Python

极夜酱

目录

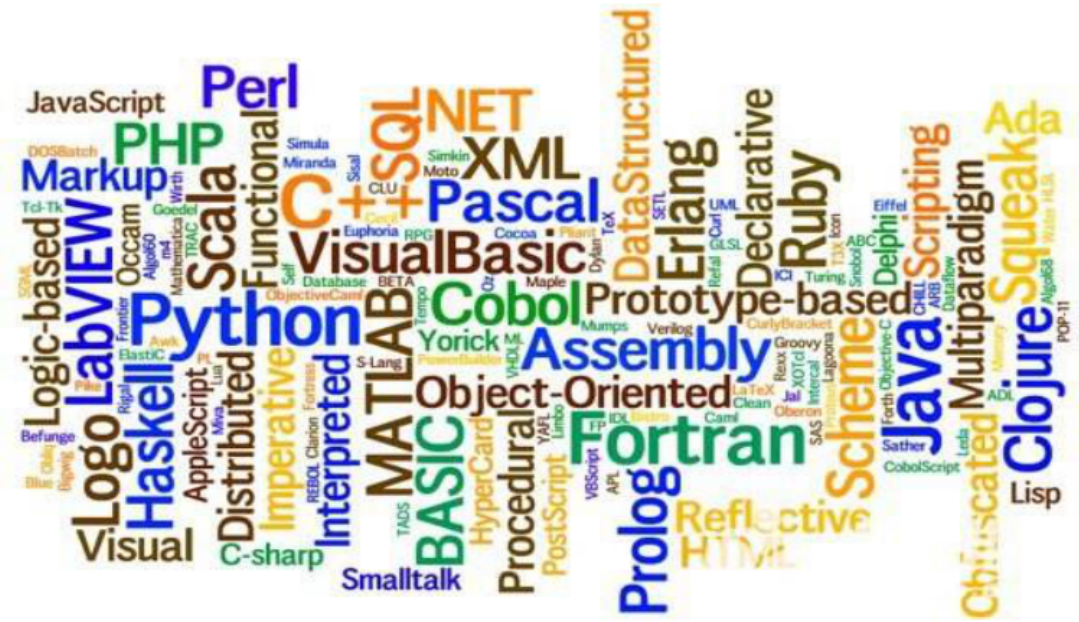
Chapter 1 Hello World!

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。

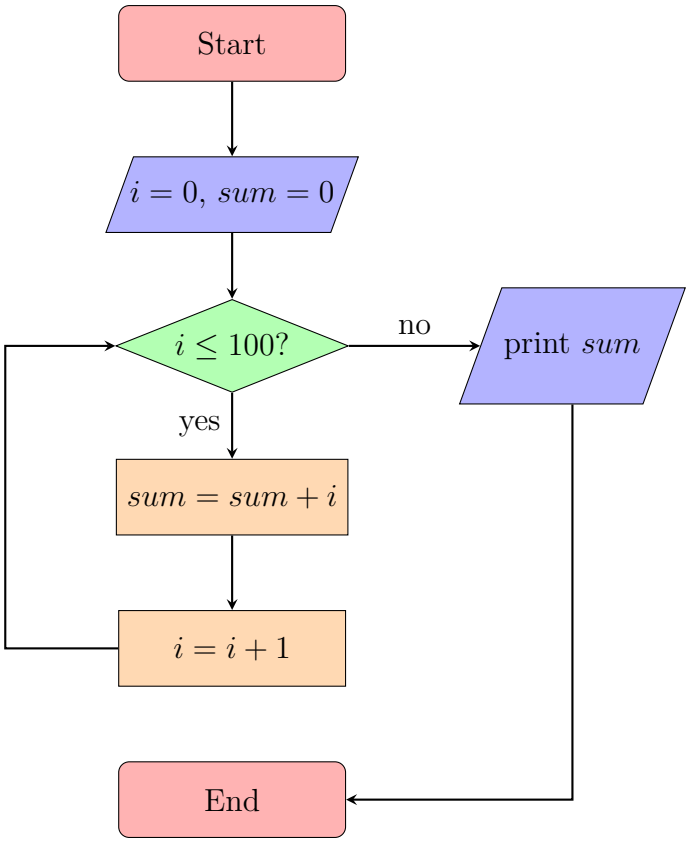


图 1.1: 计算 $\sum_{i=1}^{100} i$ 的流程图

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. 序列
    - 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
```

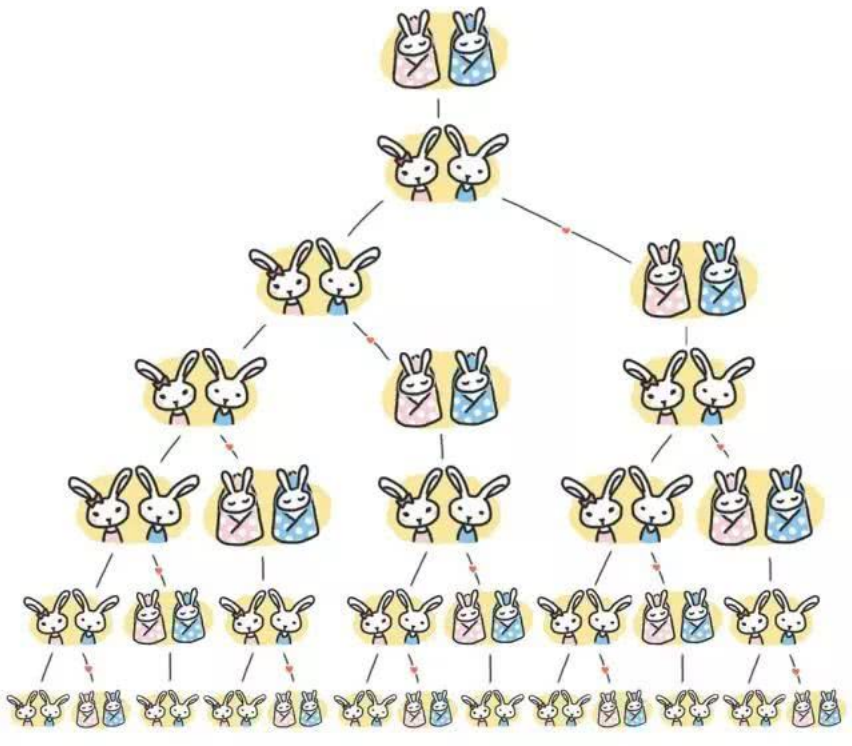
累加

```
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 序列

4.1 列表

4.1.1 列表（List）

列表用于存储多个数据，使用 [] 或 list() 函数创建。列表中的元素可以通过下标来访问，下标从 0 开始。列表除了正向索引访问之外，也可以进行反向索引访问。

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

print(lst[0])       # 1
print(lst[1])       # 2
print(lst[2])       # 3

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

print(lst[3])       # IndexError
```

+ 运算符可以用于列表的拼接，* 运算符可以用于列表的重复。

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

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

4.1.2 in

in 运算符用于判断某个元素是否在序列中，如果在则返回 True，否则返回 False。

查询

```python
languages = ["C", "C++", "Python", "Java"]
key = input("Enter a language: ")

if key in languages:
    print("Found")
else:
    print("Not found")
```

运行结果

```
Enter a language: Python
Found
```

4.1.3 切片（Slicing）

切片用于截取列表中的一部分元素，切片使用 [start:end:step] 进行操作，其中：

- start：切片开始下标（包含），默认为 0
- end：切片结束下标（不包含），默认为列表长度
- step：切片的步长，默认为 1

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

print(lst)          # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(lst[2:7])     # [2, 3, 4, 5, 6]
print(lst[:5])      # [0, 1, 2, 3, 4]
print(lst[3:])      # [3, 4, 5, 6, 7, 8, 9]
print(lst[:])       # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(lst[::2])     # [0, 2, 4, 6, 8]
```

