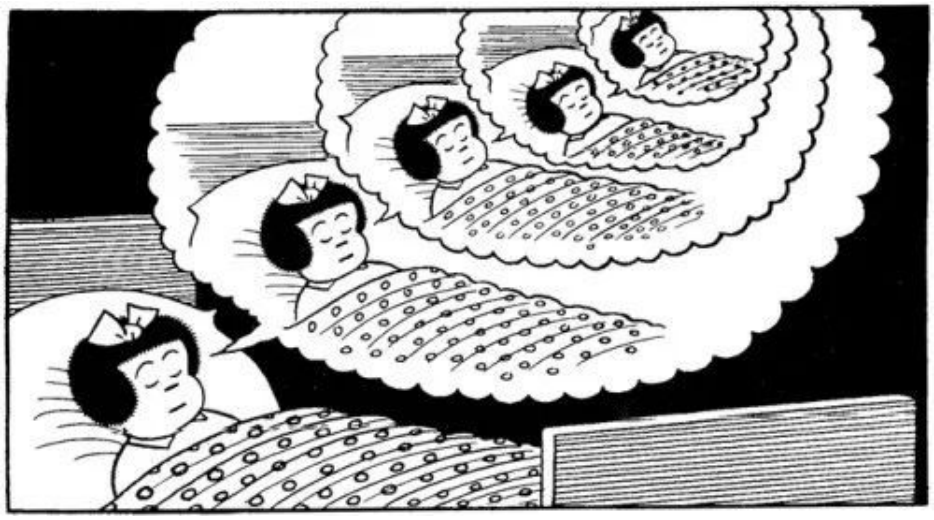
要理解递归，先得理解递归（见5.6章节）。

在函数的内部，直接或者间接的调用自己的过程就叫作递归。对于一些问题，使用递归可以简洁易懂的解决问题，但是递归的缺点是性能低，占用大量系统栈空间。

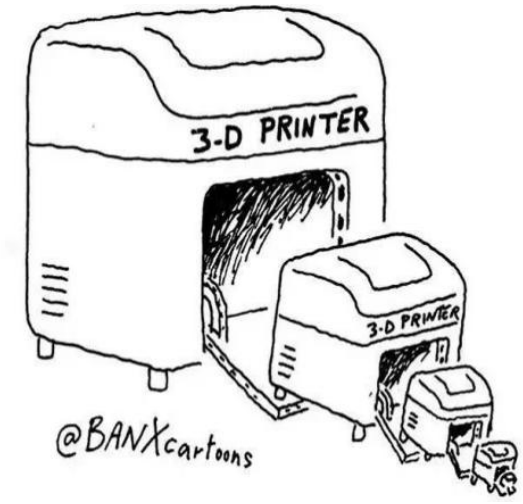
递归算法很多时候可以处理一些特别复杂、难以直接解决的问题。例如：

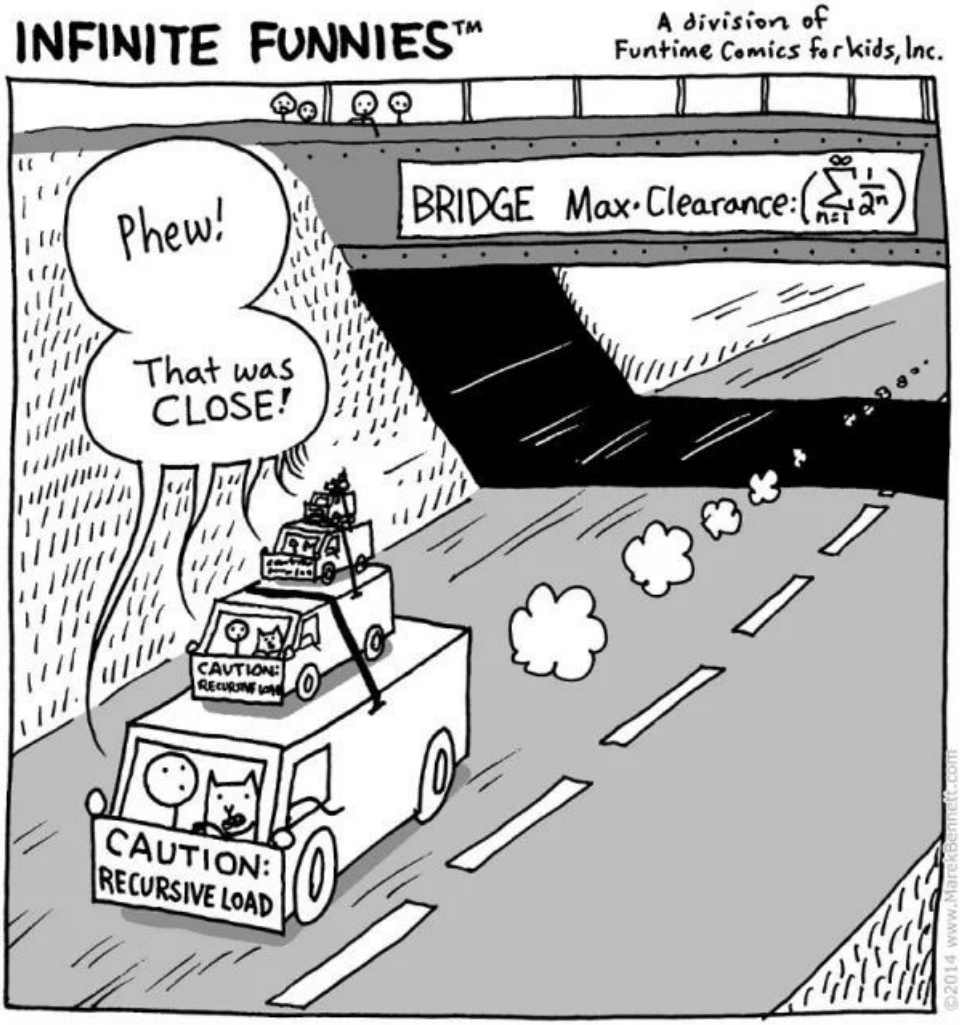
- 迷宫
- 汉诺塔
- 八皇后
- 排序
- 搜索

在定义递归函数时，一定要确定一个结束条件，否则会造成无限递归的情况，最终会导致栈溢出。




CN
COLLECTION


3-D PRINTER
3-D PRINTER
@BANXcartoons



无限递归

```python
def tell_story():
    print("从前有座山")
    print("山里有座庙")
    print("庙里有个老和尚和小和尚")
    print("老和尚在对小和尚讲故事")
    print("他讲的故事是：")
    tell_story()

def main():
    tell_story()

if __name__ == "__main__":
    main()
```

运行结果

```
从前有座山
山里有座庙
庙里有个老和尚和小和尚
老和尚对小和尚在讲故事
他讲的故事是:
从前有座山
山里有座庙
庙里有个老和尚和小和尚
老和尚对小和尚在讲故事
他讲的故事是:
...
```

