PYTHON

阿里云 | TIANCHI天池

# 阿里云天池龙珠计划

ALIBABA CLOUD TIANCHI DRAGON BALL PROJECT

免费CPU/GPU算力　　完整学练赛体系

天池TOP选手答疑　　天池认证奖励

# PYTHON训练营

12天　每天一小时　开启编程人生


天池龙珠计划是阿里云天池平台针对人工智能初学者提供的完整入门学习体系，其目标是为学习者奠定人工智能知识体系的学习基础，通过理论学习+项目实践的方式构建学习者对人工智能的整体认知。

## 学习路径

体系化学习内容

学习体系提供3大基础训练营，3大进阶训练营，5大应用学习赛，上百道测试题以及竞赛资源。

## DSW算力

随时随地在线编程

为每一位学习者提供在线编程实践环境，可支持随时随地代码实践，充分降低学习门槛。

## Kol带学

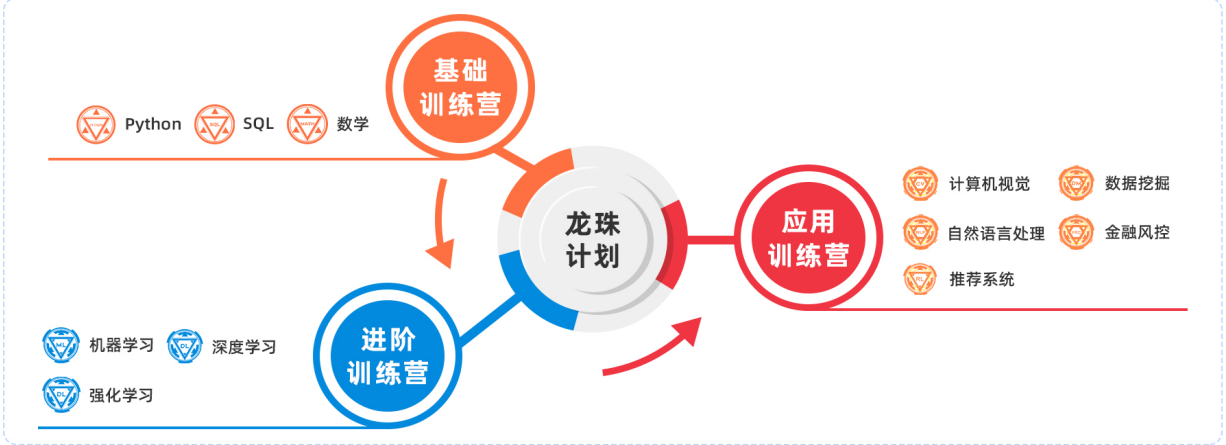
天池Kol在线答疑

天池Top选手分享学习经验，为学习者答疑解惑。社区构建完整的用户体系，面向所有学习者开放。

## 学习奖励

阿里云官方结业证书

每个训练营提供训练营挑战证书，通关六大训练营得龙珠计划系列勋章和纸质证书。




本学习体系由天池与开源学习组织Datawhale共同维护

天 池 龙 珠 计 划

# Python训练营

阿里云天池龙珠计划之Python训练营是为学习者准备的python零基础入门训练营，总计19小节，包含以下4个任务：

**Task01** Python基础知识介绍，包含变量、位运算、条件语句和异常处理。

**Task02** Python中的6大数据结构，包括列表、元祖、字符串、字典、集合、序列。

**Task03** Python中的函数、lambda表达式、类与对象、魔法方法。

**Task04** 数据分析实战项目，从0开始动手（可以设计成一个流程）。

经过整个学习，学习者可以掌握Python基础，并且对于Python能做什么有一个基本的概念，本训练营从Python基础到数据分析实战的学习路径，能让学习者更快的走向人工智能之路。