4.1.4 列表方法

列表提供了很多内置方法，可以用于处理列表中的数据。

方法	功能
append()	追加数据
extend()	追加列表
insert()	指定位置插入
remove()	删除第一个出现的指定数据
pop()	删除指定位置的数据
index()	查询指定数据第一次出现的位置
count()	统计指定数据在列表中出现的次数
sort()	对列表进行排序
reverse()	对列表进行反转
clear()	清空列表

表 4.1: 列表方法

列表方法

```
1  lst = list(range(5))
2  print("lst =", lst)
3
4  lst.append(5)
5  print("append(5):", lst)
6
7  lst.insert(0, 8)
8  print("insert(0, 8):", lst)
9
10 lst.extend([8, 2, 3])
11 print("extend([8, 2, 3]):", lst)
12
13 lst.remove(5)
14 print("remove(5):", lst)
```

```
15
16 lst.pop(0)
17 print("pop(0):", lst)
18
19 print("index(3):", lst.index(3))
20
21 print("count(8):", lst.count(8))
22
23 lst.sort()
24 print("sort():", lst)
25
26 lst.sort(reverse=True)
27 print("sort(reverse=True):", lst)
28
29 lst.reverse()
30 print("reverse():", lst)
31
32 lst.clear()
33 print("clear():", lst)
```

运行结果

```
lst = [0, 1, 2, 3, 4]
append(5): [0, 1, 2, 3, 4, 5]
insert(0, 8): [8, 0, 1, 2, 3, 4, 5]
extend([8, 2, 3]): [8, 0, 1, 2, 3, 4, 5, 8, 2, 3]
remove(5): [8, 0, 1, 2, 3, 4, 8, 2, 3]
pop(0): [0, 1, 2, 3, 4, 8, 2, 3]
index(3): 3
count(8): 1
sort(): [0, 1, 2, 2, 3, 3, 4, 8]
sort(reverse=True): [8, 4, 3, 3, 2, 2, 1, 0]
reverse(): [8, 4, 3, 3, 2, 2, 1, 0]
clear(): []
```

4.1.5 二维列表

二维列表由行和列两个维度组成，行和列的下标同样也都是从 0 开始。二维列表可以看成是由多个列表组成的，因此二维列表中的每个元素都是一个列表。

```
lst = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

lst[0][0]	lst[0][1]	lst[0][2]	lst[0][3]
lst[1][0]	lst[1][1]	lst[1][2]	lst[1][3]
lst[2][0]	lst[2][1]	lst[2][2]	lst[2][3]

在初始化二维列表时，为了能够更直观地看出二维列表的结构，可以将每一行单独写在一行中。

```
lst = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
]
```

矩阵运算

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1-0 & 3-0 \\ 1-7 & 0-5 \\ 1-2 & 2-1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ -6 & -5 \\ -1 & 1 \end{bmatrix}$$

```
A = [
    [1, 3],
```

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

B = [
    [0, 0],
    [7, 5],
    [2, 1]
]

C = []
print("Matrix Addition")
for i in range(3):
    C.append([])
    for j in range(2):
        C[i].append(A[i][j] + B[i][j])
        print("%3d" % C[i][j], end='')
    print()

C = []
print("Matrix Subtraction")
for i in range(3):
    C.append([])
    for j in range(2):
        C[i].append(A[i][j] - B[i][j])
        print("%3d" % C[i][j], end='')
    print()
```

运行结果

```
Matrix Addition
  1  3
  8  5
  3  3
Matrix Subtraction
  1  3
 -6 -5
 -1  1
```

4.2 元组

4.2.1 元组（Tuple）

元组与列表类似，但是元组中的元素是不可修改的。元素使用 () 或 tuple() 定义，当元组只有一个元素时，必须在元素后面加上逗号。

两点间距离

```
1 import math
2
3 p1 = (0, 0)
4 p2 = (3, 4)
5 distance = math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)
6 print(distance)
```

运行结果

```
5.0
```

4.2.2 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

```
lst = [4, 0, 1, 3, 2]
tup = (8, 5, 7, 9)

print(len(lst))     # 5
print(len(tup))     # 4

print(max(lst))     # 4
print(max(tup))     # 9

print(min(lst))     # 0
print(min(tup))     # 5

print(sum(lst))     # 10
print(sum(tup))     # 29
```

4.3 集合

4.3.1 集合（Set）

集合表示一组无序且不重复的元素，使用 {} 或 set() 定义。

集合是无序的，因此不能通过下标来访问集合中的元素，但是可以通过 in 来判断元素是否在集合中。

集合支持数学上的集合运算，包括交集、并集、差集等：

- 交集：intersection() 或 &
- 并集：union() 或 |
- 差集：difference() 或-

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}

print(s1 & s2)              # {3}
print(s1.intersection(s2))  # {3}

print(s1 | s2)              # {1, 2, 3, 4, 5}
print(s1.union(s2))         # {1, 2, 3, 4, 5}

print(s1 - s2)              # {1, 2}
print(s1.difference(s2))    # {1, 2}
```

列表去重

```
lst = [1, 9, 2, 0, 0, 9]
lst = list(set(lst))
print(lst)
```

运行结果

```
[0, 1, 2, 9]
```

4.4 字符串

4.4.1 字符串修改

方法	功能
lower()	转换小写
upper()	转换大写
capitalize()	首字母大写
strip()	去除首尾空白字符
replace()	字符串替换

字符串修改

```
1  s = "Hello World!"
2
3  print("[Lower]")
4  print(s.lower())
5
6  print("[Upper]")
7  print(s.upper())
8
9  print("[Capitalize]")
10 print(s.capitalize())
11
12 print("[Strip]")
13 print("   Hello World!\n \t".strip())
14
15 print("[Replace]")
16 print(s.replace("Hello", "Bye"))
```

运行结果

```
[Lower]
hello world!
[Upper]
HELLO WORLD!
[Capitalize]
Hello world!
[Strip]
Hello World!
[Replace]
Bye World!
```

4.4.2 字符串分割

方法	功能
join()	字符串拼接
split()	字符串分割

日期

```
1  date_time = "2023/1/14 23:26:51"
2
3  date, time = date_time.split(" ")
4
5  year, month, day = date.split("/")
6  hour, minute, second = time.split(":")
7
8  date = [day, month, year]
9  date = "/".join(date)
10
11 if int(hour) < 12:
```

```
    time = [hour, minute, second]
    time = ":".join(time) + " AM"
else:
    time = [str(int(hour) - 12), minute, second]
    time = ":".join(time) + " PM"

date_time = date + " " + time
print(date_time)
```

运行结果

```
14/1/2023 11:26:51 PM
```

4.5 字典

4.5.1 字典（Dictionary）

字典中的每个元素都是一组键值对（key-value pair），其中 key 是唯一的，value 可以重复。字典是一种无序的结构，通过 {} 或 dict() 定义，通过 key 来访问 value。

```
1 info = {"name": "Terry", "age": 24, "height": 179.2}
2 info = dict(name="Terry", age=24, height=179.2)
```

在使用 for 循环迭代字典时，迭代的是字典中的 key，并不是一个键值对。

```
1 for key in info:
2     print("key=%s, value=%s" % (key, info[key]))
```

运行结果

```
key=name, value=Terry
key=age, value=24
key=height, value=179.2
```

通过 items() 方法可以返回一个包含所有键值对的列表，列表中的每个元素都是一个键值对。这样再使用 for 循环迭代字典时，就可以同时迭代 key 和 value。

```
1 for key, value in info.items():
2     print("key=%s, value=%s" % (key, value))
```

运行结果