递归函数一般需要定义递归的出口，即结束条件，确保递归能够在适合的地方退出。

阶乘

```
1  def factorial(n):
2      if n == 0 or n == 1:
3          return 1
4      return n * factorial(n-1)
5
6  def main():
7      print("5! = %d" % factorial(5))
8
9  if __name__ == "__main__":
10     main()
```

运行结果

```
5! = 120
```

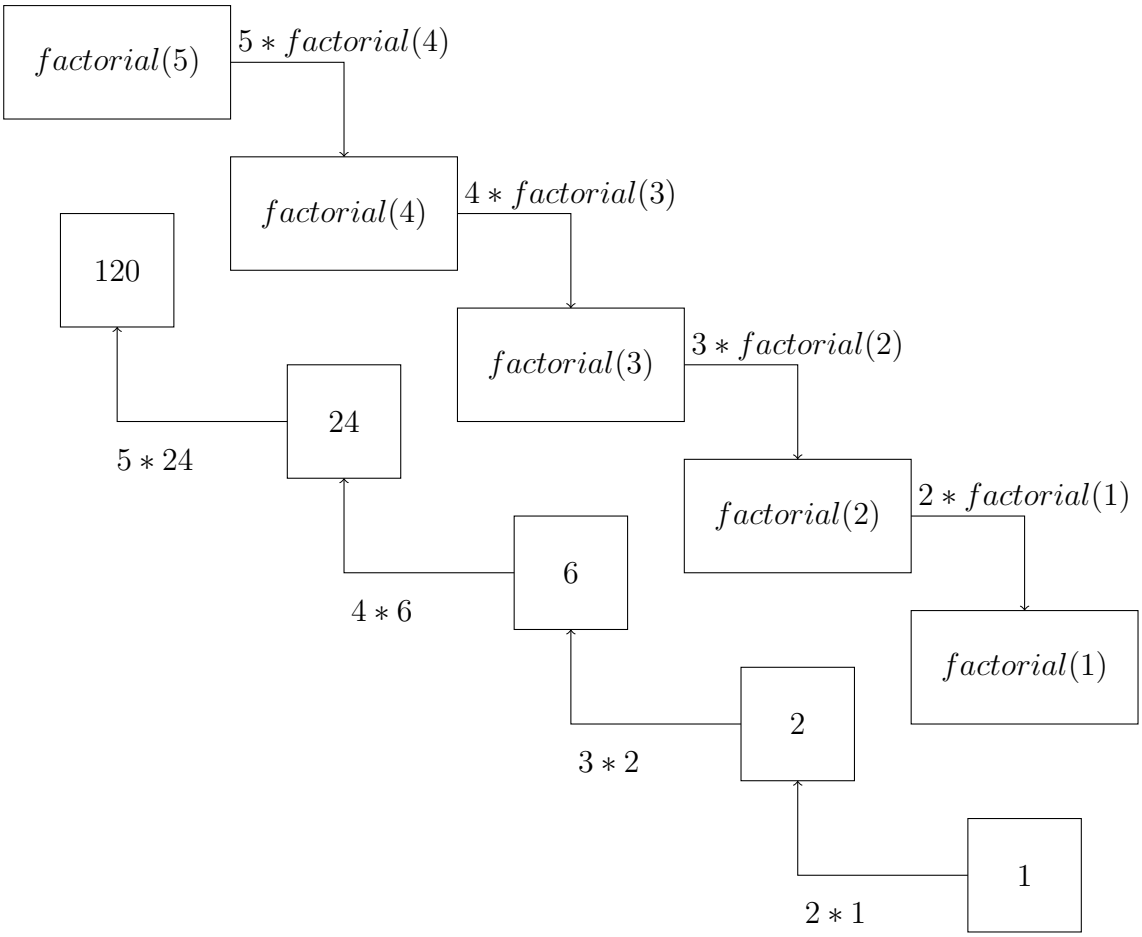


图 5.4: 阶乘

斐波那契数列（递归）

```
1  def fibonacci(n):
2      if n == 1 or n == 2:
3          return 1
4      return fibonacci(n-2) + fibonacci(n-1)
5
6  def main():
7      n = 7
8      print("斐波那契数列第%d位: %d" % (n, fibonacci(n)))
9
10 if __name__ == "__main__":
11     main()
```

运行结果

斐波那契数列第7位：13

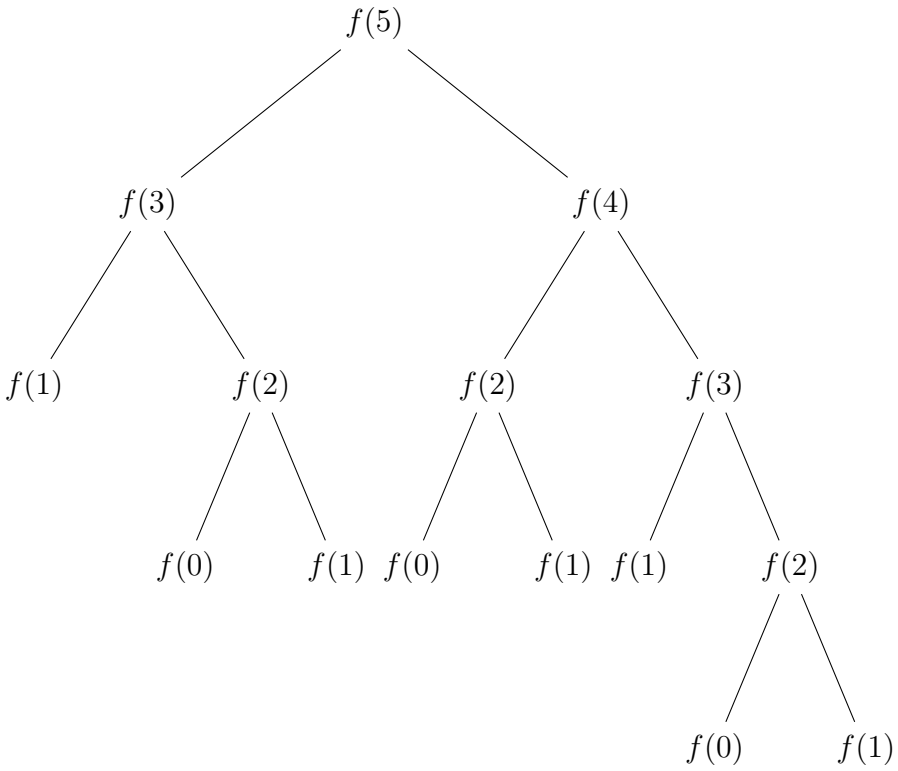


图 5.5: 递归树

斐波那契数列（迭代）

```
1  def fibonacci(n):
2      f = [0] * n
3      f[0] = f[1] = 1
4      for i in range(2, n):
5          f[i] = f[i-2] + f[i-1]
6      return f[n-1]
7
8  def main():
9      n = 7
10     print("斐波那契数列第%d位：%d" % (n, fibonacci(n)))
11
```

```
12 if __name__ == "__main__":
13     main()
```

