数据结构与算法

Python 基础

张晓平

1 Python 基础

1.1 数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值. 但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据. 不同的数据，需要定义不同的数据类型. 在 Python 中，能够直接处理的数据类型有以下几种：

- 整型
- 浮点型
- 字符串
- 布尔值
- 空值

1.1.1 整型

Python 可以处理任意大小的整数，包括负整数，在程序中的表示方法和数学上的写法一模一样，如：

```
1, 100, -8080, 0, ...
```

因计算机使用二进制，故有时使用十六进制表示整数比较方便，十六进制用前缀`0x`和`0-9, a-f`表示，如：

```
0xff00, 0xa432bf
```

1.1.2 浮点数

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，1.23×10^9 和 12.3×10^8 是完全相等的。

浮点数可以用数学写法，如

```
1.24, 3.14, -9.80, ...
```

而对于很大或 hen 小的浮点数，就必须用科学计数法表示，如

```
1.23e9, 1.2e-6, ...
```

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的，而浮点数运算则可能会有四舍五入的误差。

1.2 字符串

字符串是以单引号'或双引号"括起来的任意文本，如

```
'abc', "xyz", ...
```

请注意，''或""本身只是一种表示方式，不是字符串的一部分，因此，字符串'abc'只有a，b，c这 3 个字符。

如果'本身也是一个字符，那就可以用""括起来，比如"I'm OK"包含的字符是I，'，m，空格，O，K这 6 个字符。

问题. 如果字符串内部既包含'又包含"怎么办？

可用转义字符\来标识，如'I\'m \"OK\"!'表示的字符串内容为

```
I'm "OK"!
```

转义字符\可以转义很多字符，比如

- \n表示换行
- \t表示制表符
- \\表示字符'\'

```
print("I\' ok.")
print("I\'m learning\nPython.")
print('\\\n\\')
```

```
I' ok.
I'm learning
Python.
\
\
```

若字符串里面有很多字符需要转义，就需要加很多\。为了简化，Python 允许使用r''表示''内部的字符串默认不转义。

```
print('\\\n\\')
print(r'\\\n\\')
```

```
\
\
\\\n\\\
```

若字符串内部有很多换行，用\n写在一行不好阅读。为了简化，Python 允许使用'''...'''格式表示多行内容。

```
print('''line1
line2
line3''')
print(r'''hello,\n
world''')
```

```
line1
line2
line3
hello,\n
world
```

1.2.1 布尔值

在 Python 中，可直接用True和False表示布尔值，也可通过布尔运算计算出来：

```
print(True)
print(False)
print(3 > 2)
print(3 > 5)
```

```
True
False
True
False
```

布尔值可用and，or，not运算

```
print(True and True)
print(True and False)
print(False and False)
print(5 > 3 and 1 > 3)
```

```
True
False
False
False
```

```
print(True or True)
print(True or False)
print(False or False)
print(5 > 3 or 1 > 3)
```

```
True
True
False
True
```

```
print(not True)
print(not False)
print(not 1 > 2)
```

```
False
True
True
```

布尔值经常用在条件判断中，比如：

```
age = int(input('Enter your age: '))
if age >= 18:
    print('adult')
else:
    print('teenager')
```

```
Enter your age: 14
teenager
Enter your age: 24
adult
```

1.2.2 空值

空值是 Python 中一个特殊的值，用None表示。None不能理解为0，因为0是有意义的，而None是一个特殊的空值。

1.3 变量

变量在程序中用一个变量名表示，变量名必须是字母、数字和下划线的组合，且不能以数字开头。如

- ```
  n = 1
  ```

  变量n是一个整数

- ```
  a = 1.0
  ```

  变量a是一个浮点数

- ```
  str = 'Hello world!'
  ```

  变量str是一个字符串

- ```
  answer = True
  ```

变量answer是一个布尔值

在 Python 中，等号=为赋值运算符，可以把任意数据类型的对象赋值给变量，同一变量也可重复赋值，并且可以是不同类型的对象。如

```
a = 123
print(a, type(a))
a = 3.14
print(a, type(a))
a = 'ABC'
print(a, type(a))
a = 5 > 3 or 2 < 1
print(a, type(a))
```

```
123 <class 'int'>
3.14 <class 'float'>
ABC <class 'str'>
True <class 'bool'>
```

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。 例如 C 语言是静态语言，

```
int a = 123;  // a is an integer type variable
a = "ABC";    // error: cannot assign string to a
```

和静态语言相比，动态语言更灵活.

请理解变量在计算机内存中的表示。比如说

```
a = 'ABC'
```

Python 解释器干了两件事情:

- 在内存中创建了一个'ABC'的字符串
- 在内存中创建了一个名为a的变量，并将它指向'ABC'

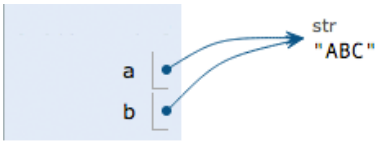
可以把一个变量a赋值给另一个变量b，这个操作实际上是把变量b指向a所指向的数据，如

```
a = 'ABC'
b = a
a = 'XYZ'
print(b)
```

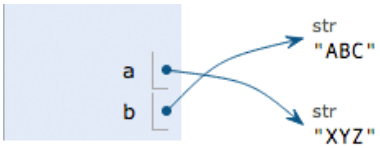
```
ABC
```

- 执行a = 'ABC'，解释器创建了字符串'ABC'和变量a，并把a指向'ABC'.

- 执行b = a，解释器创建了变量b，并把b指向a指向的字符串'ABC'.



- 执行a = 'XYZ'，解释器创建了字符串'XYZ'，并把a的指向改为'ABC'，但b并没有改变.



1.4 常量

在 Python 中，通常用全部大写的变量名表示常量：

```
PI = 3.1415926
```

但事实上PI仍然是一个变量，Python 根本没有任何机制保证 PI 不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量 PI 的值，也没人能拦住你。

1.5 关于除法

在 Python 中，有两种除法，一种除法是/：

```
print(10/3)
```

```
3.3333333333333335
```

/除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数：

```
print(9/3)
```

```
3.0
```

还有一种除法是//，称为地板除，两个整数的除法仍然是整数

```
print(9//3)
print(10//3)
```

```
3
3
```

整数的地板除//永远是整数，即使除不尽，实际上//除法只取结果的整数部分。
Python 还提供一个余数运算，来计算两个整数相除的余数：

```
print(10 % 3)
```

```
1
```

2 字符串与编码

字符串也是一种数据类型。但是，字符串比较特殊的是还有一个编码问题，因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。

最早的计算机在设计时采用 8 个比特（bit）作为一个字节（byte），故一个字节能表示的最大的整数就是 $2^8-1=255((11111111)_2=(255)_{10})$。如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是 $2^{16}-1=65535$，4 个字节可以表示的最大整数是 $2^{32}-1=4294967295$。

由于计算机是美国人发明的，因此，最早只有 127 个字符被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为 ASCII 编码，比如大写字母A的编码是 65，小写字母z的编码是 122。

但是要处理中文的话，一个字节是显然不够的，至少需要两个字节，而且还不能与 ASCII 码冲突，所以我们国家制定了GB2312编码，用于汉字的编码。而全世界有上百种语言，日文的编码为Shift_JIS，韩文编码为Euc-kr，等等。各国有各自的标准，就会不可避免的出现冲突，结果就是在多语言混合的文本中，显示出来就会出现乱码。

因此，Unicode 应运而生。Unicode 把所有语言都统一到一套编码里，这样就不会再有乱码问题了。Unicode 标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要 4 个字节）。现代操作系统和大多数编程语言都直接支持 Unicode。现在，捋一捋 ASCII 编码和 Unicode 编码的区别：ASCII 编码是 1 个字节，而 Unicode 编码通常是 2 个字节。 如

- 'A'用 ASCII 编码是十进制的65，二进制的01000001
- '0'用 ASCII 编码是十进制的48，二进制的00110000。注意字符'0'和整数0的不同
- '中'已经超出了 ASCII 码的范围，用 Unicode 编码是十进制的20013，二进制的01001110 00101101