```
key=name, value=Terry
key=age, value=24
key=height, value=179.2
```

4.5.2 字典方法

方法	功能
keys()	获取字典中全部的 key
values()	获取字典中全部的 value
update()	更新字典数据
get()	根据 key 获取 value
pop()	根据 key 删除键值对
popitem()	随机删除一个键值对
clear()	清空字典

表 4.3: 字典方法

字典方法

```
1  info = {"name": "Terry"}
2  print("info =", info)
3
4  info.update({"age": 24, "height": 179.2})
5  print("update() =", info)
6
7  print("keys() =", info.keys())
8  print("values() =", info.values())
9
10 print("get('age') = ", info.get("age"))
11
12 info.pop("height")
13 print("pop('height') =", info)
14
15 info.popitem()
16 print("popitem() =", info)
17
18 info.clear()
19 print("clear() =", info)
```

运行结果

```
info = {'name': 'Terry'}
update() = {'name': 'Terry', 'age': 24, 'height': 179.2}
keys() = dict_keys(['name', 'age', 'height'])
values() = dict_values(['Terry', 24, 179.2])
get('age') =  24
pop('height') = {'name': 'Terry', 'age': 24}
popitem() = {'name': 'Terry'}
clear() = {}
```

词频统计

```
import string

text = """John sat on the park bench,
eating his sandwich and enjoying the warm sun on his face.
He watched the children playing and the ducks swimming in the pond,
feeling content and at peace.
"""

words = text.split()
frequency = {}

for word in words:
    word = word.lower().strip(string.punctuation)

    if word not in frequency:
        frequency[word] = 1
    else:
        frequency[word] += 1

for word, count in frequency.items():
    print("%s: %d" % (word, count))
```

运行结果

```
john: 1
sat: 1
on: 2
the: 5
park: 1
bench: 1
eating: 1
his: 2
sandwich: 1
and: 3
enjoying: 1
warm: 1
sun: 1
face: 1
he: 1
watched: 1
children: 1
playing: 1
ducks: 1
swimming: 1
in: 1
pond: 1
feeling: 1
content: 1
at: 1
peace: 1
```

Chapter 5 函数

5.1 函数

5.1.1 函数（Function）

数学中的函数 $y = f(x)$，通过输入 x 的值，经过计算可以得到 y 的值。计算机中的函数也是如此，将输入传给函数，经过处理后，会得到输出。

函数是一段可重复使用的代码，做了一个特定的任务。例如 print() 和 len() 就是函数，其中 print() 的功能是输出字符串，len() 的功能是计算序列的长度。

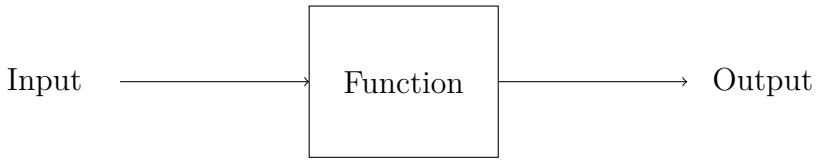


图 5.1: 函数

除了这些内置的函数以外，开发者还可以自定义函数，将程序中会被多次使用的代码或做了一件特定的任务的代码写成一个函数，这样就能避免重复写相同的代码，提高开发效率，也利于维护。

在编写函数时需要：

1. 确定函数的功能
    - 函数名
    - 确保一个函数只做一件事
2. 确定函数的输入（参数）
    - 是否需要参数
    - 参数个数

- 参数类型

3. 确定函数的输出（返回值）

   - 是否需要返回值
   - 返回值类型

最大值

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

    return num1 if num1 > num2 else num2

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

运行结果

```
12
54
-774
```

函数也可以没有返回值，因为它执行完函数后，并不需要将结果返回给调用者。

棋盘

```python
def print_board():
    for i in range(3):
        for j in range(2):
            print("   |", end='')
```

```
5         print()
6         if i < 2:
7             print("---+---+---")
8
9 print_board()
```

运行结果

```
   |   |
---+---+---
   |   |
---+---+---
   |   |
```

5.1.2 函数调用

当调用函数时，程序会记录下当前的执行位置，并跳转到被调用的函数处执行。当被调用的函数执行结束后，程序会回到之前的位置继续执行。

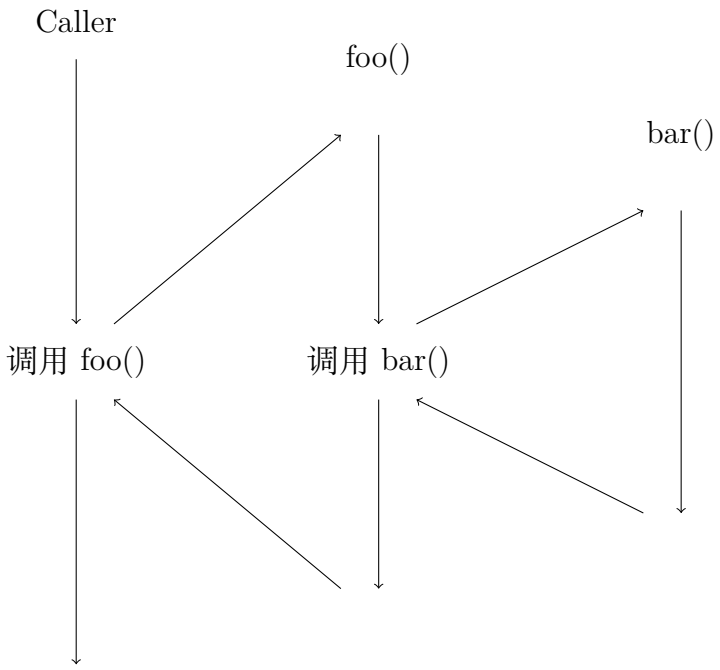


图 5.2: 函数调用

两点间距离

```
import math

def square(x):
    return x ** 2

def distance(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    return math.sqrt(square(x1 - x2) + square(y1 - y2))

x1, y1 = eval(input("Enter (x1, y1): "))
x2, y2 = eval(input("Enter (x2, y2): "))
print("Distance:", distance((x1, y1), (x2, y2)))
```

运行结果

```
Enter (x1, y1): 0, 0
Enter (x2, y2): 3, 4
Distance: 5.0
```

5.1.3 main()

很多编程语言都使用 main() 函数作为程序的入口，在 Python 中 main() 并不是必须的，但是使用 main() 函数可以使代码的结构更加清晰。

___name___ 是一个内置变量，当文件作为程序执行时，其值为 ___main___。

```
def main():
    pass

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

5.2 作用域

5.2.1 局部变量（Local Variable）

定义在块中的变量称为局部变量，它只能在块中访问。当代码块结束时，局部变量就会被销毁。因此，局部变量的生命周期为从声明时开始到所在块结束。

块与块之间的局部变量是互相独立的，即使变量名相同，它们也不是同一个变量。

例如在函数调用中，函数的参数也是局部变量，它们的作用域仅限于函数内。

例如一个用于交换两个变量的函数 swap()，在 main() 中的变量 a 和 b 与 swap() 中的 a 和 b 并不是同一个变量。在调用 swap() 时，是将 main() 中的 a 和 b 的值复制给 swap() 中的 a 和 b。swap() 交换的是其内部的局部变量，并不会对 main() 中的 a 和 b 产生任何影响。

局部变量

```
1  def swap(a, b):
2      a, b = b, a
3      print("swap(): a = %d, b = %d" % (a, b))
4
5  def main():
6      a, b = 1, 2
7
8      print("Before: a = %d, b = %d" % (a, b))
9      swap(a, b)
10     print("After: a = %d, b = %d" % (a, b))
11
12 if __name__ == "__main__":
13     main()
```