运行结果

```
斐波那契数列第7位： 13
```

阿克曼函数

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0, n = 0 \\ A(m-1, A(m,n-1)) & m > 0, n > 0 \end{cases}$$

```
1  def A(m, n):
2      if m == 0:
3          return n + 1
4      elif m > 0 and n == 0:
5          return A(m-1, 1)
6      else:
7          return A(m-1, A(m, n-1))
8
9  def main():
10     print(A(3, 4))
11
12 if __name__ == "__main__":
13     main()
```

运行结果

```
125
```

m \ n	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$2+(n+3)-3$
2	3	5	7	9	11	$2(n+3)-3$
3	5	13	29	61	125	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$A(3,2^{65536}-3)$	$A(3,A(4,3))$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3\ twos}-3$
5	65533	$A(4,65533)$	$A(4,A(5,1))$	$A(4,A(5,2))$	$A(4,A(5,3))$	...
6	$A(5,1)$	$A(5,A(5,1))$	$A(5,A(6,1))$	$A(5,A(6,2))$	$A(5,A(6,3))$	...

表 5.1: 阿克曼函数

汉诺塔

给定三根柱子，其中 A 柱子从大到小套有 n 个圆盘，问题是如何借助 B 柱子，将圆盘从 A 搬到 C。

规则：

- 一次只能搬动一个圆盘
- 不能将大圆盘放在小圆盘上面

递归算法求解汉诺塔问题：

1. 将前 n-1 个圆盘从 A 柱借助于 C 柱搬到 B 柱。
2. 将最后一个圆盘直接从 A 柱搬到 C 柱。
3. 将 n-1 个圆盘从 B 柱借助于 A 柱搬到 C 柱。

```
1  move = 0            # 移动次数
2
3  def hanoi(n, src, mid, dst):
4      global move
5      """
6          汉诺塔算法
7          把 n 个盘子从 src 借助 mid 移到 dst
8          Args:
9              n (int): 层数
10             src (str): 起点柱子
11             mid (str): 临时柱子
12             dst (str): 目标柱子
13     """
14     if n == 1:
15         print("%d号盘: %c -> %c" % (n, src, dst))
16         move += 1
17     else:
18         # 把前 n-1 个盘子从 src 借助 dst 移到 mid
```

```
19          hanoi(n-1, src, dst, mid)
20          # 移动第 n 个盘子
21          print("%d号盘: %c -> %c" % (n, src, dst))
22          move += 1
23          # 把刚才的 n-1 个盘子从 mid 借助 src 移到 dst
24          hanoi(n-1, mid, src, dst)
25
26  def main():
27      hanoi(4, 'A', 'B', 'C')
28      print("步数 ==> %d" % move)
29
30  if __name__ == "__main__":
31      main()
```

运行结果

```
1号盘: A -> B
2号盘: A -> C
1号盘: B -> C
3号盘: A -> B
1号盘: C -> A
2号盘: C -> B
1号盘: A -> B
4号盘: A -> C
1号盘: B -> C
2号盘: B -> A
1号盘: C -> A
3号盘: B -> C
1号盘: A -> B
2号盘: A -> C
1号盘: B -> C
步数 ==> 15
```

Chapter 6 模块

6.1 模块导入

6.1.1 模块（Module）

模块是进行大型项目拆分组织的一种有效技术手段，它可以将一个庞大的代码分割成若干个小的组成单元，方便进行代码的开发与维护。利用模块的划分，在进行代码维护的时候，可以保证局部的更新不影响其它的程序的运行操作。

使用 import 关键字可以进行模块的导入，import 可以同时导入多个模块，但是从开发的角度来讲，强烈建议分开导入。

```
1 import package.module [as alias] [, ...]
```

模块导入

algorithm/search.py

```
1  def sequence_search(list, key):
2      """
3          顺序查找
4          Args:
5              list (list): 待查找数组
6              key (int): 关键字
7      """
8      for i in range(len(list)):
9          if list[i] == key:
10             return i
11     return -1
12
13 def binary_search(list, key):
14     """
15         二分查找
16         Args:
```

```
17              list (list): 待查找数组
18              key (int): 关键字
19      """
20      start = 0
21      end = len(list) - 1
22      while start <= end:
23          mid = (start + end) // 2
24          if list[mid] == key:
25              return mid
26          elif list[mid] < key:
27              start = mid + 1
28          else:
29              end = mid - 1
30      return -1
```

import_as.py

```
1  import algorithm.search as search
2
3  def main():
4      list = [40, 9, 20, 93, 7, 34, 85, 91]
5      key = 34
6      print("%d所在位置: %d" % (key, search.sequence_search(list, key)))
7
8  if __name__ == "__main__":
9      main()
```

运行结果

```
34所在位置: 5
```

在 Python 里有一个作者写的 Python 开发禅道（开发的 19 条哲学），如果想看到这个彩蛋的信息，可以在 Python 的交互模式输入 import this。

运行结果

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to
do it.
Although that way may not be obvious at first unless you're
Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

6.1.2 from-import 模块导入

使用 import 导入模块之后需要采用 module_name.func() 形式进行调用，每一次调用函数非常麻烦。from-import 导入语法可以简化调用语句。

```
from package.module import name [as alias] [, ...]
```

在实践中，from-import 不是良好的编程风格，因为如果导入的变量与作用域中现有变量同名，那么变量就会被悄悄覆盖掉。使用 import 语句的时候就不会发生这种问题，通过 module.var 或 module.func() 获取的变量或方法不会与现有作用域冲突。

from-import 导入

algorithm/search.py

```
def sequence_search(list, key):
    """
        顺序查找
        Args:
            list (list): 待查找数组
            key (int): 关键字
    """
    for i in range(len(list)):
        if list[i] == key:
            return i
    return -1

def binary_search(list, key):
    """
        二分查找
        Args:
            list (list): 待查找数组
            key (int): 关键字
    """
    start = 0
    end = len(list) - 1
    while start <= end:
        mid = (start + end) // 2
        if list[mid] == key:
            return mid
        elif list[mid] < key:
            start = mid + 1
```

```
28          else:
29              end = mid - 1
30      return -1
```

from_import.py

```
1  from algorithm.search import binary_search
2
3  def main():
4      list = [7, 9, 20, 34, 40, 85, 91, 93]
5      key = 34
6      print("%d所在位置: %d" % (key, binary_search(list, key)))
7
8  if __name__ == "__main__":
9      main()
```

运行结果

```
34所在位置: 3
```

6.2 math 模块

6.2.1 math 模块

现代计算机的基础学科就是数学，如果没有数学理论作为基础，计算机是无法的得到正常发展的。数学模块只提供了数学的基本计算功能，在很多的开发之中，有可能会需要使用到更加复杂的数学逻辑的时候就需要采用一些第三方模块进行数学计算了。

math 模块

```
1  import math
2
3  def main():
4      print("累加：%d" % math.fsum(range(101)))
5      print("阶乘：%d" % math.factorial(10))
6      print("乘方：%d" % math.pow(2, 10))
7      print("对数：%f" % math.log(10))
8      print("余数：%d" % math.fmod(22, 5))
9
10 if __name__ == "__main__":
11     main()
```