可以猜测，如果把 ASCII 编码的'A'用 Unicode 编码，只需要在前面补0就可以，因此，'A'的 Unicode 编码是00000000 01000001。

新的问题又出现了：如果统一成 Unicode 编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用 Unicode 编码比 ASCII 编码需要多一倍的存储空间，在存储和传输上就十分不划算。

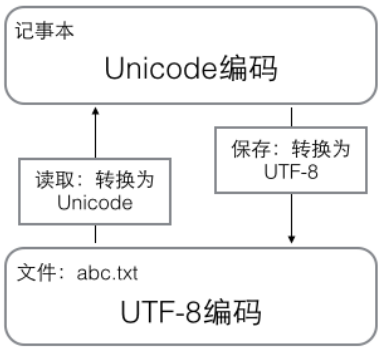
本着节约的精神，又出现了把 Unicode 编码转化为“可变长编码”的 UTF-8 编码。UTF-8 编码把一个 Unicode 字符根据不同的数字大小编码成 1-6 个字节，常用的英文字母被编码成 1 个字节，汉字通常是 3 个字节，只有很生僻的字符才会被编码成 4-6 个字节。如果你要传输的文本包含大量英文字符，用 UTF-8 编码就能节省空间：从上面的表格还可以发现，UTF-8 编码有一个额外的好处，就是 ASCII 编码实际上可以被看成

字符	ASCII	Unicode	UTF-8
'A'	01000001	00000000 01000001	01000001
'中'	01001110 00101101	11100100 10111000 10101101	

是 UTF-8 编码的一部分，所以，大量只支持 ASCII 编码的历史遗留软件可以在 UTF-8 编码下继续工作。

搞清楚了 ASCII、Unicode 和 UTF-8 的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

- 在计算机内存中，统一使用 Unicode 编码；当需要保存到硬盘或者需要传输时，就转换为 UTF-8 编码。
- 用记事本编辑的时候，从文件读取的 UTF-8 字符被转换为 Unicode 字符到内存里，编辑完成后，保存的时候再把 Unicode 转换为 UTF-8 保存到文件：



2.1 Python 字符串

在 Python 3 中，字符串是以 Unicode 编码的，也就是说 Python 字符串支持多语言，如

```
print('包含中文的string')
```

```
包含中文的string
```

对于单个字符的编码，Python 提供了 ord() 函数获取字符的整数表示，chr() 函数把编码转换为对应的字符：

```
print(ord('A'))
print(ord('中'))
print(chr(66))
print(chr(25991))
```

```
65
20013
B
文
```

如果知道字符的整数编码，还可以用十六进制这么写：

```
print('\u4e2d\u6587')
```

```
中文
```

Python 常见的几种字符串

- b 字符串
- r 字符串
- f 字符串

2.1.1 b 字符串

由于 Python 的字符串类型是 str，在内存中以 Unicode 表示，一个字符对应若干个字节。如果要在网络上传输，或者保存到磁盘上，就需要把 str 变为以字节为单位的 bytes。

Python 对 bytes 类型的数据用带 b 前缀的单引号或双引号表示：

```
x = b'ABC'
```

要注意区分'ABC'和b'ABC'，前者是str，后者虽然内容显示得和前者一样，但 bytes 的每个字符都只占用一个字节。

```
print(type('ABC'))
print(type(b'ABC'))
```

```
<class 'str'>
<class 'bytes'>
```

以 Unicode 表示的 str 通过 encode() 方法可以编码为指定的 bytes，例如：

```
>>> 'ABC'.encode('ascii')
b'ABC'
>>> '中文'.encode('utf-8')
b'\xe4\xb8\xad\xe6\x96\x87'
>>> '中文'.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
ordinal not in range(128)
```

纯英文的 str 可以用 ASCII 编码为 bytes，内容是一样的，含有中文的 str 可以用 UTF-8 编码为 bytes。含有中文的 str 无法用 ASCII 编码，因为中文编码的范围超过了 ASCII 编码的范围，Python 会报错。

反过来，如果我们从网络或磁盘上读取了字节流，那么读到的数据就是 bytes。要把 bytes 变为 str，就需要用 decode() 方法：

```
>>> b'ABC'.decode('ascii')
'ABC'
>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
'中文'
```

如果 bytes 中包含无法解码的字节，decode() 方法会报错：

```
>>> b'\xe4\xb8\xad\xff'.decode('utf-8')
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 3: invalid
 start byte
```

如果 bytes 中只有一小部分无效的字节，可以传入 errors='ignore' 忽略错误的字节：

```
>>> b'\xe4\xb8\xad\xff'.decode('utf-8', errors='ignore')
'中'
```

要计算 str 包含多少个字符，可以用 len() 函数：

```
>>> len('ABC')
3
>>> len('中文')
2
```

len() 函数计算的是 str 的字符数，如果换成 bytes，len() 函数就计算字节数：

```
>>> len(b'ABC')
3
>>> len(b'\xe4\xb8\xad\xe6\x96\x87')
6
>>> len('中文'.encode('utf-8'))
6
```

由此可见，1 个中文字符经过 UTF-8 编码后通常会占用 3 个字节，而 1 个英文字符只占用 1 个字节。

在操作字符串时，我们经常遇到 str 和 bytes 的互相转换。为了避免乱码问题，应当始终坚持使用 UTF-8 编码对 str 和 bytes 进行转换。

由于 Python 源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为 UTF-8 编码。当 Python 解释器读取源代码时，为了让它按 UTF-8 编码读取，我们通常在文件开头写上这两行：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

- 第一行注释是为了告诉 Linux/OS X 系统，这是一个 Python 可执行程序，Windows 系统会忽略这个注释；
- 第二行注释是为了告诉 Python 解释器，按照 UTF-8 编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

2.2 字符串的格式化

%-formatting 这是 Python 格式化的 OG(original generation)，伴随着 python 语言的诞生。不建议使用%格式。

其使用方式类似于 C 语言，使用占位符来格式化字符串。如

```
print('Hello, %s!' % 'world')
name = 'Ming'
age = 23
print('Hello, %s. You are %s.' % (name, age))
```

```
Hello, world!
Hello, Ming. You are
23.
```

表 1: 常用占位符

占位符	替换内容
%d	整数
%f	浮点数
%s	字符串
%x	十六进制整数

请写出以下代码的运行结果

```
print('%3d-%03d' % (22, 11))
print('%.3f %8.3e' % (3.1415926, 4000.21))
```

但如果你使用多个参数和更长的字符串，\%-formatting 将变得不太好用。

```
first = 'Ming'
last = 'Li'
age = 23
profession = 'student'
affiliation = 'WHU'
print('Hello, %s %s. You are %s. You are a %s. You were a member of %s.'
      % (first, last, age, profession, affiliation))
```

```
Hello, Ming Li. You are 23. You are a student. You were a member of WHU.
```

这种格式过于冗长，不太好用。

str.format() 该格式是在 Python 2.6 中引入的。str.format()是对%-formatting的改进，它使用正常的函数调用语法，并且可以通过对要转换为字符串的对象的__format__()方法进行扩展。

使用str.format()，替换字段用大括号标记：

```
name = 'Ming'
age = 23
print('Hello, {}. You are {}.'.
format(name, age))
```

```
Hello, Ming. You are 23.
```

通过引用其索引来以任何顺序引用变量：

```
name = 'Ming'
age = 23
print('Hello, {1}. You are {0}.'.
format(age, name))
```

```
Hello, Ming. You are 23.
```

如果插入变量名称，则会获得额外的能够传递对象的权限，然后在大括号之间引用参数和方法：

```
person = {'name': 'Ming', 'age': 23}
print('Hello, {name}. You are {age}.'.format(name=person['name'], age=person['
age']))
```

```
Hello, Ming. You are 23.
```

也可以使用 ** 来用字典来完成这个巧妙的技巧：

```
person = {'name': 'Ming', 'age': 23}
print('Hello, {name}. You are {age}.'.format(**person))
```

```
Hello, Ming. You are 23.
```