运行结果

```
Before: a = 1, b = 2
swap(): a = 2, b = 1
After: a = 1, b = 2
```

5.2.2 全局变量（Global Variable）

全局变量拥有比局部变量更长的生命周期，它的生命周期贯穿整个程序。全局变量可以被程序中所有函数访问。

全局变量一般用于：

- 定义在整个程序中都会被使用到的常量（例如数组容量）
- 被函数间共享的变量（例如计数器）

全局变量

```
1  a, b = 1, 2
2
3  def swap():
4      global a, b
5      a, b = b, a
6      print("swap(): a = %d, b = %d" % (a, b))
7
8  def main():
9      print("Before: a = %d, b = %d" % (a, b))
10     swap()
11     print("After: a = %d, b = %d" % (a, b))
12
13 if __name__ == "__main__":
14     main()
```

运行结果

```
Before: a = 1, b = 2
swap(): a = 2, b = 1
After: a = 2, b = 1
```

5.3 函数参数

5.3.1 默认参数

函数参数可以有默认值，如果在调用函数时不指定某个参数的值，则使用默认值。默认参数必须放在参数列表的最后。

日期

```
def format_date(year=1970, month=1, day=1):
    return "%04d/%02d/%02d" % (year, month, day)

def main():
    print(format_date(2022, 12, 16))
    print(format_date(2022, 12))
    print(format_date(2022))
    print(format_date())

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

运行结果

```
2022/12/16
2022/12/01
2022/01/01
1970/01/01
```

5.3.2 可变参数

函数允许传入任意数量的参数，可变参数包括位置可变参数和关键字可变参数。

位置可变参数使用 *args 将参数打包成一个元组。

乘法

```
def multiply(*args):
    product = 1
    for arg in args:
        product *= arg
    return product

def main():
    print(multiply(1, 2, 3))
    print(multiply(4, 2, 6, 1, 2))

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

运行结果

```
6
96
```

关键字可变参数使用 **kwargs 将参数打包成一个字典。

平均分

```
def get_score_info(name, **kwargs):
    info = "[%s]\n" % name
    for subject, score in kwargs.items():
        info += "%s: %d\n" % (subject, score)
    info += "Average: %.2f" % (sum(kwargs.values()) / len(kwargs))

    return info

def main():
```

```
    print(get_score_info("Alice", Python=85, Math=80))
    print(get_score_info("Bob", Java=77, Math=82))

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

运行结果

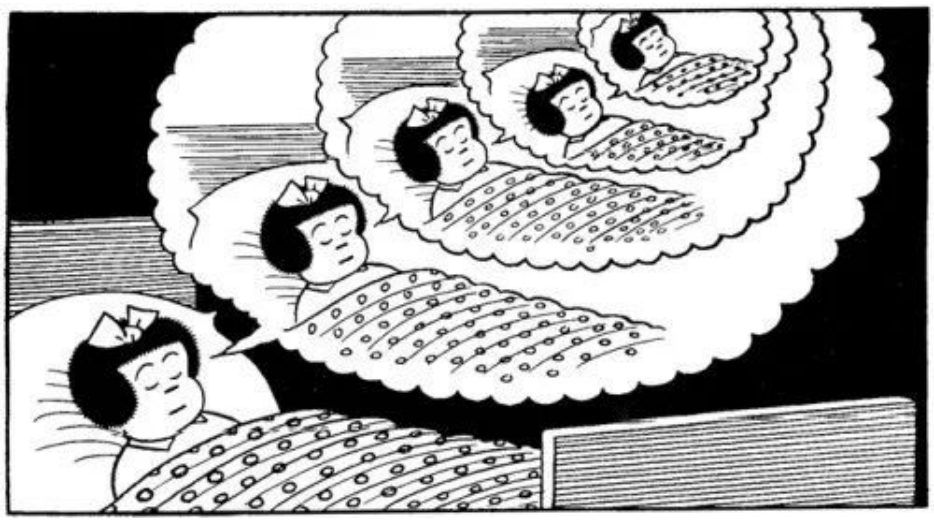
```
[Alice]
Python: 85
Math: 80
Average: 82.50
[Bob]
Java: 77
Math: 82
Average: 79.50
```

5.4 递归

5.4.1 递归（Recursion）

要理解递归，得先理解递归（见5.4章节）。

一个函数调用自己的过程被称为递归。递归可以轻松地解决一些复杂的问题，很多著名的算法都利用了递归的思想。



讲故事

```
def tell_story():
    story = "从前有座山，山里有座庙\n"
    story += "庙里有个老和尚\n"
    story += "老和尚在对小和尚讲故事：\n"
    print(story)

    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
老和尚在对小和尚讲故事：
从前有座山，山里有座庙
庙里有个老和尚
老和尚在对小和尚讲故事：
从前有座山，山里有座庙
庙里有个老和尚
老和尚在对小和尚讲故事：
...
```

一个永远无法结束的递归函数最终会导致栈溢出。因此递归函数需要确定一个结束条件，确保在递归过程中能在合适的地方停止并返回。

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! =", factorial(5))
8
9  if __name__ == "__main__":
10     main()
```

运行结果

```
5! = 120
```

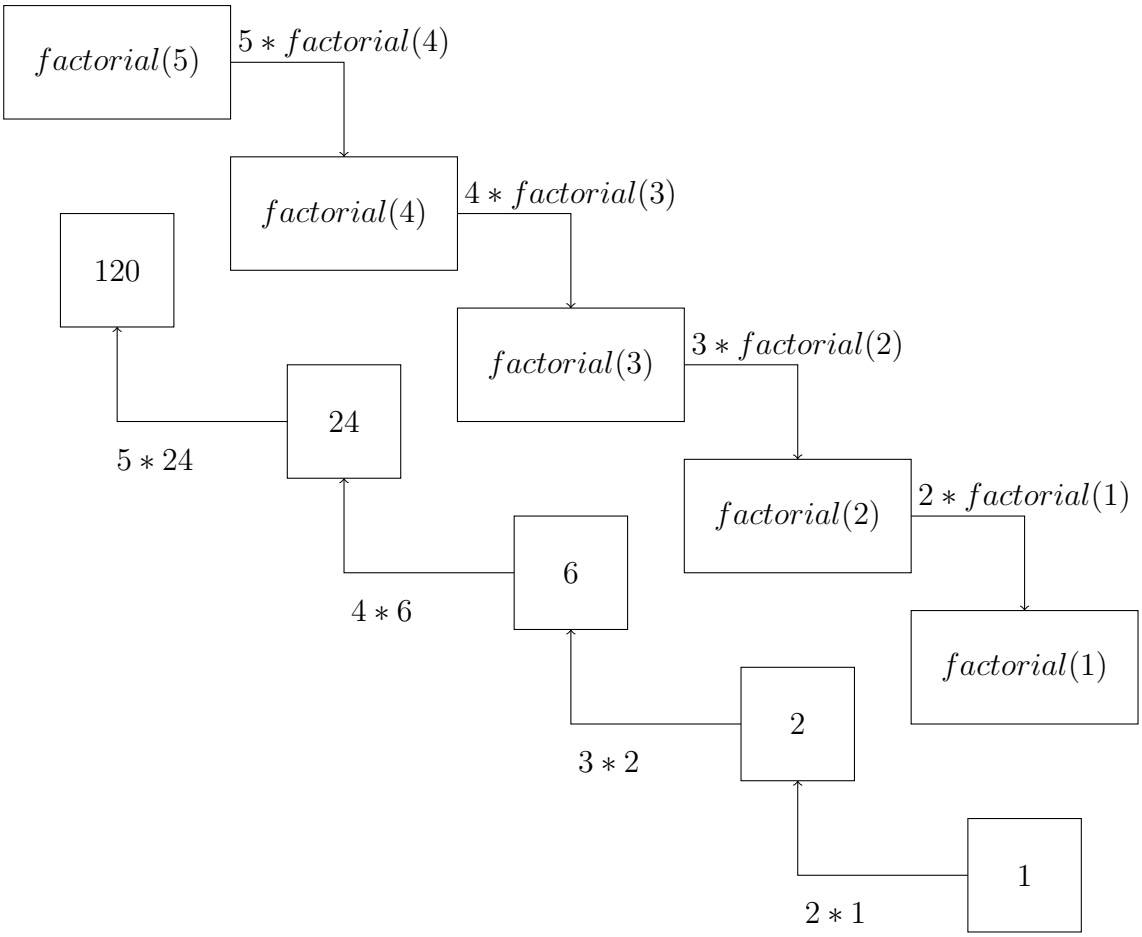


图 5.3: 阶乘

斐波那契数列

```python
def fibonacci(n):
    if n == 1 or n == 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

def main():
    n = 7
    print(fibonacci(n))

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

运行结果