主办方

TIANCHI天池 大数据众智平台

合作方

Datawhale

扫码回复：Python答案，获取练习题答案

扫码回复：Python训练营，即可进行学习获取证书

# 目录

# 1 Python介绍

## 1.1 学了Python你能做什么？

1. 数据分析挖掘

   利用Python基础库，如Numpy、Pandas与可视化Matplotlib等等库实现对数据分析挖掘

   a. Kaggle入门：泰坦尼克号幸存者

   b. 电影人物关系提取

   c. 出租车与网约车调度

   d. 租房问题

   e. NBA比赛结果预测

2. 机器学习与深度学习

   利用机器学习算法和深度学习算法解决问题

   a. 人脸识别

   b. K-近邻算法实现手写数字识别

   c. 中文错别字高亮系统

   d. 街边字符识别

3. 网络开发

   利用Python网络框架，如Flask、Django及异步框架Toronto等等实现网站开发

   a. Django搭建个人博客

   b. Flask实现简易聊天室

   c. 信息管理系统

4. 爬虫

   利用Python基础库，如Request库及相应提取文本方式获取目标信息

   a. 微信好友信息批量获取

   b. 微博热搜

   c. 爬取知乎图片

   d. 爬取天气预报

   e. 网易云音乐

## 1.2 入门体验小游戏：猜数字游戏

题目描述:

电脑产生一个零到100之间的随机数字，然后让用户来猜，如果用户猜的数字比这个数字大，提示太大，否则提示太小，当用户正好猜中电脑会提示，"恭喜你猜到了这个数是....."。在用户每次猜测之前程序会输出用户是第几次猜测，如果用户输入的根本不是一个数字，程序会告诉用户"输入无效".

(尝试使用try catch异常处理结构对输入情况进行处理)

获取随机数采用random模块。

```
import random
guess=random.randint(1,101)

i=1
while True:
    print ("第%d次猜，请输入一个整数数字："%(i))
    try:
        temp=int(input())
        i+=1
    except ValueError :
        print ("输入无效")
        continue
    if  temp==guess:
        print ("恭喜你猜对了，就是这个数",guess)
        break;
    elif (temp>guess):
        print ("大了")
    elif (temp<guess):
        print ("小了")
```

# 2 变量、运算符与数据类型

## 2.1 注释

1. 在 Python 中，`#` 表示注释，作用于整行。

【例子】单行注释

```
# 这是一个注释
print("Hello world")

# Hello world
```

1. `''' '''` 或者 `""" """` 表示区间注释，在三引号之间的所有内容被注释

【例子】多行注释

```
'''
这是多行注释，用三个单引号
这是多行注释，用三个单引号
这是多行注释，用三个单引号
'''
print("Hello china")
# Hello china

"""
这是多行注释，用三个双引号
这是多行注释，用三个双引号
这是多行注释，用三个双引号
"""
print("hello china")
# hello china
```

## 2.2 运算符

算术运算符

| 操作符 | 名称 | 示例 |
| --- | --- | --- |
| + | 加 | 1+1 |
| - | 减 | 2-1 |
| * | 乘 | 3*4 |
| / | 除 | 3/4 |
| // | 整除（地板除） | 3//4 |
| % | 取余 | 3%4 |
| ** | 幂 | 2**3 |

【例子】

```
print(3 % 2)  # 1
print(11 / 3)  # 3.6666666666666665
print(11 // 3)  # 3
print(2 ** 3)  # 8
```

比较运算符

| 操作符 | 名称 | 示例 |
| --- | --- | --- |
| > | 大于 | 2>1 |
| >= | 大于等于 | 4>=2 |
| < | 小于 | 1>2 |
| <= | 小于等于 | 2<=5 |
| == | 等于 | 3==3 |
| != | 不等于 | 3!=5 |

【例子】

```
print(1 > 3)  # False
print(2 < 3)  # True
print(1 == 1)  # True
print(1 != 1)  # False
```

逻辑运算符

| 操作符 | 名称 | 示例 |
| --- | --- | --- |
| and | 与 | (2>1) and (3>7) |
| or | 或 | (1>3) or (2<9) |

| | | |
|---|---|---|
| `not` | 非 | `not(2>1)` |

【例子】

```
print((3 > 2) and (3 < 5))  # True
print((1 > 3) and (2 < 1))  # False
print((1 > 3) or (3 < 5))  # True
```

位运算符

| 操作符 | 名称 | 示例 |
|---|---|---|
| `~` | 按位取反 | ~4 |
| `&` | 按位与 | 4 & 5 |
| `\|` | 按位或 | `4 \| 5` |
| `^` | 按位异或 | 4 ^ 5 |
| `<<` | 左移 | 4 << 2，表示整数 4 按位左移 2 位 |
| `>>` | 右移 | 4 >> 2，表示整数 4 按位右移 2 位 |