使用str.format()的代码比使用\%-formatting 的代码更易读，但当处理多个参数和更长的字符串时，str.format()仍然可能非常冗长。

2.3 f 字符串

f 字符串从 Python 3.6 开始加入标准库，也被称为“格式化字符串文字 “。与其他格式化方式相比，它们不仅更易读，更简洁，不易出错，而且速度更快!

```
name = 'Ming'
age = 23
print(f'Hello, {name}. You are {age}.')
print(F'Hello, {name}. You are {age}.')
```

```
Hello, Ming. You are 23.
Hello, Ming. You are 23.
```

f 字符串是在运行时进行渲染的，因此可以将任何有效的 Python 表达式放入其中。

- 可做简单的计算

```
print(f'{2 * 37}')
```

```
74
```

- 可调用函数

```
name = 'Ming'
print(f'{name.lower()} is funny.')
```

```
ming is funny.
```

- 可以使用带有 f 字符串的类创建对象。

```
class Student:
    def __init__(self, first, last, age):
        self.first = first
        self.last = last
        self.age = age

    def __str__(self):
        return f'{self.first} {self.last} is {self.age}.'

    def __repr__(self):
        return f'{self.first} {self.last} is {self.age}. Surprise!'

std = Student('Ming', 'Li', 23)
```

```
print(f'{std}')
print(f'{std!r}')
```

```
Ming Li is 23.
Ming Li is 23. Surprise!
```

多行 f 字符串

```
name = 'Ming'
profession = 'student'
affiliation = 'WHU'
msg1 = (f"Hi {name}. "
        f"You are a {profession}. "
        f"You were in {affiliation}.")
msg2 = (f"Hi {name}. "
        "You are a {profession}. "
        "You were in {affiliation}.")
msg3 = f"""
Hi {name}.
You are a {profession}.
You were in {affiliation}.
"""
print(msg1)
print(msg2)
print(msg3)
```

```
Hi Ming. You are a student. You were in WHU.
Hi Ming. You are a {profession}. You were in {affiliation}.

Hi Ming.
You are a student.
You were in WHU.
```

3 列表 (list) 与元组 (tuple)

3.1 列表

列表是 Python 内置的一种数据类型，它是一种有序的集合，可以随时添加和删除其中的元素。

例 1. 列出所有课程，可用一个列表表示

```
courses = ['math', 'phys', 'chem']
print(courses)
```

```
['math', 'phys', 'chem']
```

- 用len()函数获取列表中元素的个数

```
print(len(courses))
```

```
3
```

- 用索引来访问列表中每一个位置的元素，记住索引从 0 开始：

```
print(courses[0])
print(courses[1])
print(courses[2])
# print(courses[3])
```

```
math
phys
chem
Traceback (most recent call last):
  File "list.py", line 5, in <module
  >
    print(courses[3])
IndexError: list index out of range
```

当索引超出了范围时，Python 会报一个 IndexError 错误，所以，要确保索引不要越界，记得最后一个元素的索引是len(classmates)- 1。

- 如果要取最后一个元素，除了计算索引位置外，还可以用-1 做索引，直接获取最后一个元素，用-2 做索引直接获取倒数第 2 个元素，以此类推：

```
print(courses[-1])
print(courses[-2])
print(courses[-3])
# print(courses[-4])
```

```
chem
phys
math
Traceback (most recent call last):
  File "list.py", line 5, in <module
  >
    print(courses[-4])
IndexError: list index out of range
```

- 列表是一个可变的有序表，可向列表中追加元素到末尾：

```
courses.append('biology')
print(courses)
```

```
['math', 'phys', 'chem', 'biology']
```

- 也可以把元素插入到指定的位置，比如索引号为 1 的位置：

```
courses.insert(1, 'history')
print(courses)
```

```
['math', 'history', 'phys', 'chem',
'biology']
```

- 要删除列表末尾的元素，用pop()方法：

```
courses.pop()
print(courses)
```

```
['math', 'history', 'phys', 'chem']
```

- 要删除指定位置的元素，用pop(i)方法，其中i为索引位置：

```
courses.pop(1)
print(courses)
```

```
['math', 'phys', 'chem']
```

- 要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

```
courses[1] = 'chinese'
print(courses)
```

```
['math', 'chinese', 'chem']
```

注 1. *1. 列表中元素的数据类型可以不同，如*

```
list1 = ['Apple', 123, 3.14, True]
print(len(list1))
```

```
4
```

2. 列表中元素也可以是另一个列表，如

```
list2 = ['python', 'java', ['c', 'c++', 'c#'], 'matlab']
print(len(list2))
```

```
4
```

3. 列表也可以为空，即空列表，如

```
list3 = []
print(len(list3))
```

```
0
```

3.2 元组 (tuple)

元组也是 Python 内置的一种数据类型，也是一种有序的集合。tuple 和 list 非常类似，但是 tuple 一旦初始化就不能修改，比如同样是列出课程名：

```
courses = ('math', 'phys', 'chem')
```

现在，courses是一个 tuple，它不能改变，没有append()，insert()这样的方法。其他获取元素的方法和 list 是一样的，你可以正常地使用courses[0]，courses[-1]，但不能赋值成另外的元素。

- 不可变的 tuple 有什么意义？因为 tuple 不可变，所以代码更安全。如果可能，能用 tuple 代替 list 就尽量用 tuple。
- tuple 的定义

```
tuple1 = (1, 2)
tuple2 = 1, 2
tuple3 = ()
tuple4 = (1, )
```

注 2. *定义只有一个元素的 tuple，不能这样做*

```
t = (1)
```

这样定义的不是 tuple，而是数1！这是因为括号()既可以表示 tuple，又可以表示数学公式中的小括号，这就产生了歧义。因此，Python 规定，这种情况下，按小括号进行计算，计算结果自然是 1。 于是，为消除歧义，定义只有一个元素的 tuple 时必须加上一个逗号,：

```
tuple4 = (1, )
```

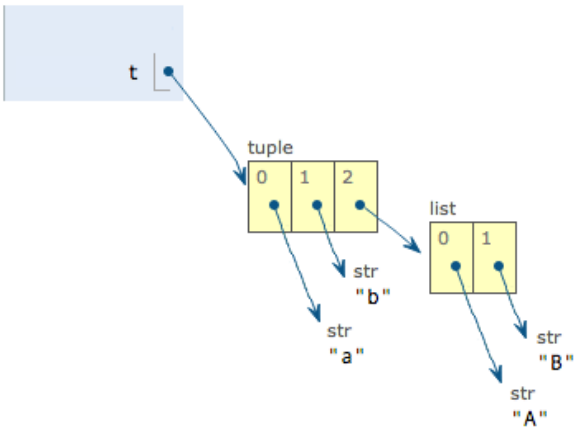
例 2. ”可变的“*tuple*

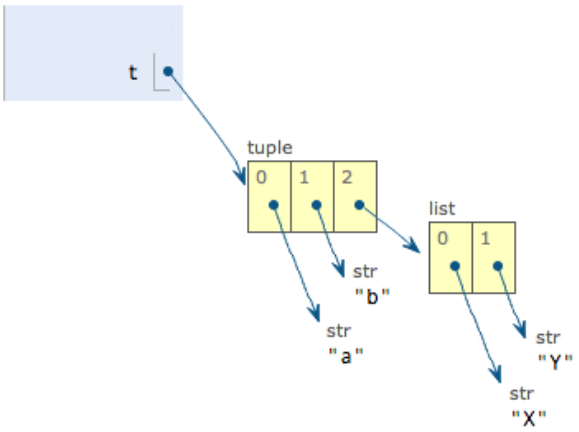
```
t = ('a', 'b', ['A', 'B'])
print(t)

t[2][0] = 'X'
t[2][1] = 'Y'
print(t)
```

```
('a', 'b', ['A', 'B'])
('a', 'b', ['X', 'Y'])
```