运行结果

```
累加：5050
阶乘：3628800
乘方：1024
对数：2.302585
余数：2
```

6.3 random 模块

6.3.1 random 模块

随机数可以在一个指定的范围之内随机地生成一些数字供使用。例如，手机验证码发送来的数字就是使用随机数的方式生成的。

方法	功能
random()	生成一个 0 到 1 的随机浮点数：$0.0 \leq n \leq 1.0$
uniform(x, y)	生成一个在指定范围内的随机浮点数
randint(x, y)	生成一个指定范围内的随机整数 $x \leq n \leq y$
choice(sequence)	从序列中随机抽取数据
shuffle(x [, random])	将一个列表中的元素打乱
sample(sequence, k)	从指定序列中随机获取指定序列分片

表 6.1: random 模块

random 模块

```
1  import random
2
3  def main():
4      lst = [random.randint(1, 100) for _ in range(10)]
5      print("初始序列：%s" % lst)
6      print("随机抽取：", end='')
7      for _ in range(5):
8          print(random.choice(lst), end=' ')
9
10 if __name__ == "__main__":
11     main()
```

运行结果

```
初始序列：[85, 83, 83, 29, 2, 93, 30, 65, 41, 54]
随机抽取：41 93 2 54 93
```

6.4 time 模块

6.4.1 time 模块

time 模块是 Python 内置的一个实现时间的操作模块，用于描述日期时间的数据类型分为三种：

时间戳（timestamp）

从 1970 年 1 月 1 日 00 时 00 分 00 秒开始的按秒计算的时间偏移量。

计算操作耗时

```python
import time

def main():
    start = time.time()
    print("【开始】%s" % start)
    sum = 0
    for i in range(99999999):
        sum += i
    end = time.time()
    print("【结束】%s" % end)
    print("【耗时】%.2fs" % (end - start))

if __name__ == "__main__":
    main()
```

运行结果

```
【开始】1617327695.1819823
【结束】1617327701.1566718
【耗时】5.97s
```

时间元组

保存日期时间数据的元素结构对象。

属性	功能	数值
tm_year	年	yyyy
tm_mon	月	1 ~ 12
tm_mday	日	1 ~ 32
tm_hour	时	0 ~ 23
tm_min	分	0 ~ 59
tm_sec	秒	0 ~ 61（60 或 61 是闰秒）
tm_wday	一周第几天	0 ~ 6（0 表示周一）
tm_yday	一年第几天	1 ~ 366

表 6.2: 时间元组

时间戳与时间元组的转换

```
1 import time
2
3 def main():
4     current_time = time.time()
5     current_time_tuple = time.localtime(current_time)
6     print("时间戳转换时间元组： " + str(current_time_tuple))
7
8 if __name__ == "__main__":
9     main()
```

运行结果

```
时间戳转换时间元组： time.struct_time(tm_year=2021, tm_mon=4,
tm_mday=2, tm_hour=9, tm_min=45, tm_sec=16, tm_wday=4,
tm_yday=92, tm_isdst=0)
```

格式化日期时间

可以按照指定的标记进行格式化处理。时间戳与时间元组更多情况下还是描述程序层次上的概念，格式化日期时间可以给出人们都认可的显示格式。

格式化日期时间

```
1  import time
2
3  def main():
4      current_time = time.time()
5      current_time_tuple = time.localtime(current_time)
6
7      print(time.strftime("%Y-%m-%d %H:%M:%S", current_time_tuple))
8      print("date: %s" % time.strftime("%F", current_time_tuple))
9      print("time: %s" % time.strftime("%T", current_time_tuple))
10
11 if __name__ == "__main__":
12     main()
```

运行结果

```
2021-04-02 09:49:01
date: 2021-04-02
time: 09:49:01
```

6.5 copy 模块

6.5.1 copy 模块

copy 是一个专门进行内容复制的处理模块。拷贝分为浅拷贝（shallow copy）和深拷贝（deep copy）两种：

- 浅拷贝：只是复制第一层的内容，而更深入的数据嵌套关系不会拷贝。
- 深拷贝：会进行完整的复制。

6.5.2 引用

引用的本质在于将同一块内存空间，交给不同的对象进行同时操作，当一个对象修改了内存数据之后，其它对象的内存数据也会同时发生改变。

```
1 a = {1:[1, 2, 3]}
2 b = a
```

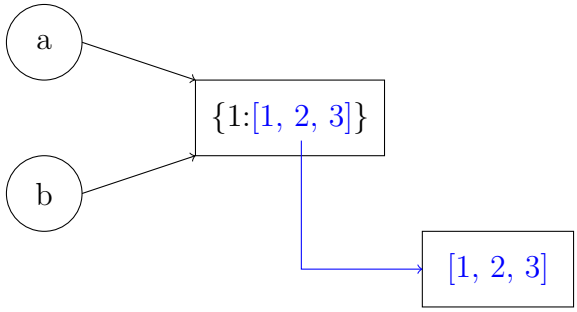


图 6.1: 引用

引用传递

```
1 def main():
2     info = dict(name="小灰", age=16, skills=["Python", "C/C++"])
3     copy_info = info        # 引用传递
4     copy_info["skills"].append("Java")
```

```
    print(info)

if __name__ == "__main__":
    main()
```

运行结果

```
{'name':'小灰', 'age':16, 'skills':['Python', 'C/C++', 'Java']}
```

6.5.3 浅拷贝

拷贝和引用传递是不同的，拷贝是将原始的内存的数据进行一份复制，而后为其分配单独的对象的指向。

```
a = {1:[1, 2, 3]}
b = a.copy()
```

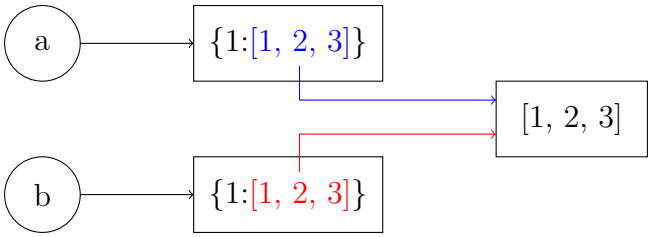


图 6.2: 浅拷贝

浅拷贝

```
import copy

def main():
    info = dict(name="小灰", age=16, skills=["Python", "C/C++"])
    copy_info = copy.copy(info)      # 浅拷贝
    copy_info.pop("age")
    copy_info["skills"].append("Java")
```

```
    print(info)
    print(copy_info)

if __name__ == "__main__":
    main()
```

运行结果

```
{'name':'小灰', 'age':16, 'skills':['Python', 'C/C++', 'Java']}
{'name':'小灰', 'skills':['Python', 'C/C++', 'Java']}
```