按位非操作 ~

```
~ 1 = 0
~ 0 = 1
```

按位与操作 &

```
1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0
```

按位或操作 |

```
1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0
```

按位异或操作 ^

```
1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 1 = 1
0 ^ 0 = 0
```

按位左移操作 <<

【例子】`num << i` 将`num`的二进制表示向左移动`i`位所得的值。

```
00 00 10 11 -> 11
11 << 3
---
01 01 10 00 -> 88
```

按位右移操作 >>

【例子】`num >> i` 将`num`的二进制表示向右移动`i`位所得的值。

```
00 00 10 11 -> 11
11 >> 2
---
00 00 00 10 -> 2
```

三元运算符

```python
x, y = 4, 5
if x < y:
    small = x
else:
    small = y

print(small)  # 4
```

有了这个三元操作符的条件表达式，你可以使用一条语句来完成以下的条件判断和赋值操作。

【例子】

```python
x, y = 4, 5
small = x if x < y else y
print(small)  # 4
```

其他运算符

| 操作符 | 名称 | 示例 |
| --- | --- | --- |
| `is` | 是 | 'hello' is 'hello' |
| `not is` | 不是 | 3 is not 5 |
| `in` | 存在 | 5 in [1, 2, 3, 4, 5] |
| `not in` | 不存在 | 2 not in [1, 2, 3, 4, 5] |

【例子】

```python
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
if 'A' in letters:
    print('A' + ' exists')
if 'h' not in letters:
    print('h' + ' not exists')

# A exists
# h not exists
```

【例子】比较的两个变量均指向不可变类型

```python
a = "hello"
b = "hello"
print(a is b, a == b)
# True True
```

【例子】比较的两个变量均指向可变类型

```python
a = ["hello"]
b = ["hello"]
print(a is b, a == b)
# False True
```

注意：

1. is, is not 对比的是两个变量的内存地址
2. ==, != 对比的是两个变量的值

即：

1. 假如比较的两个变量，指向的都是地址不可变的类型（str等），那么is，is not 和 ==，！= 是完全等价的。
2. 假如对比的两个变量，指向的是地址可变的类型（list，dict，tuple等），则两者是有区别的。

运算符的优先级

1. 一元运算符优于二元运算符。如正负号。
2. 先算术运算，后移位运算，最后位运算。例如 1 << 3 + 2 & 7等价于 (1 << (3 + 2)) & 7
3. 逻辑运算最后结合

【例子】

```python
print(-3 ** 2)  # -9
print(3 ** -2)  # 0.1111111111111111
print(-3 * 2 + 5 / -2 - 4)  # -12.5
print(3 < 4 and 4 < 5)  # True
```

## 2.3 变量和赋值

1. 在使用变量之前，需要对其先赋值。
2. 变量名可以包括字母、数字、下划线、但变量名不能以数字开头。
3. Python 变量名是大小写敏感的，foo != Foo。

【例子】

```
teacher = "小马的程序人生"
print(teacher)  # 小马的程序人生
teacher = "老马的程序人生"
print(teacher)  # 老马的程序人生
```

【例子】

```
first = 2
second = 3
third = first + second
print(third)  # 5
```

【例子】

```
myTeacher = "老马的程序人生"
yourTeacher = "小马的程序人生"
ourTeacher = myTeacher + yourTeacher
print(ourTeacher)  # 老马的程序人生小马的程序人生
```

---

## 2.4 数据类型与转换

| 类型 | 名称 | 示例 |
| --- | --- | --- |
| int | 整型 | -876, 10 |
| float | 浮点型 | 3.149, 11.11 |
| bool | 布尔型 | True, False |

整型 `<class 'int'>`

【例子】

```python
a = 1031
print(a, type(a))

# 1031 <class 'int'>
```

通过 `print` 可看出 `a` 的值，以及类 (class) 是 `int`。

Python 里面万物皆对象（object），整型也不例外，只要是对象，就有相应的属性 （attributes） 和方法（methods）。

```python
b = dir(int)
print(b)

# ['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
# '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__',
# '__float__', '__floor__', '__floordiv__', '__format__', '__ge__',
# '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
# '__index__', '__init__', '__init_subclass__', '__int__', '__invert__',
# '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__',
# '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
# '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
# '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
# '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
# '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
# '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
# 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
# 'numerator', 'real', 'to_bytes']
```