此时，元组t包含三个元素，分别是a，b和一个列表。





表面上看，tuple 的元素确实变了，但其实变的不是 tuple 的元素，而是 list 的元素。tuple 一开始指向的 list 并没有改成别的 list，所以，tuple 所谓的“不变”是说，tuple 的每个元素，指向永远不变。即指向’a’，就不能改成指向’b’，指向一个 list，就不能改成指向其他对象，但指向的这个 list 本身是可变的！

4 字典 (dict) 和集合 (set)

Python 内置了字典：dict 的支持，dict 全称 dictionary，在其他语言中也称为 map，使用键-值 (key-value) 存储，具有极快的查找速度。

例 3. 根据课程名查找相应的成绩。

- 如果用 *list* 实现，需要两个 *list:*

```
courses = ['math', 'phys', 'chem']
scores = [95, 90, 85]
```

给定一课程名，要查找相应的成绩，就先要在`courses`中找到对应的索引位置，再从`scores`中取出相应的成绩。显然，*list* 越长，耗时越多。

- 如果用 *dict* 实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。

```
info = {'math': 95, 'phys':
90, 'chem': 85}
print(info['math'])
```

```
95
```

为什么 dict 查找速度会很快？ 因为 dict 的实现原理和查字典是一样的。
假设字典包含了 1 万个汉字，我们要查某一个字，

- 第一种方法是把字典从第一页往后翻，直到找到我们想要的字为止，这种方法就是在 list 中查找元素的方法，list 越大，查找越慢。
- 第二种方法是先在字典的索引表里（比如部首表）查这个字对应的页码，然后直接翻到该页，找到这个字。无论找哪个字，这种查找速度都非常快，不会随着字典大小的增加而变慢。

dict 就是第二种实现方式，给定一个课程名，比如'math'，dict 在内部就可以直接计算出'math'对应的存放成绩的“页码”，也就是95这个数字存放的内存地址，直接取出来，所以速度非常快。对于这种 key-value 存储方式，在放进去的时候，必须根据 key 算出 value 的存放位置，这样，取的时候才能根据 key 直接拿到 value。

1. 把数据放入 dict 的方法，除了初始化时指定外，还可以通过 key 放入：

```
info['chinese'] = 88
print(info['chinese'])
```

```
88
```

2. 由于一个 key 只能对应一个 value，所以，多次对一个 key 放入 value，后面的值会把前面的值冲掉：

```
info['chinese'] = 94
print(info['chinese'])
```

```
94
```

3. 如果 key 不存在，dict 就会报错：

```
# print(info['history'])
```

```
Traceback (most recent call last):
  File "dict1.py", line 10, in <module>
    print(d['history'])
KeyError: 'history'
```

4. 要避免 key 不存在的错误，有两种办法：
   - 通过in判断key是否存在

```
print('history' in info)
```

```
False
```

   - 通过 dict 提供的get()方法，如果key不存在，可以返回None，或自己指定的 value：

```
print(info.get('history'))
print(info.get('history',
-1))
```

```
None
-1
```

5. 要删除一个 key，用pop(key)方法，对应的 value 也会从 dict 中删除：

```
info.pop('chem')
print(info)
```

```
{'math': 95, 'phys': 90, 'chinese': 94}
```

6. 请务必注意：dict 内部存放的顺序和 key 放入的顺序没有关系，也就是 dict 是无序的。

注 3. *和 list 相比，dict 有以下几个特点：*

- 查找和插入的速度极快，不会随着 *key* 的增加而变慢；
- 需要占用大量的内存，内存浪费多。

而 *list* 相反：

- 查找和插入的时间随着元素的增加而增加；
- 占用空间小，浪费内存很少。

dict 常用于需要高速查找的地方，在 Python 代码中几乎无处不在。正确使用 dict 非常重要，需要牢记的第一条就是 dict 的 key 必须是不可变对象。这是因为 dict 根据 key 来计算 value 的存储位置，如果每次计算相同的 key 得出的结果不同，那 dict 内部就完全混乱了。这个通过 key 计算位置的算法称为哈希算法（Hash）。

要保证 hash 的正确性，作为 key 的对象就不能变。在 Python 中，字符串、整数等都是不可变的，因此，可以放心地作为 key。而 list 是可变的，就不能作为 key：

```
d = {}
print(d)

key = [1, 2, 3]
d[key] = 'a list'
```

```
{}
Traceback (most recent call last):
  File "dict2.py", line 5, in <module>
    d[key] = 'a list'
TypeError: unhashable type: 'list'
```

4.1 集合 (set)

set 和 dict 类似，也是一组 key 的集合，但不存储 value。由于 key 不能重复，所以，在 set 中，没有重复的 key。

1. 创建一个 set，
   - 用花括号：

     ```
     s1 = {1, 2, 3}
     print(type(s1))
     print(s1)
     ```

     ```
     <class 'set'>
     {1, 2, 3}
     ```

   - 提供一个 list 作为输入集合：

     ```
     s2 = set([1, 2, 3])
     print(type(s2))
     print(s2)
     ```

     ```
     <class 'set'>
     {1, 2, 3}
     ```

     注意，传入的参数`[1, 2, 3]`是一个 list，而显示的`{1, 2, 3}`只是告诉你这个 set 内部有`1, 2, 3`这 3 个元素，显示的顺序也不表示 set 是有序的。

2. 重复元素在 set 中自动被过滤：

   ```
   s3 = {1, 1, 2, 2, 3, 3}
   print(s3)
   ```

   ```
   {1, 2, 3}
   ```

3. 通过add(key)方法可以添加元素到 set 中，可以重复添加，但不会有效果：

```
s3.add(4)
print(s3)
s3.add(4)
print(s3)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4}
```

4. 通过remove(key)方法可以删除元素：

```
s3.remove(4)
print(s3)
```

```
{1, 2, 3}
```

5. set 可以看成数学意义上的无序和无重复元素的集合，因此，两个 set 可以做数学意义上的交集、并集等操作：

```
s1 = {1, 2, 3}
s2 = set([2, 3, 4])
print(s1 & s2)
print(s1 | s2)
```

```
{2, 3}
{1, 2, 3, 4}
```

注 4. *set 和 dict 的唯一区别仅在于没有存储对应的 value，但是，set 的原理和 dict 一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证 set 内部“不会有重复元素”。*

4.2 再议不可变对象

str 是不变对象，而 list 是可变对象。

- 对可变对象 list 进行操作，其内部的内容会发生变化，如

```
a = ['c', 'b', 'a']
print(a)
a.sort()
print(a)
```

```
['c', 'b', 'a']
['a', 'b', 'c']
```

- 对不可变对象 str 进行操作?

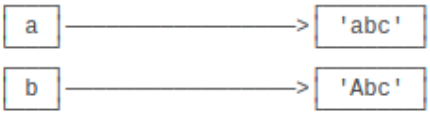
```
a = 'abc'
print(a.replace('a', 'A'))
print(a)
```

```
Abc
abc
```

请牢记，a是变量，'abc'才是字符串对象！我们经常说，对象a的内容是'abc'，但其实是指，a本身是一个变量，它指向的对象的内容才是'abc'。

当调用a.replace('a', 'A')时，实际上调用方法replace是作用在字符串对象'abc'上的，而这个方法虽然名字叫replace，但却没有改变字符串'abc'的内容。相反，replace方法创建了一个新字符串'Abc'并返回，

如果我们用变量b指向该新字符串，就容易理解了，变量a仍指向原有的字符串'abc'，但变量b却指向新字符串'Abc'了。



因此，对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回，这就保证了不可变对象本身永远是不可变的。

5 条件判断

5.1 if语句

语法

```
if condition:
  statements
```

例:

```
age = 20
if age >= 18:
    print(f'your age is {age}')
    print('adult')
```

```
your age is 20
adult
```

注 5. *根据 Python 的缩进原则，如果 if 语句判断是 True，就把缩进的两行 print 语句执行了，否则，什么也不做。*

5.2 if ... else ...语句

语法

```
if condition:
  statements1
else:
  statements2
```

例:

```
age = 12
if age >= 18:
    print(f'your age is {age}')
    print('adult')
else:
    print(f'your age is {age}')
    print('teenager')
```

```
your age is 12
teenager
```

5.3 if ... elif ... else ...语句

语法

```
if condition1:
  statements1
elif condition2:
  statements2
elif condition3:
  statements3
else:
  statements4
```

例:

```
age = 12
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
```

```
teenager
```

注 6. *if*语句执行有个特点，它是从上往下判断，如果在某个判断上是*True*，把该判断对应的语句执行后，就忽略掉剩下的*elif*和*else*.