```
21
```

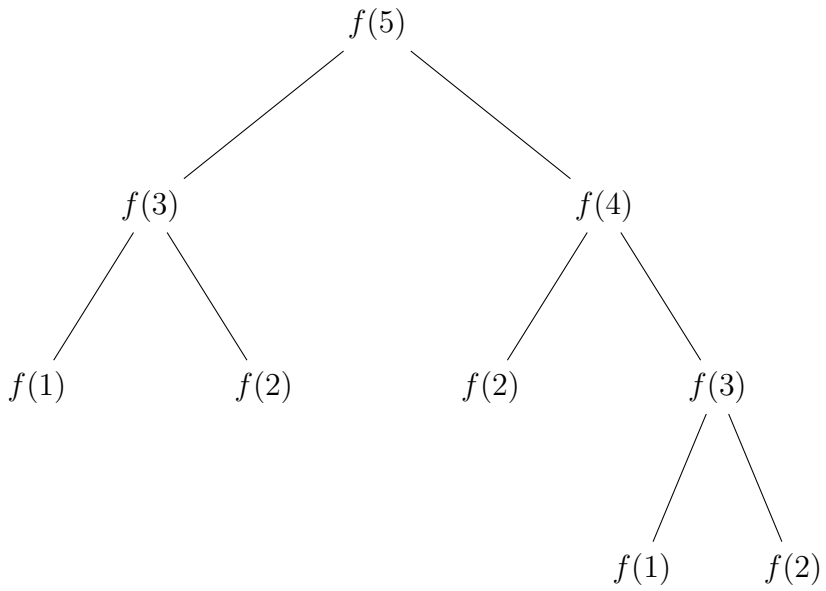


图 5.4: 递归树

递归的特点就是将一个复杂的大问题逐步简化为一个可以解决的小问题，然后再逐步计算出大问题的解。

递归的优点在于代码简洁易懂，但是缺点也很明显，就是效率很低。每次递归都会产生函数调用，而函数调用的开销是很大的，不适合用来解决大规模的问题。

例如在计算斐波那契数列的第 40 项时，递归需要花费大量时间，因为其中包含了大量的重复计算。相比而言，使用循环的方式能够节省大量的时间。因此像阶乘和斐波那契数列这样的情况，通常会采用循环，而不是递归进行计算。

然而还存在很多问题不得不使用递归的思想才能解决。

阿克曼函数

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

```python
def A(m, n):
    if m == 0:
        return n + 1
    elif m > 0 and n == 0:
        return A(m - 1, 1)
    else:
        return A(m - 1, A(m, n - 1))

def main():
    print(A(3, 4))

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

运行结果

```
125
```

汉诺塔

有三根柱子 A、B、C，A 柱子上从下到上套有 n 个圆盘，要求将 A 柱子上的圆盘移动到 C 柱子上。每次只能移动一个圆盘，且大圆盘始终不能叠在小圆盘上面。

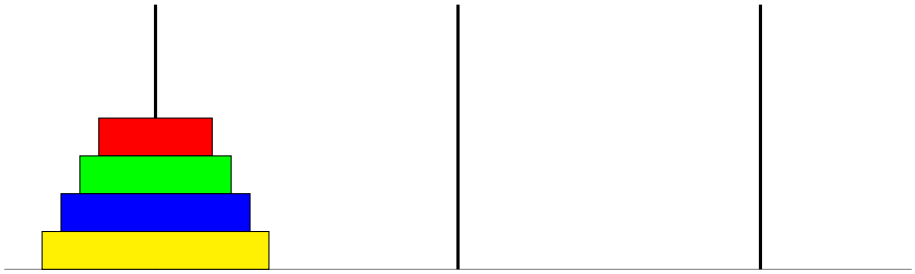


图 5.5: 汉诺塔

递归算法求解汉诺塔问题：

1. 将 n-1 个圆盘从 A 借助 C 移到 B。
2. 将第 n 个圆盘从 A 移到 C。
3. 将 n-1 个圆盘从 B 借助 A 移到 C。

```
move = 0

def hanoi(n, src, mid, dst):
    global move

    if n == 1:
        print(src, "->", dst)
        move += 1
    else:
        # move top n-1 disks from src to mid
        hanoi(n - 1, src, dst, mid)
        print(src, "->", dst)
        move += 1
        # move top n-1 disks from mid to dst
        hanoi(n - 1, mid, src, dst)

def main():
    hanoi(3, "A", "B", "C")
    print("Moves:", move)

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

运行结果

```
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
Moves: 7
```

假设每次移动花费 1 秒，解决一个 64 层的汉诺塔问题大约需要 5800 亿年。

Chapter 6 模块

6.1 random

6.1.1 random

random 模块提供了生成随机数据的功能。

方法	功能
random()	随机生成一个 $[0, 1]$ 之间的浮点数
randint(x, y)	随机生成一个 $[x, y]$ 之间的整数
choice()	从序列中随机返回一个元素
shuffle()	将序列打乱
sample()	从序列中生成一组唯一的随机元素

表 6.1: random 模块

随机密码生成

```python
import random
import string

def password_generator(length):
    characters = string.ascii_letters + string.digits
    return "".join(random.choice(characters) for _ in range(length))

def main():
    length = int(input("Enter length of password: "))
    print(password_generator(length))

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

运行结果

```
Enter length of password: 8
@o8VBuiV
```

6.2 copy

6.2.1 引用（Reference）

引用是指同一块内存空间被交给不同的对象进行操作，当一个对象修改了数据后，另一个对象的数据也会发生改变。

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

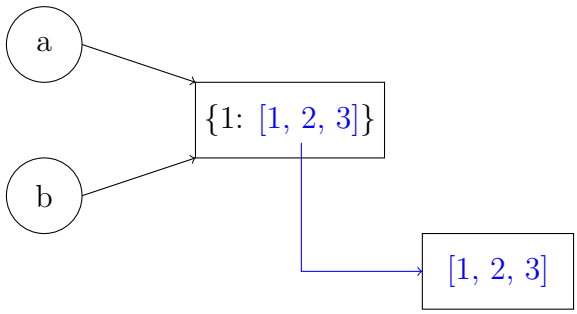


图 6.1: 引用

引用

```
1 info = dict(name="Alice", skills=["Python", "C"])
2 info_ref = info
3 info_ref["skills"].append("Java")
4 print(info)
```

运行结果

```
{'name': 'Alice', 'skills': ['Python', 'C', 'Java']}
```

6.2.2 copy

copy 模块提供了拷贝对象的功能。

copy 是一个专门进行内容拷贝的模块。拷贝分为浅拷贝（shallow copy）和深拷贝（deep copy）：

1. 浅拷贝：只拷贝父对象，不会拷贝对象的内部的子对象。
2. 深拷贝：完全拷贝父对象及其子对象。

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

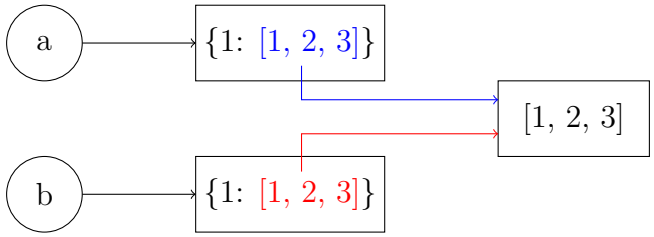


图 6.2: 浅拷贝

浅拷贝

```
import copy

info = dict(name="Alice", skills=["Python", "C"])
info_copy = copy.copy(info)
info.pop("name")
info_copy["skills"].append("Java")

print(info)
print(info_copy)
```