6.5.4 深拷贝

```
a = {1:[1, 2, 3]}
b = copy.deepcopy(a)
```

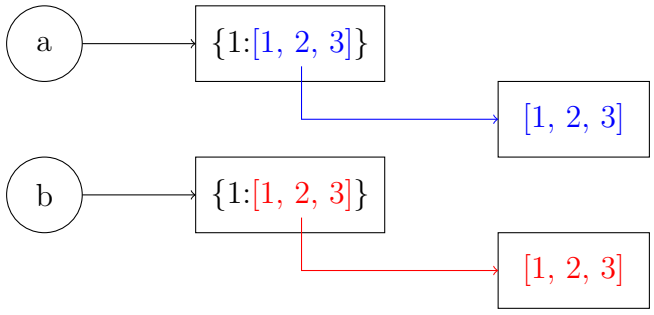


图 6.3: 深拷贝

深拷贝

```
import copy

def main():
    info = dict(name="小灰", age=16, skills=["Python", "C/C++"])
    copy_info = copy.deepcopy(info)      # 深拷贝
    copy_info.pop("age")
```

```
    copy_info["skills"].append("Java")
    print(info)
    print(copy_info)

if __name__ == "__main__":
    main()
```

运行结果

```
{'name': '小灰', 'age': 16, 'skills': ['Python', 'C/C++']}
{'name': '小灰', 'skills': ['Python', 'C/C++', 'Java']}
```

6.6 MapReduce 数据处理

6.6.1 MapReduce

Python 在数据分析领域上使用非常广泛，并且实现简单。在 Python 中可以进行大量数据的快速处理，进行数据的过滤、分析操作。在数据量小的情况下可以方便地使用 for 循环进行数据的逐个处理，但是在数据量大的情况下，就需要使用一些特定的处理函数进行过滤、分析、统计等操作。

在 Python 中默认提供有了 filter()、map()，但是要进行统计处理，则需要导入 reduce()。

函数	功能
filter(function, sequence)	对传入的序列数据进行过滤
map(function, sequence)	对传入的序列数据进行处理
reduce(function, sequence)	对传入的序列数据进行统计

表 6.3: MapReduce 数据处理函数

在进行数据处理的过程之中都需要有一个处理函数，这个处理函数就定义了数据该如何进行处理或统计，一般而言这样的函数都比较短，所以大部分情况下都可以利用 lambda 函数来完成。

MapReduce 数据处理

```
1  from functools import reduce
2
3  def main():
4      lst = list(range(10))
5
6      filter_lst = list(filter(lambda x: x % 2 ==0, lst))
7      print("过滤出偶数：%s" % filter_lst)
8
9      map_lst = list(map(lambda x: x ** 2, filter_lst))
10     print("平方：%s" % map_lst)
```

```
11
12      result = reduce(lambda x, y: x+y, map_lst)
13      print("求和：%d" % result)
14
15  if __name__ == "__main__":
16      main()
```

运行结果

```
过滤出偶数：[0, 2, 4, 6, 8]
平方：[0, 4, 16, 36, 64]
求和：120
```

6.7 pip 模块管理工具

6.7.1 pip

Python 本地有一些系统模块开发者可以直接进行使用，但是开发者仅仅是依靠系统模块是不够的，需要大量的使用第三方模块。为了解决这些模块的管理问题，在 Python 中内置了 pip 管理工具。通过此工具可以直接连接到 Python 远程服务模块仓库，通过仓库下载所需要的模块。

在 Python 安装的时候会自动进行 pip 工具的相关安装，输入 pip –help 查看 pip 的相关命令选项。

功能	**命令**
搜索模块	pip search 模块名
安装模块	pip install 模块名
查看已安装模块	pip list
列出过期模块	pip list –outdated
更新模块	pip install –upgrade 模块名
卸载模块	pip uninstall 模块名

表 6.4: pip 命令

6.8 jieba 分词

6.8.1 jieba

分词是一种数学的应用，它可以直接根据词语之间的数学关系进行文字或单词的抽象。例如对“中华人民共和国”进行分词处理，可以拆分为“中华”、“华人”、“人民”、“共和”、“共和国”、“中华人民共和国”。如果没有分词，就无法进行搜索引擎的开发。

jieba 是在中文自然语言处理中用得最多的工具包之一，它以分词起家，目前已经能够实现包括分词、词性标注以及命名实体识别等多种功能。

考虑到分词的效果与性能，在 jieba 组件中提供有 3 种分词模式：

1. 精确模式：将句子进行最精确的切分，分词速度相对较低。
2. 全模式：基于词汇列表将句子中所有可以成词的词语都扫描出来，该模式处理速度非常快，但是不能有效解决歧义的问题。
3. 搜索引擎模式：在精确模式的基础上，对长词进行再次切分，该模式适用于搜索引擎构建索引的分词。

统计《西游记》中出现次数最多的 20 个词语

```
import jieba

PATH = "西游记.txt"      # 文件路径

def main():
    word_frequence = {}      # 词频表

    # 打开文件
    with open(file=PATH, mode="r", encoding="UTF-8") as file:
        line = file.readline()  # 读取一行数据
        while line:
```

```
12              words = jieba.lcut(line)    # 分词
13              for word in words:
14                  if len(word) == 1:  # 舍弃长度为1的词
15                      continue
16                  else:
17                      # dict.get(key, default=None)
18                      word_frequence[word]
19                          = word_frequence.get(word, 0) + 1
20              line = file.readline()
21
22          # 获取所有数据项
23          items = list(word_frequence.items())
24          # 根据出现次数降序排序
25          items.sort(key=lambda x: x[1], reverse=True)
26
27          # 取前20项
28          for i in range(20):
29              word, count = items[i]
30              print("%s: %s" % (word, count))
31
32  if __name__ == "__main__":
33      main()
```

Chapter 7 面向对象

7.1 面向过程与面向对象

7.1.1 面向过程（Procedure Oriented）

面向过程是一种以过程为中心的编程思想，以什么正在发生为主要目标进行编程，分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用。

C 语言就是一种面向过程的编程语言，但是面向过程的缺陷是数据和函数并不完全独立，使用两个不同的实体表示信息及其操作。

7.1.2 面向对象（Object Oriented）

面向对象是相对于面向过程来讲的，面向对象方法把相关的数据和方法组织为一个整体来看待，从更高的层次来进行系统建模，更贴近事物的自然运行模式。

在面向对象中，把构成问题的事物分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

Java、C++、Python 等都是面向对象的编程语言，面向对象的优势在于只是用一个实体就能同时表示信息及其操作。

面向对象三大特性：

1. 封装（encapsulation）：数据和代码捆绑，避免外界干扰和不确定性访问。
2. 继承（inheritance）：让某种类型对象获得另一类型对象的属性和方法。
3. 多态（polymorphism）：同一事物表现出不同事物的能力。