请写出以下程序的输出结果：

```
age = 20
if age >= 6:
    print('teenager')
elif age >= 18:
    print('adult')
else:
    print('kid')
```

*if*判断条件可以简写，如

```
if x:
  print('True')
```

5.4 关于input

请运行以下程序

```
age = input('Enter your age: ')
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
```

如果输入20，则会报错：

```
Enter your age: 20
Traceback (most recent call last):
  File "input.py", line 2, in <module>
    if age >= 18:
TypeError: '>=' not supported between instances of 'str' and 'int'
```

这是因为input()返回的数据类型是str，str不能直接和int比较，必须先把str转换成int。

```
age = int(input('Enter your age: '))
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
```

```
Enter your age: 20
adult
```

5.5 习题

练习 1. 输入你的身高 *(m)* 与体重 *(kg)*，根据 *BMI* 公式 *(*即体重除以身高的平方*)*，计算 *BMI* 指数，并根据该指数判断你的体重水平：

- ≤ 18.5：过轻
- $18.5 - 25$：正常
- $25 - 28$：过重
- $28 - 32$：肥胖
- ≥ 32：严重肥胖

6 循环

Python 循环有两种：

- for ... in循环
- while循环

6.1 for ... in循环

for ... in循环可依次把 list 或 tuple 中的每个元素迭代出来。

例:

```
animals = ['dog', 'cat', 'monkey',
 'pig']

for animial in animals:
    print(animial)

print()
for i, animial in enumerate(
animals):
    print(i, animial)
```

```
dog
cat
monkey
pig

0 dog
1 cat
2 monkey
3 pig
```

例:
计算 1-100 的整数之和。

```
sum = 0
for x in range(101):
    sum += x
print(sum)
```

```
5050
```

6.2 while循环

例:
计算 1000 以内所有奇数之和。

```
sum = 0
n = 99
while n > 0:
    sum += n
    n -= 2
print(sum)
```

```
2500
```

6.3 break

在循环中，可使用break语句可以提前退出循环。

例:
输入q时，程序退出。

```
print('====================')
print('a. apple    b. banana')
print('o. orange   q. quit')
print('====================')

while True:
    letter = input('Enter a, b, o
    and q: ')
    if letter == 'a':
        print('apple')
    elif letter == 'b':
        print('banana')
    elif letter == 'o':
        print('orange')
    elif letter == 'q':
        break
print("Exit!")
```

```
====================
a. apple    b. banana
o. orange   q. quit
====================
Enter a, b, o and q: a
apple
Enter a, b, o and q: b
banana
Enter a, b, o and q: o
orange
Enter a, b, o and q: q
Exit!
```

6.4 continue

可用continue语句退出当前循环，直接进入下一个循环。

例:
计算 100 以内的所有奇数之和。

```
sum = 0
for i in range(100):
    if i % 2 == 0:
        continue
    sum += i
print(sum)
```

```
2500
```

7 函数

7.1 内置函数

Python 内置了很多有用的函数，我们可以直接调用。

例:

```
# abs
print(abs(1000))
print(abs(-20))
print(abs(12.34))
# print(abs('a'))

print(max(1, 2))
print(max(2, 3, -1, 5))
print(max([1, 2]))
print(max((2, 3, -1, 5)))
```

```
1000
20
12.34
2
5
2
5
```

Python 内置的常用函数还包括数据类型转换函数，比如 int() 可以把其他数据类型转换为整数：

例:

```
print(int('123'))
print(int(12.34))
print(float('12.34'))
print(str(1.23))
print(str(100))
print(bool(1))
print(bool(''))
```

```
123
12
12.34
1.23
100
True
False
```

例:

```
print(hex(255))
print(hex(1000))
print(oct(255))
print(oct(1000))
print(bin(255))
print(bin(1000))
```

```
0xff
0x3e8
0o377
0o1750
0b11111111
0b1111101000
```

注 7. 函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

```
a = abs
print(a(-1))
```

```
1
```

7.2 函数的定义

在 Python 中，定义一个函数要使用def语句，依次写出函数名、括号、括号中的参数和冒号:，然后，在缩进块中编写函数体，函数的返回值用return语句返回。

例:
自定义一个求绝对值的函数my_abs():

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x

if __name__ == '__main__':
    print(my_abs(-1))
```

```
20
```

例:
如果你把my_abs()的函数定义保存在my_functions.py文件中,可以使用from my_functions import my_abs来导入my_abs(),注意my_functions是文件名,不含.py扩展名。

```
from my_functions import my_abs
print(my_abs(-20))
```

```
20
```

7.2.1 空函数

如果想定义一个什么事也不做的空函数,可以用pass语句:

```
def empty():
    pass
```

pass语句什么都不做,那有什么用?实际上pass可以用来作为占位符,比如现在还没想好怎么写函数的代码,就可以先放一个pass,让代码能运行起来。

pass还可以用在其他语句里,比如:

```
if age >= 18:
    pass
```

缺少了pass,代码运行就会有语法错误。

7.2.2 参数检查

调用函数时,如果参数个数不对,Python 解释器会自动检查出来,并抛出TypeError:

```
>>> my_abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: my_abs() takes 1 positional argument but 2 were given
```

但是如果参数类型不对,Python 解释器就无法帮我们检查。试试my_abs和内置函数abs的差别:

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 2, in my_abs
TypeError: unorderable types: str() >= int()
>>> abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

当传入了不恰当的参数时，内置函数abs会检查出参数错误，而自定义的my_abs没有参数检查，会导致if语句出错，出错信息和abs不一样。所以，这个函数定义不够完善。

让我们修改一下my_abs的定义，对参数类型做检查，只允许整数和浮点数类型的参数。数据类型检查可以用内置函数isinstance()实现：

例:

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type'
        )
    if x >= 0:
        return x
    else:
        return -x

if __name__ == '__main__':
    print(my_abs(-1))
    print(my_abs('A'))
```

```
1
Traceback (most recent call
last):
  File "src/lec01/code/my_abs1.
  py", line 11, in <module>
    print(my_abs('A'))
  File "src/lec01/code/my_abs1.
  py", line 3, in my_abs
    raise TypeError('bad
    operand type')
TypeError: bad operand type
```

7.2.3 返回多个值

Python 函数允许返回多个值。

例:

给定平面空间中的一个点 (x, y)，设定位移和角度将其移动，计算新坐标：

```python
import math

def move(x, y, step, angle=0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny

if __name__ == '__main__':
    x, y = 100, 100
    step = 60
    angle = math.pi / 6
    x_new, y_new = move(x, y, step, angle)
    print(f"{x_new:.3f}, {y_new:.3f}")
```

```
151.962, 70.000
```

注 8. 请注意以下两点：

- *import math*语句表示导入*math*包，并允许后续代码引用*math*包里的*sin*、*cos*等函数。
- 事实上，函数返回多个值其实就是返回一个 *tuple*，这涉及到 *tuple* 的解压缩。

7.3 函数的参数

Python 的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

7.3.1 位置参数

例:
计算 x^2.

```python
def power(x):
    return x * x

print(power(5))
print(power(15))
```

```
25
225
```

对于power()函数，x就是一个位置参数。调用它时，必须传入有且仅有的一个参数x。

如果要计算 x^3 怎么办？可以再定义一个power3()函数。但如果要计算 $x^4, x^5, \cdots$ 怎么办？我们不可能定义无限多个函数。

例:
计算 x^n.

```
def power(x, n):
    s = 1
    while n > 0:
        n -= 1
        s *= x
    return s

print(power(5, 2))
print(power(5, 3))
```

```
25
125
```