运行结果

```
{'skills': ['Python', 'C', 'Java']}
{'name': 'Alice', 'skills': ['Python', 'C', 'Java']}
```

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

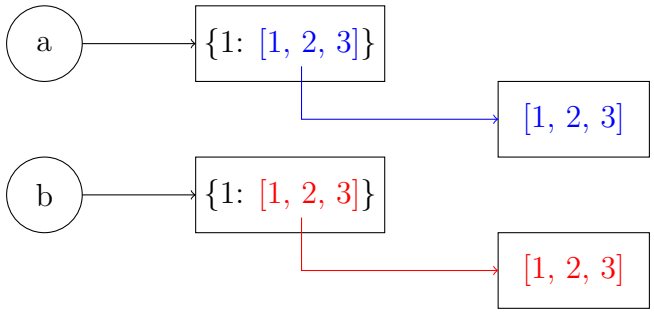


图 6.3: 深拷贝

深拷贝

```
import copy

info = dict(name="Alice", skills=["Python", "C"])
info_copy = copy.deepcopy(info)
info.pop("name")
info_copy["skills"].append("Java")

print(info)
print(info_copy)
```

运行结果

```
{'skills': ['Python', 'C']}
{'name': 'Alice', 'skills': ['Python', 'C', 'Java']}
```

6.3 MapReduce

6.3.1 MapReduce

MapReduce 在数据处理中可以用于大规模的数据过滤、分析、统计等操作。

函数	功能
filter()	过滤序列
map()	对序列数据进行处理
reduce()	对序列数据进行统计

表 6.2: MapReduce 数据处理函数

在数据处理的过程中需要一个处理函数，用于指定数据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。

奇数平方和

```
1  from functools import reduce
2
3  lst = list(range(10))
4  print("lst =", lst)
5
6  odds = list(filter(lambda x: x % 2 == 1, lst))
7  print("odds =", odds)
8
9  squares = list(map(lambda x: x ** 2, odds))
10 print("squares =", squares)
11
12 sum = reduce(lambda x, y: x + y, squares)
13 print("sum =", sum)
```

运行结果

```
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
odds = [1, 3, 5, 7, 9]
squares = [1, 9, 25, 49, 81]
sum = 165
```

6.4 jieba

6.4.1 pip

除了 Python 内置的模块，开发者还可以安装由社区开发并维护的第三方模块。pip 管理工具可以用于安装、卸载、更新第三方模块。

功能	**命令**
搜索模块	pip search [module]
安装模块	pip install [module]
查看已安装模块	pip list
查看过期模块	pip list –outdated
更新模块	pip install –upgrade [module]
卸载模块	pip uninstall [module]

表 6.3: pip 命令

6.4.2 jieba

jieba 是一个在中文自然语言处理中较为常用的工具包之一，用于将中文文本分割成词语。

jieba 支持三种分词模式：

1. 精确模式：最精确的切分，但是速度较慢。
2. 全模式：扫描出所有可以成词的词语，速度非常快，但是不能有效解决歧义。
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，适用于搜索引擎分词。

词频统计

```python
import jieba

FILENAME = "西游记.txt"

def read_file(FILENAME):
    with open(file=FILENAME, mode="r", encoding="UTF-8") as file:
        return file.readlines()

def word_frequency(text, n=10):
    frequency = {}

    for line in text:
        words = jieba.lcut(line)
        for word in words:
            if len(word) == 1:
                continue

            frequency[word] = frequency.get(word, 0) + 1

    items = list(frequency.items())
    items.sort(key=lambda x: x[1], reverse=True)
    return items[:n]

def main():
    text = read_file(FILENAME)
    frequency = word_frequency(text, 20)
    for i, item in enumerate(frequency):
        print("No.%2d: %s - %d" % (i + 1, item[0], item[1]))

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

运行结果

```
No. 1: 行者 - 4078
No. 2: 八戒 - 1677
No. 3: 师父 - 1604
No. 4: 三藏 - 1324
No. 5: 一个 - 1089
No. 6: 大圣 - 889
No. 7: 唐僧 - 802
No. 8: 那里 - 767
No. 9: 怎么 - 754
No.10: 菩萨 - 730
No.11: 我们 - 725
No.12: 沙僧 - 721
No.13: 不知 - 657
No.14: 和尚 - 644
No.15: 妖精 - 631
No.16: 两个 - 594
No.17: 甚么 - 551
No.18: 长老 - 512
No.19: 不是 - 507
No.20: 只见 - 485
```

Chapter 7 文件

7.1 文件 I/O

7.1.1 open()

文件是存储数据的一种常用方式，程序可以从文件中读取和写入数据，从而实现对数据的持久化存储。

在对文件进行操作之前，首先需要使用 open() 函数打开文件，open() 函数可以指定要打开的文件名和打开方式。

打开方式	功能
r	只读，文件必须存在，否则打开失败
w	只写，创建一个新文件
a	追加，如果文件不存在则创建；存在则将数据追加到末尾
r+	以 r 模式打开文件，附带写的功能
w+	以 w 模式打开文件，附带读的功能
a+	以 a 模式打开文件，附带读的功能

表 7.1: 文件打开方式

在对文件操作完成之后，需要调用 close() 函数关闭文件。采用 with 语句可以实现对资源的自动释放，不需要手动调用 close() 函数。

7.1.2 文件 I/O

通过文件的方法可以对文件的 I/O 操作。

方法	功能
read()	默认读取全部，可设置读取个数
readline()	读取每行数据，可设置读取个数
readlines()	读取所有数据行，以列表形式返回
write()	写入数据
writelines()	写入一组数据

表 7.2: 文件 I/O

解析单词

quotes.txt

```
Talk is cheap. Show me the code.
Code never lies, comments sometimes do.
Stay Hungry Stay Foolish.
```

```
import string

output_file = open("words.txt", "w")

with open("quotes.txt") as file:
    for line in file:
        words = line.split()
        for word in words:
            word = word.strip(string.punctuation)
            output_file.write(word + "\n")

output_file.close()
```

运行结果 words.txt

```
Talk
is
cheap
Show
me
the
code
Code
never
lies
comments
sometimes
do
Stay
Hungry
Stay
Foolish
```

7.2 csv

7.2.1 csv

CSV（Comma-Separated Values）是一种以纯文本方式进行数据记录的存储格式，每行数据使用逗号分隔。

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和传输，同时 CSV 文件也可以使用 Excel 查看。

邮箱

user_info.csv

```
ID,First Name,Last Name
9012,Rachel,Booker
2070,Laura,Grey
4081,Craig,Johnson
9346,Mary,Jenkins
5079,Jamie,Smith
```

```
import csv

with open("user_info.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)
    user_info = list(reader)

with open("user_email.csv", "w", newline="\n") as file:
    writer = csv.writer(file)
    writer.writerow(["ID", "First Name", "Last Name", "Email"])
    for user in user_info:
        id, first_name, last_name = user

        writer.writerow([
            id, first_name, last_name,
```

```
16              (last_name[0] + first_name + id + "@gmail.com").lower()
17          ])
```

运行结果 user_email.csv