对它们有个大概印象就可以了，具体怎么用，需要哪些参数 （argument），还需要查文档。看个`bit_length`的例子。

【例子】找到一个整数的二进制表示，再返回其长度。

```python
a = 1031
print(bin(a))  # 0b10000000111
print(a.bit_length())  # 11
```

浮点型`<class 'float'>`

【例子】

```python
print(1, type(1))
# 1 <class 'int'>

print(1., type(1.))
# 1.0 <class 'float'>

a = 0.00000023
b = 2.3e-7
print(a)  # 2.3e-07
print(b)  # 2.3e-07
```

有时候我们想保留浮点型的小数点后 `n` 位。可以用 `decimal` 包里的 `Decimal` 对象和 `getcontext()` 方法来实现。

```
import decimal
from decimal import Decimal
```

Python 里面有很多用途广泛的包 (package)，用什么你就引进 (import) 什么。包也是对象，也可以用上面提到的`dir(decimal)` 来看其属性和方法。比如 `getcontext()` 显示了 `Decimal` 对象的默认精度值是 28 位 (`prec=28`)，展示如下：

```
a = decimal.getcontext()
print(a)

# Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=
[], traps=[InvalidOperation, DivisionByZero, Overflow])
```

让我们看看 1/3 的保留 28 位长什么样？

```
b = Decimal(1) / Decimal(3)
print(b)

# 0.3333333333333333333333333333
```

那保留 4 位呢？用 `getcontext().prec` 来调整精度。

```
decimal.getcontext().prec = 4
c = Decimal(1) / Decimal(3)
print(c)

# 0.3333
```

布尔型`<class 'bool'>`

布尔 (boolean) 型变量只能取两个值，`True` 和 `False`。当把布尔变量用在数字运算中，用 `1` 和 `0` 代表 `True` 和 `False`。

【例子】

```
print(True + True)  # 2
print(True + False)  # 1
print(True * False)  # 0
```

除了直接给变量赋值 `True` 和 `False`，还可以用 `bool(X)` 来创建变量，其中 `X` 可以是

1. 基本类型：整型、浮点型、布尔型
2. 容器类型：字符、元组、列表、字典和集合

【例子】`bool` 作用在基本类型变量：`X` 只要不是整型 `0`、浮点型 `0.0`，`bool(X)` 就是 `True`，其余就是 `False`。

```python
print(type(0), bool(0), bool(1))
# <class 'int'> False True

print(type(10.31), bool(0.00), bool(10.31))
# <class 'float'> False True

print(type(True), bool(False), bool(True))
# <class 'bool'> False True
```

【例子】`bool` 作用在容器类型变量：`X` 只要不是空的变量，`bool(X)` 就是 `True`，其余就是 `False`。

```python
print(type(''), bool(''), bool('python'))
# <class 'str'> False True

print(type(()), bool(()), bool((10,)))
# <class 'tuple'> False True

print(type([]), bool([]), bool([1, 2]))
# <class 'list'> False True

print(type({}), bool({}), bool({'a': 1, 'b': 2}))
# <class 'dict'> False True

print(type(set()), bool(set()), bool({1, 2}))
# <class 'set'> False True
```

确定`bool(X)` 的值是 `True` 还是 `False`，就看 `X` 是不是空，空的话就是 `False`，不空的话就是 `True`。

1. 对于数值变量，`0`，`0.0` 都可认为是空的。
2. 对于容器变量，里面没元素就是空的。

获取类型信息

1. 获取类型信息 `type(object)`

【例子】

```python
print(type(1))  # <class 'int'>
print(type(5.2))  # <class 'float'>
print(type(True))  # <class 'bool'>
print(type('5.2'))  # <class 'str'>
```

1. 获取类型信息 `isinstance(object, classinfo)`

【例子】

```python
print(isinstance(1, int))  # True
print(isinstance(5.2, float))  # True
print(isinstance(True, bool))  # True
print(isinstance('5.2', str))  # True
```

注：

1. `type()` 不会认为子类是一种父类类型，不考虑继承关系。
2. `isinstance()` 会认为子类是一种父类类型，考虑继承关系。

如果要判断两个类型是否相同推荐使用 `isinstance()`。

**类型转换**

1. 转换为整型 `int(x, base=10)`
2. 转换为字符串 `str(object='')`
3. 转换为浮点型 `float(x)`

【例子】

```python
print(int('520'))  # 520
print(int(520.52))  # 520
print(float('520.52'))  # 520.52
print(float(520))  # 520.0
print(str(10 + 10))  # 20
print(str(10.1 + 5.2))  # 15.3
```