修改后的power(x, n)函数有两个位置参数：x和n。调用函数时，传入的两个值按照位置顺序依次赋给参数x和n。

7.3.2 默认参数

新的power(x, n)函数定义没有问题，但是，旧的调用失败了，原因是我们增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

```
>>> power(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'n'
```

调用power()时缺少了一个位置参数n。

这个时候默认参数就派上用场了。由于经常计算 x^2，完全可以将第二个参数n的默认值设定为 2：

例:
计算 x^n.

```
def power(x, n=2):
    s = 1
    while n > 0:
        n -= 1
        s *= x
    return s

print(power(5))
print(power(5, 2))
```

```
25
25
```

- 调用power(5)时，相当于调用power(5, 2)；
- 而对于 $n \geq 2$ 的情形，则必须明确传入第二个参数，如power(5, 3)。

设置默认参数时，有几点要注意：

1. 位置参数在前，默认参数在后，否则 Python 解释器会报错；

2. 如何设置默认参数？当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处？最大的好处是简化函数的调用。

例:
编写一个小学一年级新生注册的函数，需要传入name和gender两个参数：

```
def enroll(name, gender):
    print(f"name: {name}")
    print(f"gender: {gender}")

enroll('Sarah', 'F')
```

```
name: Sarah
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。可以把年龄和城市设为默认参数：

例:

```
def enroll(name, gender, age=6,
city='Wuhan'):
    print(f"name: {name}")
    print(f"gender: {gender}")
    print(f"age: {age}")
    print(f"city: {city}")
    print('')

enroll('Sarah', 'F')
enroll('Bob', 'M', '7')
enroll('Adam', 'F', city='Beijing'
)
```

```
name: Sarah
gender: F
age: 6
city: Wuhan

name: Bob
gender: M
age: 7
city: Wuhan

name: Adam
gender: F
age: 6
city: Beijing
```

- 大部分学生注册时不需要提供年龄和城市，只需要提供必须的两个参数；
- 只有与默认参数不符的学生才需要提供额外的信息。

由此可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

如果存在多个默认参数，则在调用时，

- 既可以按顺序提供默认参数，如enroll('Bob', 'M', 7)，除位置参数name和gender外，最后一个参数应用在参数age上，参数city仍使用默认值；
- 也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。如enroll('Adam', 'M', city='Beijing')：参数city用传进去的值，其他默认参数仍使用默认值。

7.3.3 可变参数

在 Python 函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是 1 个、2 个到任意个，还可以是 0 个。

例:

给定一组数字 $a, b, c, \cdots$，计算 $a^2 + b^2 + c^2 + \cdots$。要定义该函数，必须先确定输入的参数。由于参数个数不确定，可把 $a, b, c, \cdots$ 作为一个 list 或 tuple 传进来，即

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum += n*n
    return sum

print(calc([1, 2, 3, 4]))
print(calc((1, 3, 5, 7)))
```

```
30
84
```

此时，必须先组装出一个 list 或 tuple 才能调用。

如果利用可变参数，调用方式可以简化：

例:

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum += n*n
    return sum

print(calc(1, 2, 3, 4))
print(calc(1, 3, 5, 7))
print(calc())
```

```
30
84
0
```

与上例相比，定义可变参数时仅需在参数前加上一个*号。在函数内部，参数numbers接收到的是一个 tuple，故函数代码完全不变。但是，调用函数时，可传入任意个参数，包括 0 个参数。

如果已经有一个 list 或者 tuple，要调用一个可变参数怎么办？可以这样做：

```
nums = [1, 2, 3]
print(calc(nums[0], nums[1], nums[2]))
```

不过这样做过于繁琐。事实上，Python 允许你在 list 或 tuple 前加一个*号，把 list 或 tuple 中的元素变成可变参数传进去：

```
print(calc(*nums))
```

这种写法非常有用，且很常见。

7.3.4 关键字参数

可变参数允许你传入 0 个或任意个参数，这些可变参数在函数调用时自动组装为一个 tuple。而关键字参数允许你传入 0 个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个 dict。

```
def person(name, age, **kw):
    print(f"name: {name}, age: {age
    }, other: {kw}")

person('Michael', 30)
person('Bob', 35, city='Wuhan')
person('Sarah', 22, gender='F',
city='Beijing')
```

```
name: Michael, age: 30, other: {}
name: Bob, age: 35, other: {'city':
 'Wuhan'}
name: Sarah, age: 22, other: {'
gender': 'F', 'city': 'Beijing'}
```

函数person()除了必选参数name和age外，还接受关键字参数kw。调用该函数时，

- 可以只传入必选参数，如person('Michael', 30)
- 也可传入任意个数的关键字参数,如person('Bob', 35, city='Beijing'),person('Adam', 45, gender ='M', job='Engineer')。

关键字参数有什么用？它可以扩展函数的功能。

- 在person()中，可保证接受参数name和age，但如果调用者愿意提供更多的参数，也能做到。
- 试想你要实现一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

和可变参数类似，也可以先组装出一个 dict，然后，把该 dict 转换为关键字参数传进去：

```
extra = {'city': 'Beijing', 'job':
'Engineer'}
person('Jack', 24, **extra)
```

```
name: Jack, age: 24, other: {'city'
: 'Beijing', 'job': 'Engineer'}
```

extra表示把extra这个 dict 的所有key-value用关键字参数传入到函数的kw参数，kw将获得一个dict。注意kw获得的 dict 是extra的一份拷贝，对kw的改动不会影响到函数外的extra。

7.3.5 命名关键字参数

如果要限制关键字参数的名字，就可以用命名关键字参数。

例:
如果只接收city和job作为关键字参数，可以这么定义函数

```
def person(name, age, *, gender,
city):
    print(f"name: {name}, age: {
    age}, gender: {gender}, city:
    {city}")

person('Sarah', 22, gender='F',
city='Beijing')
```

```
name: Sarah, age: 22, gender: F,
city: Beijing
```

和关键字参数**kw不同，命名关键字参数需要一个特殊分隔符*，*后面的参数被视为命名关键字参数。

如果函数定义中已经有了一个可变参数，则后面可直接命名关键字参数，而不需要特殊分隔符*：

```
def person(name, age, *args, gender, city):
    print(f"name: {name}, age: {age}, args: {args}, gender: {gender}, city: {
    city}")

person('Sarah', 22, 'Engineer', gender='F', city='Beijing')
```

```
name: Sarah, age: 22, args: ('Engineer',), gender: F, city: Beijing
```

命名关键字参数可以有缺省值，从而简化调用：

```
def person(name, age, *, gender='M', city):
    print(f"name: {name}, age: {age}, gender: {gender}, city: {city}")

person('Sarah', 22, city='Beijing')
```

```
name: Sarah, age: 22, gender: M, city: Beijing
```

7.3.6 参数组合

在 Python 中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这 5 种参数都可以组合使用。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数。

例:

```
def f1(a, b, c=0, *args, **kw):
    print(f'a = {a}, b = {b}, c = {c}, args: {args}, kw: {kw}')

def f2(a, b, c=0, *, d, **kw):
    print(f'a = {a}, b = {b}, c = {c}, d: {d}, kw: {kw}')
```

函数调用时，Python 解释器自动按照参数位置和参数名把对应的参数传进去。

```
f1(1, 2)
f1(1, 2, c=3)
f1(1, 2, 3, 'a', 'b', x=99)
f1(1, 2, d=99, ext=None)
```

```
a = 1, b = 2, c = 0, args: (), kw: {}
a = 1, b = 2, c = 3, args: (), kw: {}
a = 1, b = 2, c = 3, args: ('a', 'b'), kw: {'x': 99}
a = 1, b = 2, c = 0, args: (), kw: {'d': 99, 'ext': None}
```

最神奇的是通过一个 tuple 和 dict，你也可以调用上述函数：

```
args = (1, 2, 3, 4)
kw = {'d': 99, 'x': '#'}
f1(*args, **kw)

args = (11, 22, 33)
```

```
kw = {'d': 44, 'x': '##'}
f2(*args, **kw)
```

```
a = 1, b = 2, c = 3, args: (4,), kw: {'d': 99, 'x': '#'}
a = 11, b = 22, c = 33, d: 44, kw: {'x': '##'}
```

因此，对于任意函数，都可以通过类似func(*args, **kw)的形式调用它，无论它的参数是如何定义的。

7.4 习题

练习 2. 定义一个函数*quadratic(a, b, c)*，它接受 3 个参数，返回一元二次方程 $ax^2 + bx + c = 0$ 的两个根。

8 Python 高级特性

掌握了 Python 的数据类型、语句和函数，基本上就可以编写出很多有用的程序了。

例:
构造一个元素为1, 3, 5, 7, ..., 99的列表.