```
ID,First Name,Last Name,Email
9012,Rachel,Booker,brachel9012@gmail.com
2070,Laura,Grey,glaura2070@gmail.com
4081,Craig,Johnson,jcraig4081@gmail.com
9346,Mary,Jenkins,jmary9346@gmail.com
5079,Jamie,Smith,sjamie5079@gmail.com
```

Chapter 8 面向对象

8.1 封装

8.1.1 类与对象

在面向对象编程中，把构成问题的事物分解成各个对象，每个对象都有自己的数据和行为，程序通过对象之间的交互来实现功能。

类（class）是一个模板，定义了对象的属性和方法，用来描述同一类对象的共同特征和行为。对象（object）是类的实例，它具有类定义的属性和方法。

实例化一个类对象之后就可以通过访问对象的属性和方法来操作对象。

银行账户

```
class BankAccount:
    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

def main():
    account = BankAccount()
    account.owner = "Terry"
    account.account = "6250941006528599"
    account.balance = 50

    print("Owner:", account.owner)
    print("Account:", account.account)
    print("Balance:", account.balance)

```

```
18      account.deposit(100)
19      print("Balance:", account.balance)
20
21      account.withdraw(70)
22      print("Balance:", account.balance)
23
24  if __name__ == "__main__":
25      main()
```

运行结果

```
Owner: Terry
Account: 6250941006528599
Balance: 50
Balance: 150
Balance: 80
```

8.1.2 封装（Encapsulation）

封装是面向对象的重要原则，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随意访问。当要访问该类的数据时，必须通过指定的接口。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。

为了实现封装，需要对类的属性和方法进行访问权限的控制。通常会将类的属性设置为私有属性，然后对外提供一对 setter/getter 方法来访问该属性。

为了避免方法的参数与类的属性重名造成歧义，可以使用 self 关键字用来指代当前对象。

银行账户

```
class BankAccount:
    __ACCOUNT_DIGITS = 16
    __owner = ""
    __account = ""
    __balance = 0

    def set_owner(self, owner):
        if owner != "":
            self.__owner = owner

    def get_owner(self):
        return self.__owner

    def set_account(self, account):
        if len(account) == self.__ACCOUNT_DIGITS:
            self.__account = account

    def get_account(self):
        return self.__account

    def set_balance(self, balance):
        if balance >= 0:
            self.__balance = balance

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount <= 0:
            return False
        self.__balance += amount
        return True

    def withdraw(self, amount):
        if amount <= 0 or amount > self.__balance:
            return False
        self.__balance -= amount
```

```
38          return True
39
40  def main():
41      account = BankAccount()
42      account.set_owner("Terry")
43      account.set_account("6250941006528599")
44      account.set_balance(50)
45
46      print("Owner:", account.get_owner())
47      print("Account:", account.get_account())
48      print("Balance:", account.get_balance())
49
50      account.deposit(100)
51      print("Balance:", account.get_balance())
52
53      account.withdraw(70)
54      print("Balance:", account.get_balance())
55
56  if __name__ == "__main__":
57      main()
```

运行结果

```
Owner: Terry
Account: 6250941006528599
Balance: 50
Balance: 150
Balance: 80
```

8.1.3 构造方法（Constructor）

构造方法是一种特殊的方法，会在创建对象时自动调用，用于创建并初始化对象，构造方法的名字为 ___init___()。

银行账户

```
class BankAccount:
    def __init__(self, owner, account, balance):
        self.__ACCOUNT_DIGITS = 16

        if owner != "":
            self.__owner = owner

        if len(account) == self.__ACCOUNT_DIGITS:
            self.__account = account

        if balance >= 0:
            self.__balance = balance

    def set_owner(self, owner):
        if owner != "":
            self.__owner = owner

    def get_owner(self):
        return self.__owner

    def set_account(self, account):
        if len(account) == self.__ACCOUNT_DIGITS:
            self.__account = account

    def get_account(self):
        return self.__account

    def set_balance(self, balance):
        if balance >= 0:
            self.__balance = balance

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
```

```
36          if amount <= 0:
37              return False
38
39          self.__balance += amount
40          return True
41
42      def withdraw(self, amount, fee=0):
43          if amount <= 0 or amount + fee > self.__balance:
44              return False
45
46          self.__balance -= amount + fee
47          return True
48
49  def main():
50      account = BankAccount("Terry", "6250941006528599", 50)
51
52      print("Account Balance:", account.get_balance())
53
54      account.withdraw(20)
55      print("Account Balance:", account.get_balance())
56
57      account.withdraw(10, 1)
58      print("Account Balance:", account.get_balance())
59
60  if __name__ == "__main__":
61      main()
```

运行结果

```
Account Balance: 50
Account Balance: 30
Account Balance: 19
```

8.2 继承

8.2.1 继承（Inheritance）

继承指一个类可以继承另一个类的特征和行为，并可以对其进行扩展。这样就可以避免在多个类中重复定义相同的特征和行为。

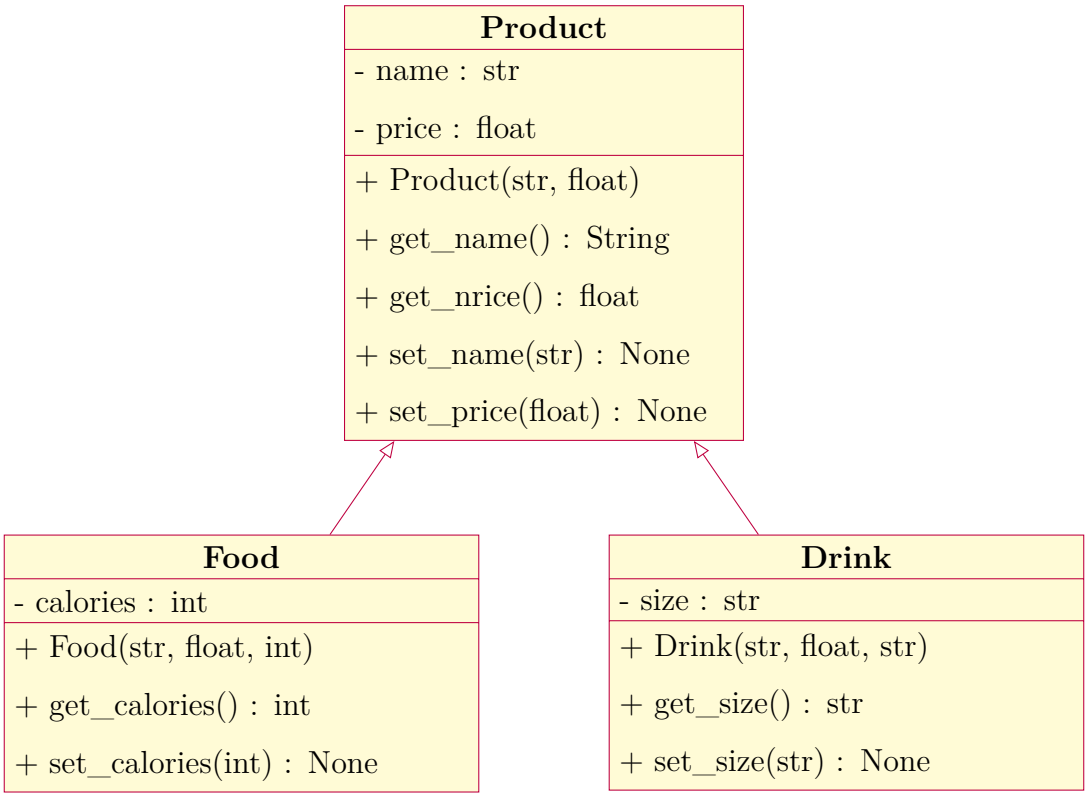


图 8.1: 继承

产生继承关系后，子类可以通过 super 关键字调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。

在创建子类对象时，会先调用父类的构造方法，然后再调用子类的构造方法。因此父类中必须存在一个构造方法，否则将无法创建子类对象。

麦当劳