7.2 类与对象

7.2.1 类与对象

类（class）表示同一类具有相同特征和行为的对象的集合，类定义了对象的属性和方法。

对象（object）是类的实例，对象拥有属性和方法。

类的设计需要使用关键字 class，类名是一个标识符，遵循大驼峰命名法。类中可以包含属性和方法。其中，属性通过变量表示，又称实例变量；方法用于描述行为，又称实例方法。

在程序之中如果需要使用类，那么一般都会通过对象来进行操作。

```
obj_name = class_name([param])
```

当实例化了一个对象之后，就可以通过此对象进行类中成员的访问：

- 对象. 属性：访问类中的属性内容，如果程序中访问了没有定义的实例属性，那么将引发 AttributeError 异常。
- 对象. 方法 ()：调用类中的方法。对于类中每一个方法的当前对象都会有 Python 自己来负责该对象的传入，这一操作不是由用户负责的。

类和对象

```
class Person:
    name = ""
    age = 0
    gender = ""

    def eat(self):
        print("吃饭")

```

```
9       def sleep(self):
10          print("睡觉")
11
12  def main():
13      person = Person()
14
15      person.name = "小灰"
16      person.age = 16
17      person.gender = "男"
18
19      print("姓名：%s，年龄：%d，性别：%s" % (
20              person.name, person.age, person.gender))
21      person.eat()
22      person.sleep()
23
24  if __name__ == "__main__":
25      main()
```

运行结果

```
姓名：小灰，年龄：16，性别：男
吃饭
睡觉
```

7.2.2 垃圾回收机制

引用传递的本质在于将同一块空间修改权力交由不同的对象来完成，在这样的处理之中就有可能产生垃圾空间。在 Python 的引用数据处理之中都会存在有一个引用计数器，当引用计数器为 0 的时候就表示该对象已经成为了垃圾，等待进行回收。

垃圾回收机制

```
class Person:
    pass

def main():
    person1 = Person()
    person2 = Person()

    print("【引用传递前地址】person1: %d, person2: %d" % (
            id(person1), id(person2)))
    person2 = person1
    print("【引用传递后地址】person1: %d, person2: %d" % (
            id(person1), id(person2)))

if __name__ == "__main__":
    main()
```

运行结果

```
【引用传递前地址】person1: 1988221852552, person2: 1988222686536
【引用传递后地址】person1: 1988221852552, person2: 1988221852552
```

在开发之中，实际上对于垃圾空间应该尽可能少的产生，虽然 Python 提供有垃圾收集机制，但是垃圾的回收与释放依然需要占用系统资源。

7.3 封装

7.3.1 封装（Encapsulation）

封装是面向对象方法的重要原则，就是把对象的属性和方法结合为一个独立的整体，并尽可能隐藏对象的内部实现细节。

封装可以认为是一个保护屏障，防止该类的数据被外部类随意访问。要访问该类的数据，必须通过严格的接口控制。合适的封装可以让代码更容易理解和维护，也加强了程序的安全性。

实现封装的步骤：

1. 修改属性的可见性来限制对属性的访问，一般限制为 private。
2. 对每个属性提供对外的公共方法访问，也就是提供一对 setter / getter，用于对私有属性的访问。

封装

```
class Person:
    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_age(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

def main():
    person = Person()
```

```
16      person.set_name("小灰")
17      person.set_age(17)
18      print("姓名：%s，年龄：%d" % (
19              person.get_name(), person.get_age()))
20
21  if __name__ == "__main__":
22      main()
```

运行结果

```
姓名：小灰，年龄：17
```

7.4 构造方法与析构方法

7.4.1 构造方法（Constructor）

构造方法也是一个方法，用于实例化对象，在实例化对象的时候调用。一般情况下，使用构造方法是为了在实例化对象的同时，给一些属性进行初始化赋值。

构造方法和普通方法的区别：

1. 构造方法的名称必须为 ___init___()。
2. 构造方法没有返回值。
3. 一个类中只允许定义最多 1 个构造方法。

如果一个类中没有写构造方法，系统会自动提供一个无参构造方法，以便实例化对象。

构造方法

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_info(self):
        return "姓名：%s，年龄：%d" % (self.__name, self.__age)

def main():
    person = Person("小灰", 17)
    print(person.get_info())

if __name__ == "__main__":
    main()
```

运行结果

```
姓名：小灰，年龄：17
```

7.4.2 析构方法（Destructor）

在对象实例化的时候会触发构造方法的执行，与之对应的操作称为析构，当对象不再使用的时候进行某些收尾处理的操作。析构方法名称也要要求，必须为__del__()。

析构方法

```
1  class Person:
2      def __init__(self):
3          print("构造方法被执行了")
4
5      def __del__(self):
6          print("析构方法被执行了")
7
8  def main():
9      person = Person()
10     del person
11
12 if __name__ == "__main__":
13     main()
```

运行结果

```
构造方法被执行了
析构方法被执行了
```

7.4.3 匿名对象

对象的名称只是一个地址的信息，真正的内容是保存在内存中的，如果不需要名称就可以使用匿名对象。匿名对象使用完成之后由于没有其它对象进行引用，那么就有可能被垃圾回收，同时也会调用析构方法。

如果一个对象要被反复使用，那么可以定义有名对象；如果一个对象只是用一次，就采用匿名对象完成操作即可。

匿名对象

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_info(self):
        return "姓名：%s，年龄：%d" % (self.__name, self.__age)

def main():
    print(Person("小灰", 17).get_info())

if __name__ == "__main__":
    main()
```

运行结果

```
姓名：小灰，年龄：17
```

7.5 继承

7.5.1 继承（Inheritance）

继承是面向对象的三大特征之一，程序中的继承是类与类之间的特征和行为的一种赠予或获取。两个类之间的继承必须满足“is a”的关系。子类继承自父类，父类也称基类或超类，子类也称派生类。