```
L = []
n = 1
while n <= 99:
    L.append(n)
    n += 2
```

```
L = [n for n in range(100) if n % 2 == 1]
```

但是，Python 代码不是越多越好，而是越少越好；不是越复杂越好，而是越简单越好。

基于这一思想，这一节来介绍一些 Python 中非常有用的高级特性，能一行代码实现的功能，绝不写五行。请始终牢记，代码越少，开发效率越高。

8.1 切片

取一个 list 或 tuple 的部分元素是非常常见的操作。

例:
给定一个 list

```
L = ['apple', 'banana', 'cherry', 'grape', 'peach']
```

如何取出其前三个元素?

- 笨办法

  ```
  r = [L[0], L[1], L[2]]
  print(r)
  ```

- 使用循环

```
r = []
for i in range(3):
    r.append(L[i])
print(r)
```

- 使用切片 (slice)

```
r = L[0:3]
```

例:
给定一个 list

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

通过切片操作取出其中的部分元素。

- 正向切片

```
print( L[1:3] )
print( L[:3]  )
print( L[1:]  )
print( L[:]   )
```

```
['b', 'c']
['a', 'b', 'c']
['b', 'c', 'd', 'e', 'f', 'g']
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

- 倒向切片

```
print( L[-4:-2] )
print( L[-4:]   )
print( L[:-2]   )
print( L[1:-1]  )
```

```
['d', 'e']
['d', 'e', 'f', 'g']
['a', 'b', 'c', 'd', 'e']
['b', 'c', 'd', 'e', 'f']
```

- 跳跃切片

```
print( L[1:-1:2] )
print( L[:-1:2] )
print( L[::5] )
```

```
['b', 'd', 'f']
['a', 'c', 'e']
['a', 'f']
```

例:
tuple 和 str 的切片

- tuple 的切片

```
t = (0, 1, 2, 3, 4, 5)
print( t[:3] )
print( ('a', 'b', 'c', 'd')[1::2] )
```

```
(0, 1, 2)
('b', 'd')
```

- str 的切片

```
print( 'abcdefgh'[:3] )
print( 'abcdefgh'[::3] )
```

```
abc
adg
```

在很多编程语言中，针对字符串提供了很多各种截取函数（例如，substring），其实目的就是对字符串切片。Python 没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

8.2 迭代

如果给定一个 list 或 tuple，我们可以通过for循环来遍历这个 list 或 tuple，这种遍历称为迭代（Iteration）。

只要是可迭代对象，无论有无下标，都可以迭代。

例:
dict 没有下标，但它也可以迭代：

```
d = {'a': 1, 'b': 2, 'c': 3}
for key in d:
    print(key)
```

```
a
b
c
```

因为 dict 的存储不是顺序排列的，所以迭代出的结果顺序可能会不一样。

例:
因 str 也是可迭代对象，它也可用for循环进行迭代：

```
for ch in 'abcd':
    print(ch)
```

```
a
b
c
d
```

当我们使用for循环时，只要作用于一个可迭代对象，for循环就可以正常运行，而我们不太关心该对象究竟是 list 还是其他数据类型。

8.2.1 如何判断一个对象是可迭代对象？

可通过 collections 模块的 Iterable 类型判断：

```
from collections.abc import Iterable
print( isinstance('abc', Iterable) )
print( isinstance([], Iterable) )
print( isinstance((), Iterable) )
print( isinstance({}, Iterable) )
print( isinstance({1}, Iterable) )
print( isinstance((x for x in range(10)),
Iterable) )
print( isinstance(123, Iterable) )
```

```
True
True
True
True
True
False
```

8.2.2 enumerate()函数

Python 内置的enumerate()可以把一个 list 变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

```
for i, value in enumerate(['a', 'b', 'c']):
    print(f"{i}: {value}")
```

```
0: a
1: b
2: c
```

```
for i, value in enumerate(('a', 'b', 'c')):
    print(f"{i}: {value}")
```

```
0: a
1: b
2: c
```

```
for i, value in enumerate('abc'):
    print(f"{i}: {value}")
```

```
0: a
1: b
2: c
```

以上代码中，for循环同时引用两个变量，这在 Python 中非常常见，如

```
for x, y in [[1, 1], [2, 4], [3, 9]]:
    print(f'{x}, {y}')
```

```
1, 1
2, 4
3, 9
```

8.3 列表生成式

列表生成式即 List Comprehensions，是 Python 内置的非常简单却强大的可以用来创建 list 的生成式。

例:
要生成列表[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]，可用list(range(1, 11))。但要生成[1, 4, 9, ..., 100]呢?

1. 用循环（过于繁琐）

```
L = []
for x in range(1, 11):
    L.append(x * x)
print(L)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

2. 列表生成式

```
L = [x * x for x in range(1, 11)]
print(L)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

写列表生成式时，for循环后还可以加上if判断。

例:
筛选出偶数的平方。

```
L = [x * x for x in range(1, 11)
if x % 2 == 0]
print(L)
```

```
[4, 16, 36, 64, 100]
```

还可以使用两层循环来得到全排列（三层及以上的循环很少用到）

例:

```
L = [m + n for m in 'ABC' for n in
 'XYZ']
print(L)
```

```
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', '
CX', 'CY', 'CZ']
```

8.3.1 列表生成式的应用

例:
利用os模块列出当前目录下的所有文件和目录名

```
dirs = [d for d in os.listdir('.')]
print(dirs)
```

```
['lec01.out', 'lec01.log', 'src', 'tmp.out', 'lec01.aux', 'lec01.tex', 'asset
', 'lec01.pdf', '.DS_Store', 'makefile']
```

例:
将一个 list 中的字符串变为小写

```
L = ['Hello', 'Wuhan', 'University']
l = [s.lower() for s in L]
print(l)
```

```
['hello', 'wuhan', 'university']
```

例:
将字典{'a': 1, 'b': 2, 'c': 3}转换为列表的形式['a=1', 'b=2', 'c=3']

```
d = {'a': 1, 'b': 2, 'c': 3}
l = [k + '=' + str(v) for k, v in d.items()]
print(l)
```

```
['a=1', 'b=2', 'c=3']
```

8.4 生成器

通过列表生成式，可以直接创建一个列表，而受到内存限制，列表容量肯定是有限的。假设创建了一个包含 100 万个元素的列表，而仅需访问前面几个元素，这不仅占用很大的存储空间，并且后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的 list，从而节省大量的空间。在 Python 中，这种一边循环一边计算的机制，称为生成器：generator。

8.4.1 generator 的创建

- 把列表生成式的[]改为()，即可创建一个 generator:

```
L = [x *x for x in range(3)]
print(L)
```

```
G = (x *x for x in range(3))
print(G)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
<generator object <genexpr> at 0x7fd6a02f26d8>
```

创建L和G的区别仅在于最外层的[]和()，L是一个 list，而G是一个 generator。

- 使用关键字yield. 例如，生成斐波拉契数列可以这么做

```
def fib(n):
    i, a, b = 0, 0, 1
    while i < n:
        yield b
        a, b = b, a + b
        i = i + 1
f = fib(6)
print(f)
```

```
<generator object fib at 0x7fc4e6b576d8>
```

如果一个函数中包含关键字yield，则这个函数就不再是一个普通函数，而是一个 generator.