```
1 class Product:
```

```
    def __init__(self, name, price):
        self.__name = name
        self.__price = price

    def get_name(self):
        return self.__name

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

    def get_price(self):
        return self.__price

    def set_price(self, price):
        self.__price = price
```

```
from product import Product

class Food(Product):
    def __init__(self, name, price, calories):
        super().__init__(name, price)
        self.__calories = calories

    def get_calories(self):
        return self.__calories

    def set_calories(self, calories):
        self.__calories = calories
```

```
from product import Product

class Drink(Product):
    def __init__(self, name, price, size):
        super().__init__(name, price)
        self.__size = size
```

```
7
8       def get_size(self):
9           return self.__size
10
11      def set_size(self, size):
12          self.__size = size
```

```
1  from food import Food
2  from drink import Drink
3
4  def main():
5      food = Food("Cheeseburger", 5.45, 302)
6      drink = Drink("Coke", 3.7, "Large")
7
8      print("Food: %s ($%.2f) %d Kcal" % (
9          food.get_name(), food.get_price(), food.get_calories())
10     )
11     print("Drink: %s ($%.2f) %s" % (
12         drink.get_name(), drink.get_price(), drink.get_size())
13     )
14
15 if __name__ == "__main__":
16     main()
```

运行结果

```
Food: Cheeseburger ($5.45) 302 Kcal
Drink: Coke ($3.70) Large
```

8.2.2 重写（Override）

object 类是所有类的根类，所有的类都直接或者间接地继承自 object 类。object 类中包含的方法在其它所有类中都可以使用，例如 ___str___() 方法等。

当直接输出一个对象时，会自动调用该对象的 ___str___() 方法，将其以字符串的形式输出。

```
print(food);   # <food.Food object at 0x0000011CE7BC3E10>
```

在没有重写 ___str___() 方法的情况下，输出的内容是对象的类名及其地址，但这并不是预期想要的结果。因此，可以重写从父类继承的 ___str___()，以满足程序的需求。

麦当劳

```
class Product:
    def __init__(self, name, price):
        self.__name = name
        self.__price = price

    def get_name(self):
        return self.__name

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

    def get_price(self):
        return self.__price

    def set_price(self, price):
        self.__price = price

    def __str__(self):
        return "%s ($%.2f)" % (self.__name, self.__price)
```

```
from product import Product

class Food(Product):
    def __init__(self, name, price, calories):
```

```
        super().__init__(name, price)
        self.__calories = calories

    def get_calories(self):
        return self.__calories

    def set_calories(self, calories):
        self.__calories = calories

    def __str__(self):
        return "Food: %s %d Kcal" % (
            super().__str__(), self.__calories
        )
```

```
from product import Product

class Drink(Product):
    def __init__(self, name, price, size):
        super().__init__(name, price)
        self.__size = size

    def get_size(self):
        return self.__size

    def set_size(self, size):
        self.__size = size

    def __str__(self):
        return "Drink: %s %s" % (super().__str__(), self.__size)
```

```
from food import Food
from drink import Drink

def main():
    food = Food("Cheeseburger", 5.45, 302)
```

```
    drink = Drink("Coke", 3.7, "Large")

    print(food)
    print(drink)

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

运行结果

```
Food: Cheeseburger ($5.45) 302 Kcal
Drink: Coke ($3.70) Large
```

8.3 多态

8.3.1 多态（Polymorphism）

多态是指对象可以具有多种形态，即同一个对象在不同时刻表现出不同的行为。例如 Dog 和 Cat 都是 Animal 的子类，因此可以将子类对象赋值给父类引用，从而产生多种形态。

由子类类型转型为父类类型，称为向上转型。由父类类型转型为子类类型，称为向下转型。

员工工资

```
class Employee:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

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

    def get_salary(self):
        pass

class FullTimeEmployee(Employee):
    def __init__(self, name, basic_salary, bonus):
        super().__init__(name)
        self.__basic_salary = basic_salary
        self.__bonus = bonus

    def get_salary(self):
        return self.__basic_salary + self.__bonus

```

```python
class PartTimeEmployee(Employee):
    def __init__(self, name, daily_wage, working_days):
        super().__init__(name)
        self.__daily_wage = daily_wage
        self.__working_days = working_days

    def get_salary(self):
        return self.__daily_wage * self.__working_days
```

```python
from employee import FullTimeEmployee
from employee import PartTimeEmployee

def main():
    employees = [
        FullTimeEmployee('Alice', 5783, 173),
        PartTimeEmployee('Bob', 150, 15)
    ]

    for employee in employees:
        print(
            employee.get_name() + ': $' + str(employee.get_salary())
        )

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

运行结果

```
Alice: $5956
Bob: $2250
```

Chapter 9 异常

9.1 异常

9.1.1 异常（Exception）

异常就是程序在运行过程中出现的非正常的情况，它可以被捕获并处理，以防止程序崩溃。

Exception 是一个异常类，发生异常的时候会抛出一个异常对象。如果不处理异常，程序就会被中断。

异常	描述
ArithmeticError	数学运算异常
AttributeError	访问或设置不存在的属性时引发的异常
FileNotFoundError	打开不存在的文件时引发的异常
IndexError	下标越界
KeyError	使用不存在的键访问字典时引发的异常
ZeroDivisionError	除数为零

表 9.1: 常用异常

例如当列表访问越界时，会抛出一个 IndexError 异常；当访问一个不存在的属性时，会抛出一个 AttributeError 异常。

9.1.2 捕获异常

try-except-finally 结构可以用于捕获并处理异常，将可能出现异常的代码放在 try 结构中，将异常处理的代码放在 except 结构中，finally 结构中的代码无论是否出现异常都会执行。

当在 try 结构中出现异常时，程序会跳转到 except 结构中，执行 except 结构中的代码。一个异常被处理后，将不再影响程序的执行。

整除

```
while True:
    try:
        dividend = int(input("Enter an integer for dividend: "))
        divisor = int(input("Enter an integer for divisor: "))
        quotient = dividend / divisor
        print(dividend, "/", divisor, "=", quotient)
    except ValueError:
        print("Only integers supported.")
    except ZeroDivisionError:
        print("Divisor cannot be 0.")
```

运行结果

```
Enter an integer for dividend: 21
Enter an integer for divisor: 4
21 / 4 = 5.25
Enter an integer for dividend: 5
Enter an integer for divisor: 0
Divisor cannot be 0.
Enter an integer for dividend: 3.6
Only integers supported.
```

9.1.3 raise

raise 关键抛出一个异常，抛出异常后，需要由调用者来处理异常。如果调用者没有处理异常，那么异常就继续向上抛出，直到被处理。

阶乘

```
1 def factorial(n):
2     if n < 0:
3         raise ValueError(
4             "Factorial of negative numbers is not defined."
```

```
        )

    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

n = int(input("Enter n: "))
try:
    print(n, "! =", factorial(n))
except ValueError as e:
    print(e)
```

运行结果

```
Enter n: -1
Factorial of negative numbers is not defined.
```

9.2 自定义异常

9.2.1 自定义异常

为了满足某些特定的需求，用户可以自定义异常，自定义异常继承于 Exception 类或其子类。自定义异常的目的是为了提供更具体和有意义的错误处理。

库存

```
class OutOfStockException(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message
```

```
from out_of_stock_exception import OutOfStockException

class Product:
    def __init__(self, name, stock):
        self.name = name
        self.stock = stock

    def purchase(self):
        if self.stock <= 0:
            raise OutOfStockException(self.name + " is out of stock.")
        self.stock -= 1
```

```
from product import Product
from out_of_stock_exception import OutOfStockException

def main():
    product = Product("Cheeseburger", 50)
```

```
6
7       try:
8           for _ in range(60):
9               product.purchase()
10      except OutOfStockException as e:
11          print(e)
12
13  if __name__ == "__main__":
14      main()
```

运行结果

```
Cheeseburger is out of stock.
```