## 2.5 print() 函数

```python
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

1. 将对象以字符串表示的方式格式化输出到流文件对象file里。其中所有非关键字参数都按 `str()` 方式进行转换为字符串输出；
2. 关键字参数 `sep` 是实现分隔符，比如多个参数输出时想要输出中间的分隔字符；
3. 关键字参数 `end` 是输出结束时的字符，默认是换行符 `\n`；
4. 关键字参数 `file` 是定义流输出的文件，可以是标准的系统输出 `sys.stdout`，也可以重定义为别的文件；
5. 关键字参数 `flush` 是立即把内容输出到流文件，不作缓存。

【例子】没有参数时，每次输出后都会换行。

```
shoplist = ['apple', 'mango', 'carrot', 'banana']
print("This is printed without 'end'and 'sep'.")
for item in shoplist:
    print(item)

# This is printed without 'end'and 'sep'.
# apple
# mango
# carrot
# banana
```

【例子】每次输出结束都用`end`设置的参数`&`结尾，并没有默认换行。

```
shoplist = ['apple', 'mango', 'carrot', 'banana']
print("This is printed with 'end='&''.")
for item in shoplist:
    print(item, end='&')
print('hello world')

# This is printed with 'end='&''.
# apple&mango&carrot&banana&hello world
```

【例子】，`item`值与`'another string'`两个值之间用`sep`设置的参数`&`分割。由于`end`参数没有设置，因此默认是输出解释后换行，即`end`参数的默认值为`\n`。

```
shoplist = ['apple', 'mango', 'carrot', 'banana']
print("This is printed with 'sep='&''.")
for item in shoplist:
    print(item, 'another string', sep='&')

# This is printed with 'sep='&''.
# apple&another string
# mango&another string
# carrot&another string
# banana&another string
```

---

**参考文献：**


1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443
3. https://mp.weixin.qq.com/s/DZ589xEbOQ2QLtiq8mP1qQ
4. https://www.cnblogs.com/OliverQin/p/7781019.html


---

**练习题：**

1. 怎样对python中的代码进行注释？

2. python有哪些运算符，这些运算符的优先级是怎样的？

3. python 中 `is`, `is not` 与 `==`, `!=` 的区别是什么？

4. python 中包含哪些数据类型？这些数据类型之间如何转换？

# 3 位运算

## 3.1 原码、反码和补码

二进制有三种不同的表示形式：原码、反码和补码，计算机内部使用补码来表示。

原码：就是其二进制表示（注意，有一位符号位）。

```
00 00 00 11 -> 3
10 00 00 11 -> -3
```

反码：正数的反码就是原码，负数的反码是符号位不变，其余位取反（对应正数按位取反）。

```
00 00 00 11 -> 3
11 11 11 00 -> -3
```

补码：正数的补码就是原码，负数的补码是反码+1。

```
00 00 00 11 -> 3
11 11 11 01 -> -3
```

符号位：最高位为符号位，0表示正数，1表示负数。在位运算中符号位也参与运算。

## 3.2 按位非操作 ~

```
~ 1 = 0
~ 0 = 1
```

`~` 把 `num` 的补码中的 0 和 1 全部取反（0 变为 1，1 变为 0）有符号整数的符号位在 `~` 运算中同样会取反。

```
00 00 01 01 -> 5
~
---
11 11 10 10 -> -6

11 11 10 11 -> -5
~
---
00 00 01 00 -> 4
```

## 3.3 按位与操作 &

```
1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0
```

只有两个对应位都为 1 时才为 1

```
00 00 01 01 -> 5
&
00 00 01 10 -> 6
---
00 00 01 00 -> 4
```

## 3.4 按位或操作 |

```
1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0
```

只要两个对应位中有一个 1 时就为 1

```
00 00 01 01 -> 5
|
00 00 01 10 -> 6
---
00 00 01 11 -> 7
```

## 3.5 按位异或操作 ^

```
1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 1 = 1
0 ^ 0 = 0
```

只有两个对应位不同时才为 1

```
00 00 01 01 -> 5
^
00 00 01 10 -> 6
---
00 00 00 11 -> 3
```

异或操作的性质：满足交换律和结合律