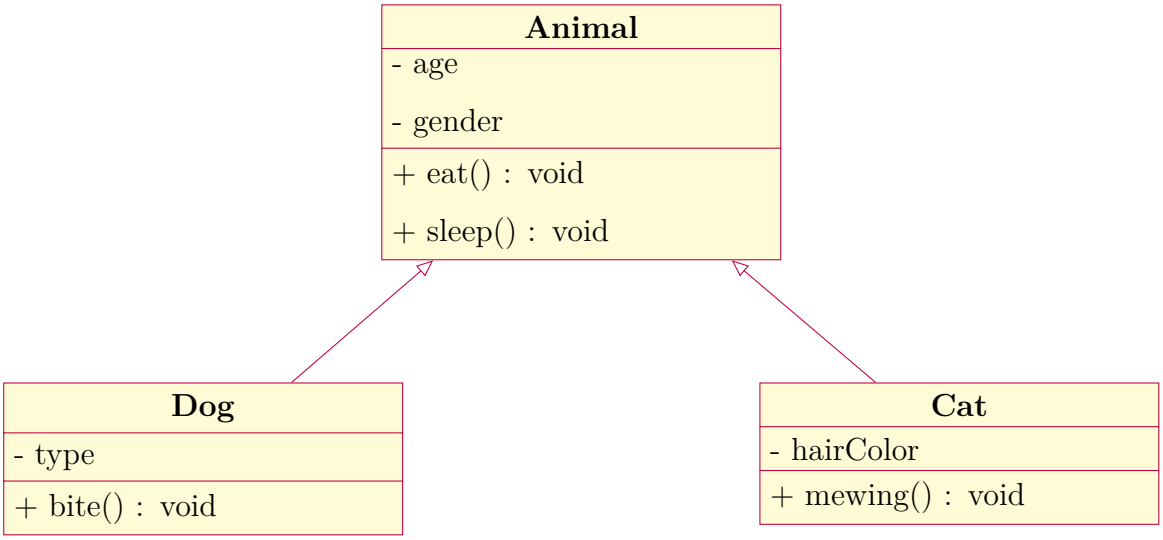


图 7.1: 继承

产生继承关系后，子类可以使用父类中的属性和方法，也可以定义子类独有的属性和方法。

```
1 class subclass(superclass1, ...):
2     # code
```

在进行继承的时候，子类会继承父类之中全部定义的结构。但是对于构造方法的继承是比较特殊的，需要考虑两种情况：

1. 当父类定义了构造方法，但是子类没有定义构造方法时，实例化子类对象会自动调用父类中提供的无参构造方法。

2. 当子类定义了构造方法时，将不再默认调用父类中的任何构造方法，但是可以手动调用。如果有需要也可以通过 super 类的实例实现子类调用父类结构的需求。

继承

```
class Animal:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_age(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

    def eat(self):
        print("吃饭")

    def sleep(self):
        print("睡觉")

class Dog(Animal):
    def __init__(self, name, age, type):
        super().__init__(name, age)
        self.__type = type

    def set_type(self, type):
        self.__type = type

    def get_type(self):
        return self.__type

    def bite(self):
```

```
36          print("咬人")
37
38  def main():
39      dog = Dog("狗子", 3, "哈士奇")
40
41      print("姓名：%s，年龄：%d，品种：%s" % (
42              dog.get_name(), dog.get_age(), dog.get_type()))
43
44      dog.eat()
45      dog.sleep()
46      dog.bite()
47
48  if __name__ == "__main__":
49      main()
```

运行结果

```
姓名：狗子，年龄：3，品种：哈士奇
吃饭
睡觉
咬人
```

7.5.2 多继承

多继承指一个子类可以同时继承多个父类的内容，在多继承实现中只需要编写多个父类的名称即可。利用多继承的最大优势在于可以在进行子类操作的时候将多个父类中定义的结构全部保留继续使用。

多继承

```
1  class Date:
2      def set_date(self, year=1970, month=1, day=1):
3          self.__year = year
4          self.__month = month
```

```
        self.__day = day

    def get_date(self):
        return "%04d/%02d/%02d" % (
            self.__year, self.__month, self.__day)

class Time:
    def set_time(self, hour=0, minute=0, second=0):
        self.__hour = hour
        self.__minute = minute
        self.__second = second

    def get_time(self):
        return "%02d:%02d:%02d" % (
            self.__hour, self.__minute, self.__second)

class DateTime(Date, Time):
    def __init__(self, year=1970, month=1, day=1,
                       hour=0, minute=0, second=0):
        super().set_date(year, month, day)
        super().set_time(hour, minute, second)

    def __repr__(self):
        return super().get_date() + " " + super().get_time()

def main():
    date_time = DateTime(2021, 4, 6, 14, 38, 40)
    print(date_time)

if __name__ == "__main__":
    main()
```

运行结果

```
2021/04/06 14:38:40
```

7.6 多态

7.6.1 多态（Polymorphism）

多态是同一个行为具有多个不同表现形式或形态的能力。例如可以把一只哈士奇，当成它的父类去看待，因此哈士奇是一只狗、一个动物或一个生物。

在类继承的结构之中，很难保证父类中的某些操作方法可以被子类继续拿来使用。这个时候子类为保留住原始的方法名称，同时也为了可以对功能实现进一步的扩充，就可以利用方法覆写。

多态

```
import math

class Shape:
    def get_area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, length):
        self.__width = width
        self.__length = length

    def get_area(self):
        return self.__length * self.__width

class Circle(Shape):
    def __init__(self, radius):
        self.__radius = radius

    def get_area(self):
        return math.pi * self.__radius ** 2

def shape_area(obj):
```

```
23      if isinstance(obj, Shape):
24          return obj.get_area()
25
26  def main():
27      print("长方形面积: %.2f" % shape_area(Rectangle(6, 11)))
28      print("圆形面积: %.2f" % shape_area(Circle(5)))
29
30  if __name__ == "__main__":
31      main()
```

运行结果

```
长方形面积: 66.00
圆形面积: 78.54
```

Chapter 8 文件操作

8.1 文件操作

8.1.1 文件操作

计算机对于数据的存储一般可以通过文件的形式来完成。Python 中直接提供有文件的 I/O（Input/Output）处理函数操作，能够方便地实现读取和写入。

open() 的功能是进行文件的打开，在进行文件打开的时候如果不设置任何的模式类型，则默认为 r（只读模式）。

```
1 def open(file, mode='r', buffering=None, encoding=None,
2         errors=None, newline=None, closefd=True
3 )
```

打开模式	功能
r	使用只读模式打开文件，此为默认模式
w	写模式，如果文件存在则覆盖，文件不存在则创建
x	写模式，新建一个文件，如果该文件已存在则会报错
a	内容追加模式
b	二进制模式
t	文本模式（默认）
+	打开一个文件进行更新（可读可写）

表 8.1: 文件打开模式

如果以只读的模式打开文件，并且文件路径不存在的话，就会出现 FileNotFoundError 的错误信息。

文件操作

```python
def main():
    file = open(file="test.txt", mode="w")
    print("文件名称: %s" % file.name)
    print("访问模式: %s" % file.mode)
    print("文件状态: %s" % file.closed)
    print("关闭文件...")
    file.close()
    print("文件状态: %s" % file.closed)

if __name__ == "__main__":
    main()
```

运行结果

```
文件名称: test.txt
访问模式: w
文件状态: False
关闭文件...
文件状态: True
```

8.1.2 文件读写

当使用 open() 打开一个文件后，就可以使用创建的文件对象进行读写操作。

方法	功能
def close(self)	关闭文件资源
def flush(self)	强制刷新缓冲区
def read(self, n: int = -1)	默认读取全部，也可设置读取个数
def readlines(self, hint: int = -1)	读取所有数据行，以列表形式返回
def readline(self, limit: int = -1)	读取每行数据，也可设置读取个数
def write(self, s: AnyStr)	文件写入
def writelines(self, lines, List[AnyStr])	写入一组数据