8.4.2 generator 的使用

- 使用next.

```
print(next(G))
print(next(G))
print(next(G))
print(next(G))
```

```
0
1
4
Traceback (most recent call last):
  File "src/lec01/code/generator.py", line 9, in <module>
    print(next(G))
StopIteration
```

这种方式过于复杂，正确的使用方式是for ... in循环，因为 generator 也是可迭代对象。

- 使用for ... in循环.

```
for i in fib(4):
    print(i)
```

```
1
1
2
3
```

注意，generator 和 function 的执行流程不一样。

- function 是顺序执行，遇到`return`语句或者最后一行语句就返回。
- 变成 generator 的 function，在每次调用`next()`时执行，遇到`yield`语句返回，再次执行时从上次返回的`yield`语句处继续执行。

8.5 可迭代对象 (Iterable) 与迭代器 (Iterator)

可以直接作用于 for 循环的数据类型有以下几种：

- 集合数据类型，如 list、tuple、dict、set、str 等；
- generator，包括生成器和带 yield 的 generator function。

这些可以直接作用于 for 循环的对象统称为可迭代对象 (Iterable)。可使用`isinstance()`来判断一个对象是否为`Iterable`对象。

```
from collections.abc import Iterable
print( isinstance('abc', Iterable) )
print( isinstance([], Iterable) )
print( isinstance((), Iterable) )
print( isinstance({}, Iterable) )
print( isinstance({1}, Iterable) )
print( isinstance((x for x in range(10)), Iterable) )
print( isinstance(123, Iterable) )
```

```
True
True
True
True
True
True
False
```

可使用`next()`函数调用并不断返回下一个值的对象称为迭代器 (Iterator). 可使用`isinstance()`来判断一个对象是否为`Iterator`对象。

```
from collections.abc import Iterator
print(isinstance([], Iterator))
print(isinstance({}, Iterator))
print(isinstance('abc', Iterator))
print(isinstance((x for x in range(5)), Iterator))
```

```
False
False
False
True
```

生成器都是 Itertor 对象，但 list、dict、str 虽然是 Iterable 对象，却不是 Iterator。可使用`iter()`函数将 list、dict、str 等 Iterable 对象变成 Iterator.

```
print(isinstance(iter([]), Iterator))
print(isinstance(iter('abc'), Iterator))
```

```
True
True
```

为什么 list、dict、str 等数据类型不是 Iterator 呢?

因为 Python 的 Iterator 对象表示的是一个数据流，它可以被`next()`函数调用并不断返回下一个数据，直到没有数据时抛出 StopIteration 错误。可以把这个数据流看做是一个有序序列，但不能提前知道其长度，只能不断通过`next()`函数来按需计算下一个数据。因此，Iterator 的计算是惰性的，只有在需要返回下一个数据时它才计算。Iterator 甚至可以是一个无限大的数据流，如全体自然数；而使用 list 则不可能存储无限个元素。

小结

- 凡是可作用于 for 循环的对象都是 Iterable 类型；
- 凡是可作用于 next() 函数的对象都是 Iterator 类型，它们表示一个惰性计算的序列。

集合数据类型如 list、dict、str 等是 Iterable 但不是 Iterator，不过可以通过 iter() 函数获得一个 Iterator 对象。

8.6 习题

1. 编写函数`min_and_max(L)`，求一个 list 的最小值和最大值，并以 tuple 将它们返回。
2. 编写一个 generator，生成杨辉三角。

9 函数式编程

通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计，而函数就是面向过程的程序设计的基本单元。而函数式编程 (Functional Programming)，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

9.1 高阶函数

- 变量可以指向函数

```
print(abs(-10))
print(abs)
x = abs(-10)
```

```
print(x)
f = abs
print(f(-10))
```

```
10
<built-in function abs>
10
10
```

注意，

1. 函数本身也是对象，可以赋值给变量，即变量可以指向函数。
2. 变量f指向了abs函数本身，故直接调用abs()和调用变量f()完全相同。

- 函数名也是变量

  函数名是什么呢？函数名其实就是指向函数的变量！

  对于abs()，完全可以把函数名abs看成变量，它指向一个可以计算绝对值的函数！如果把abs指向其他对象，会有什么情况发生？

```
print(abs)
abs = 10
print(abs(-10))
```

```
10
Traceback (most recent call last):
  File "src/lec01/code/hof.py", line 10, in <module>
    print(abs(-10))
TypeError: 'int' object is not callable
```

- 传入函数

  既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

```
def add(x, y, f):
    return f(x) + f(y)
x = -5
y = 6
print(add(x, y, abs))
```

```
11
```

9.1.1 map/reduce

map() 它接收两个参数，一个是函数，一个是 Iterable，map 将传入的函数依次作用到序列的每个元素，并把结果作为新的 Iterator 返回。

例:
将一个函数 $f(x) = x^2$ 作用到一个列表

```
def f(x):
    return x * x
L = [i+1 for i in range(9)]
r = map(f, L)
print(r)
print(list(r))
```

```
<map object at 0x7fb0e83707b8>
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

例:
把一个列表中的数字转为字符串

```
r = list(map(str, L))
print(r)
```

```
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

reduce() 它把一个函数作用在一个序列[x1, x2, x3, ...]上，该函数必须接收两个参数，reduce() 把结果继续和序列的下一个元素做累积计算，其效果为

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

例:
对一个序列求和

```
from functools import reduce
def add(x, y):
    return x + y
L = [1, 3, 5, 7, 9]
print(reduce(add, L))
```

```
25
```

例:
将序列[1, 3, 5, 7, 9]转换为整数13579

```
def f(x, y):
    return x * 10 + y
print(reduce(f, [1, 3, 5, 7, 9]))
```

```
13579
```

例:
将字符串'13579'转换为整数13579

```
def f(x, y):
    return x * 10 + y
def char2num(ch):
    digits = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4,
              '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}
    return digits(ch)
s = '13579'
print(reduce(f, map(char2num, s)))
```

```
13579
```

9.1.2 filter

filter() 函数用于过滤序列，它接收一个函数和一个序列，把传入的函数依次作用于每个元素，并根据返回值是True还是False决定保留还是丢弃该元素。

例:
在一个 list 中，删掉偶数，只保留奇数。

```
def is_odd(n):
    return n % 2 == 1
L = [1, 2, 4, 5, 6, 9, 11, 12]
print(list(filter(is_odd, L)))
```

```
[1, 5, 9, 11]
```

例:
把一个序列中的空字符删除

```
def not_empty(s):
    return s and s.strip()
L = ['A', '', 'B', None, 'C', '  ']
print(list(filter(not_empty, L)))
```

```
['A', 'B', 'C']
```

9.1.3 sorted

排序是编程中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。若为数字，则可直接比较，但如果是字符串或者两个 dict 呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。

Python 内置的 sorted() 函数可以直接对 list 进行排序。

```
L = [36, 5, -12, 9, -21]
print(sorted(L))
```

```
[-21, -12, 5, 9, 36]
```

sorted() 函数也是一个高阶函数，它还可以接收一个 key 函数来实现自定义的排序。

例:
对一个列表按绝对值大小排序

```
L = [36, 5, -12, 9, -21]
print(sorted(L, key=abs))
```

```
[5, 9, -12, -21, 36]
```

例:
字符串排序

- 默认情形，按 ASCII 码的大小比较

```
L = ['bob', 'about', 'Zoo', 'Credit']
print(sorted(L))
```

```
['Credit', 'Zoo', 'about', 'bob']
```

- 忽略大小写的排序

```
L = ['bob', 'about', 'Zoo', 'Credit']
print(sorted(L, key=str.lower))
```

```
['about', 'bob', 'Credit', 'Zoo']
```

- 反向排序

```
L = ['bob', 'about', 'Zoo', 'Credit']
print(sorted(L, key=str.lower, reverse=True))o
```

```
['Zoo', 'Credit', 'bob', 'about']
```