```
A: 00 00 11 00
B: 00 00 01 11

A^B: 00 00 10 11
B^A: 00 00 10 11

A^A: 00 00 00 00
A^0: 00 00 11 00

A^B^A: = A^A^B = B = 00 00 01 11
```

## 3.6 按位左移操作 <<

`num << i` 将 `num` 的二进制表示向左移动 `i` 位所得的值。

```
00 00 10 11 -> 11
11 << 3
---
01 01 10 00 -> 88
```

## 3.7 按位右移操作 >>

`num >> i` 将 `num` 的二进制表示向右移动 `i` 位所得的值。

```
00 00 10 11 -> 11
11 >> 2
---
00 00 00 10 -> 2
```

## 3.8 利用位运算实现快速计算

通过 `<<`，`>>` 快速计算2的倍数问题。

```
n << 1 -> 计算 n*2
n >> 1 -> 计算 n/2，负奇数的运算不可用
n << m -> 计算 n*(2^m)，即乘以 2 的 m 次方
n >> m -> 计算 n/(2^m)，即除以 2 的 m 次方
1 << n -> 2^n
```

通过 `^` 快速交换两个整数。

```
a ^= b
b ^= a
a ^= b
```

通过 `a & (-a)` 快速获取 `a` 的最后为 1 位置的整数。

```
00 00 01 01 -> 5
&
11 11 10 11 -> -5
---
00 00 00 01 -> 1

00 00 11 10 -> 14
&
11 11 00 10 -> -14
---
00 00 00 10 -> 2
```

## 3.9 利用位运算实现整数集合

一个数的二进制表示可以看作是一个集合（0 表示不在集合中，1 表示在集合中）。

比如集合 `{1, 3, 4, 8}`，可以表示成 `01 00 01 10 10` 而对应的位运算也就可以看作是对集合进行的操作。

元素与集合的操作：

```
a | (1<<i)  -> 把 i 插入到集合中
a & ~(1<<i) -> 把 i 从集合中删除
a & (1<<i)  -> 判断 i 是否属于该集合（零不属于，非零属于）
```

集合之间的操作：

```
a 补   -> ~a
a 交 b -> a & b
a 并 b -> a | b
a 差 b -> a & (~b)
```

---

整数在内存中是以补码的形式存在的，输出自然也是按照补码输出。

```
class Program
{
    static void Main(string[] args)
    {
        string s1 = Convert.ToString(-3, 2);
        Console.WriteLine(s1);
        // 11111111111111111111111111111101

        string s2 = Convert.ToString(-3, 16);
        Console.WriteLine(s2);
        // fffffffd
    }
}
```

但我们看一下 Python 的 `bin()` 输出。

```
print(bin(3))  # 0b11
print(bin(-3))  # -0b11

print(bin(-3 & 0xffffffff))
# 0b11111111111111111111111111111101

print(bin(0xfffffffd))
# 0b11111111111111111111111111111101

print(0xfffffffd)  # 4294967293
```

是不是很颠覆认知，我们从结果可以看出：

1. Python中 `bin` 一个负数（十进制表示），输出的是它的原码的二进制表示加上个负号，巨坑。

2. Python中的整型是补码形式存储的。

3. Python中整型是不限制长度的不会超范围溢出。

所以为了获得负数（十进制表示）的补码，需要手动将其和十六进制数`0xffffffff`进行按位与操作，再交给`bin()`进行输出，得到的才是负数的补码表示。

**练习题：**

leetcode 习题 136. 只出现一次的数字

给定一个**非空**整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

尝试使用位运算解决此题。

题目说明:

```
"""

Input file
example1: [2,2,1]
example2: [4,1,2,1,2]

Output file
result1: 1
result2: 4

"""



class Solution:
    def singleNumber(self, nums: List[int]) -> int:

     # your code here
```

# 4 条件语句

## 4.1 if 语句

```
if expression:
    expr_true_suite
```

1. if 语句的 `expr_true_suite` 代码块只有当条件表达式 `expression` 结果为真时才执行，否则将继续执行紧跟在该代码块后面的语句。
2. 单个 if 语句中的 `expression` 条件表达式可以通过布尔操作符 `and`， `or` 和 `not` 实现多重条件判断。

```
if 2 > 1 and not 2 > 3:
    print('Correct Judgement!')