表 8.2: 文件读写

既然所有的文件对象最终都需要被开发者关闭，那么可以结合 with 语句实现自动的关闭处理。通过 with 实现所有资源对象的连接和释放是在 Python 中编写资源操作的重要技术手段，通过这样的操作可以极大地减少和优化代码结构。

使用读模式打开文件后，可以使用循环读取每一行的数据内容。Python 在进行文件读取操作的时候也可以进一步简化操作。文件对象本身是可以迭代的，在迭代的时候是以换行符进行分割，每次迭代就读取到一行数据内容。

读取文件

data.txt

```
1 小灰    16
2 小白    17
3 小黄    21
```

read_file.py

```
1 def main():
2     with open(file="data.txt", mode="r", encoding="utf-8") as file:
3         for line in file:
4             print(line, end='')
5
6 if __name__ == "__main__":
7     main()
```

运行结果

```
小灰 16
小白 17
小黄 21
```

写入文件

```
1 def main():
2     with open(file="data.txt", mode="w", encoding="utf-8") as file:
```

```
        info = {"小灰": 16, "小白": 17, "小黄": 21}
        for name, age in info.items():
            file.write("%s\t%d\n" % (name, age))

if __name__ == "__main__":
    main()
```

运行结果 data.txt

```
小灰 16
小白 17
小黄 21
```

8.2 os 模块

8.2.1 os 模块

os 模块是 Python 用于与操作系统进行交互的一个操作模块，这个模块提供有大量与系统相关的处理函数，开发者可以直接通过 Python 程序进行操作系统的功能调用。

方法	功能
getcwd()	获取当前的工作目录
chdir(path)	修改工作目录
system()	执行操作系统命令
symlink(src, dst)	创建软链接
link(src, dst)	创建硬链接

表 8.3: os 模块

8.2.2 os.path 子模块

os.path 是 os 模块之中的一个子模块，该子模块的核心作用在于进行路径处理操作。Python 的程序代码本身是强调跨平台的，既然要进行跨平台的开发，尤其是在 I/O 路径的处理上就特别要引起注意。在 Windows 系统下路径分隔符使用的是【\】，而在 Linux 系统下使用的路径分隔符是【/】。所以在进行程序编写的时候就必须考虑到不同平台的设计问题。

使用 os.path 模块中提供的一系列函数可以针对给定的路径进行拆分处理以及判断和取得数据信息。

Python 需要考虑不同操作系统的跨平台的特点，所以对于访问路径需要进行适当的变更，根据不同的操作系统使用不同的路径分隔符。如果每一次都判断操作系统就过于繁琐了，可以直接使用 os.path 中提供的变量来完成。

变量	功能
curdir	表示当前文件夹【.】，一般可以省略
pardir	上一层文件夹【..】
sep	系统路径分隔符，Windows 为【\】，Linux 为【/】
extsep	文件名称和后缀之间的间隔符【.】

表 8.4: 路径分隔符

获取路径信息

```
import os

PATH = "code" + os.sep +"第10章 文件操作" + os.sep \
        + "10.3 os模块" + os.sep + "data.txt"

def main():
    if os.path.exists(PATH):
        print("绝对路径: %s" % os.path.abspath(PATH))
        print("文件名称: %s" % os.path.basename(PATH))
        print("文件大小: %s" % os.path.getsize(PATH))
        print("当前路径是否为文件: %s" % os.path.isfile(PATH))
        print("当前路径是否为目录: %s" % os.path.isdir(PATH))

if __name__ == "__main__":
    main()
```

运行结果

```
绝对路径: C:\Users\Administrator\Desktop\Python\code\第10章 文件操作\10.3 os模块\data.txt
文件名称: data.txt
文件大小: 15
当前路径是否为文件: True
当前路径是否为目录: False
```

8.3 csv 模块

8.3.1 csv 文件

CSV （Comma-Separated Values，逗号分隔值/字符分隔值）是一种文件的格式，在该类型的文件里面一般会保存多个数据信息的内容，但是每一个数据信息一定都有各自的组成部分，用这样的文件进行数据采集内容的记录。CSV 文件是跟人工智能和数据分析有直接联系的一种数据存储文件。

CSV 是一种以纯文件方式进行数据记录的存储格式，在 CSV 文件内容使用不同的数据行记录数据的内容，每行数据使用特定的符号（一般是逗号）进行数据项的拆分，这样就形成了一种相对简单且通用的数据格式。在实际开发中利用 CSV 数据格式可以方便实现大数据系统中对于数据采集结果的信息记录，也可以方便进行数据文件的传输，同时 CSV 文件格式也可以被 Excel 工具所读取。

CSV 文件是可以通过 Excel 工具打开的，当一个 CSV 文件被创建之后，在 Windows 系统中会自动和 Excel 软件进行关联。

8.3.2 csv 读写操作

在 Python 中直接提供有 csv 模块，利用这个模块可以方便地实现数据的写入和读取操作，在 CSV 文件内容一般对于不同的数据项都要使用逗号分隔。除了数据之外，在 CSV 文件内容还可以设置文件标题。

写入 csv 文件

```
1 import csv
2 import random
3
4 HEADER = ["Location", "Longitude", "Latitude"]
5
6 def main():
7     # 如果不使用newline，那么每行记录之间就会多出一个空行
```

```
    with open(file="location.csv", mode="w",
                newline="", encoding="utf-8") as file:
        csv_writer = csv.writer(file)       # 创建csv写入对象
        csv_writer.writerow(HEADER)         # 写入头部信息
        for i in range(1, 11):
            longitude = round(random.random() * 180, 3)   # [0, 180)
            latitude = round(random.random() * 90, 3)     # [0, 90)
            csv_writer.writerow(["loc-%d" % i, longitude, latitude])

if __name__ == "__main__":
    main()
```

运行结果 location.csv

```
Location,Longitude,Latitude
loc-1,176.165,35.458
loc-2,12.729,56.247
loc-3,6.605,45.14
loc-4,15.123,53.435
loc-5,131.984,11.927
loc-6,155.038,35.681
loc-7,98.772,15.125
loc-8,70.991,30.328
loc-9,152.967,30.372
loc-10,96.362,76.798
```

读取 csv 文件

```
import csv

def main():
    with open(file="location.csv", mode="r",
                newline="", encoding="utf-8") as file:
        csv_reader = csv.reader(file)   # 创建csv读取对象
```

```
7           header = next(csv_reader)        # 读取标题行
8           print(header)
9           for row in csv_reader:
10              print(row)
11
12  if __name__ == "__main__":
13      main()
```

运行结果

```
['Location', 'Longitude', 'Latitude']
['loc-1', '176.165', '35.458']
['loc-2', '12.729', '56.247']
['loc-3', '6.605', '45.14']
['loc-4', '15.123', '53.435']
['loc-5', '131.984', '11.927']
['loc-6', '155.038', '35.681']
['loc-7', '98.772', '15.125']
['loc-8', '70.991', '30.328']
['loc-9', '152.967', '30.372']
['loc-10', '96.362', '76.798']
```