# Correct Judgement!
```

## 4.2 if - else 语句

```
if expression:
    expr_true_suite
else
    expr_false_suite
```

1. Python 提供与 if 搭配使用的 else，如果 if 语句的条件表达式结果布尔值为假，那么程序将执行 else 语句后的代码。

【例子】

```
temp = input("猜一猜小姐姐想的是哪个数字？")
guess = int(temp) # input 函数将接收的任何数据类型都默认为 str。
if guess == 666:
    print("你太了解小姐姐的心思了！")
    print("哼，猜对也没有奖励！")
else:
    print("猜错了，小姐姐现在心里想的是666！")
print("游戏结束，不玩儿啦！")
```

`if`语句支持嵌套，即在一个`if`语句中嵌入另一个`if`语句，从而构成不同层次的选择结构。Python 使用缩进而不是大括号来标记代码块边界，因此要特别注意`else`的悬挂问题。

【例子】

```
hi = 6
if hi > 2:
    if hi > 7:
        print('好棒!好棒!')
else:
    print('切~')
```

【例子】

```
temp = input("不妨猜一下小哥哥现在心里想的是那个数字：")
guess = int(temp)
if guess > 8:
    print("大了，大了")
else:
    if guess == 8:
        print("你这么懂小哥哥的心思吗？")
        print("哼，猜对也没有奖励！")
    else:
        print("小了，小了")
print("游戏结束，不玩儿啦！")
```

## 4.3 if - elif - else 语句

```
if expression1:
    expr1_true_suite
elif expression2:
    expr2_true_suite
    .
    .
elif expressionN:
    exprN_true_suite
else:
    expr_false_suite
```

1. elif 语句即为 else if，用来检查多个表达式是否为真，并在为真时执行特定代码块中的代码。

【例子】

```
temp = input('请输入成绩:')
source = int(temp)
if 100 >= source >= 90:
    print('A')
elif 90 > source >= 80:
    print('B')
elif 80 > source >= 60:
    print('C')
elif 60 > source >= 0:
    print('D')
else:
    print('输入错误！')
```

## 4.4 assert 关键词

1. `assert` 这个关键词我们称之为“断言”，当这个关键词后边的条件为 False 时，程序自动崩溃并抛出 `AssertionError` 的异常。

【例子】

```
my_list = ['lsgogroup']
my_list.pop(0)
assert len(my_list) > 0

# AssertionError
```

1. 在进行单元测试时，可以用来在程序中置入检查点，只有条件为 True 才能让程序正常工作。

【例子】

```
assert 3 > 7

# AssertionError
```

# 5 循环语句

## 5.1 while 循环

`while`语句最基本的形式包括一个位于顶部的布尔表达式，一个或多个属于`while`代码块的缩进语句。

```
while 布尔表达式:
    代码块
```

`while`循环的代码块会一直循环执行，直到布尔表达式的值为布尔假。

如果布尔表达式不带有`<、>、==、！=、in、not in`等运算符，仅仅给出数值之类的条件，也是可以的。当`while`后写入一个非零整数时，视为真值，执行循环体；写入`0`时，视为假值，不执行循环体。也可以写入`str、list`或任何序列，长度非零则视为真值，执行循环体；否则视为假值，不执行循环体。

【例子】

```
count = 0
while count < 3:
    temp = input("不妨猜一下小哥哥现在心里想的是那个数字：")
    guess = int(temp)
    if guess > 8:
        print("大了，大了")
    else:
        if guess == 8:
            print("你是小哥哥心里的蛔虫吗？")
            print("哼，猜对也没有奖励！")
            count = 3
        else:
            print("小了，小了")
    count = count + 1
print("游戏结束，不玩儿啦！")
```

【例子】布尔表达式返回0，循环终止。

```python
string = 'abcd'
while string:
    print(string)
    string = string[1:]

# abcd
# bcd
# cd
# d
```

## 5.2 while - else 循环

```python
while 布尔表达式:
    代码块
else:
    代码块
```

当`while`循环正常执行完的情况下，执行`else`输出，如果`while`循环中执行了跳出循环的语句，比如`break`，将不执行`else`代码块的内容。

【例子】

```python
count = 0
while count < 5:
    print("%d is  less than 5" % count)
    count = count + 1
else:
    print("%d is not less than 5" % count)

# 0 is  less than 5
# 1 is  less than 5
# 2 is  less than 5
# 3 is  less than 5
# 4 is  less than 5
# 5 is not less than 5
```

【例子】

```python
count = 0
while count < 5:
    print("%d is  less than 5" % count)
    count = 6
    break
else:
    print("%d is not less than 5" % count)

# 0 is  less than 5
```

## 5.3 for 循环

`for`循环是迭代循环，在Python中相当于一个通用的序列迭代器，可以遍历任何有序序列，如`str、list、tuple`等，也可以遍历任何可迭代对象，如`dict`。

```
for 迭代变量 in 可迭代对象:
    代码块
```

每次循环，迭代变量被设置为可迭代对象的当前元素，提供给代码块使用。

【例子】

```python
for i in 'ILoveLSGO':
    print(i, end=' ')  # 不换行输出

# I L o v e L S G O
```

【例子】

```python
member = ['张三', '李四', '刘德华', '刘六', '周润发']
for each in member:
    print(each)

# 张三
# 李四
# 刘德华
# 刘六
# 周润发

for i in range(len(member)):
    print(member[i])

# 张三
```

```
# 李四
# 刘德华
# 刘六
# 周润发
```

【例子】

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

for key, value in dic.items():
    print(key, value, sep=':', end=' ')

# a:1 b:2 c:3 d:4
```

【例子】

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

for key in dic.keys():
    print(key, end=' ')

# a b c d
```

【例子】

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

for value in dic.values():
    print(value, end=' ')

# 1 2 3 4
```

## 5.4 for - else 循环

```
for 迭代变量 in 可迭代对象:
    代码块
else:
    代码块
```

当`for`循环正常执行完的情况下，执行`else`输出，如果`for`循环中执行了跳出循环的语句，比如`break`，将不执行`else`代码块的内容，与`while - else`语句一样。

【例子】

```
for num in range(10, 20):  # 迭代 10 到 20 之间的数字
    for i in range(2, num):  # 根据因子迭代
```

```
        if num % i == 0:  # 确定第一个因子
            j = num / i  # 计算第二个因子
            print('%d 等于 %d * %d' % (num, i, j))
            break  # 跳出当前循环
    else:  # 循环的 else 部分
        print(num, '是一个质数')

# 10 等于 2 * 5
# 11 是一个质数
# 12 等于 2 * 6
# 13 是一个质数
# 14 等于 2 * 7
# 15 等于 3 * 5
# 16 等于 2 * 8
# 17 是一个质数
# 18 等于 2 * 9
# 19 是一个质数
```

## 5.5 range() 函数

```
range([start,] stop[, step=1])
```

1. 这个BIF（Built-in functions）有三个参数，其中用中括号括起来的两个表示这两个参数是可选的。
2. `step=1` 表示第三个参数的默认值是1。
3. `range` 这个BIF的作用是生成一个从 `start` 参数的值开始到 `stop` 参数的值结束的数字序列，该序列包含 `start` 的值但不包含 `stop` 的值。

【例子】

```
for i in range(2, 9):  # 不包含9
    print(i)

# 2
# 3
# 4
# 5
# 6
# 7
# 8
```

【例子】

```python
for i in range(1, 10, 2):
    print(i)

# 1
# 3
# 5
# 7
# 9
```

## 5.6 enumerate()函数

```python
enumerate(sequence, [start=0])
```

1. sequence -- 一个序列、迭代器或其他支持迭代对象。
2. start -- 下标起始位置。
3. 返回 enumerate(枚举) 对象

【例子】

```python
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
lst = list(enumerate(seasons))
print(lst)
# [(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
lst = list(enumerate(seasons, start=1))  # 下标从 1 开始
print(lst)
# [(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

`enumerate()` 与 for 循环的结合使用

```python
for i, a in enumerate(A)
    do something with a
```

用 `enumerate(A)` 不仅返回了 `A` 中的元素，还顺便给该元素一个索引值 (默认从 0 开始)。此外，用 `enumerate(A, j)` 还可以确定索引起始值为 `j`。

【例子】

```python
languages = ['Python', 'R', 'Matlab', 'C++']
for language in languages:
    print('I love', language)
print('Done!')

'''
I love Python
```

```
I love R
I love Matlab
I love C++
Done!
'''

for i, language in enumerate(languages, 2):
    print(i, 'I love', language)
print('Done!')

'''
2 I love Python
3 I love R
4 I love Matlab
5 I love C++
Done!
'''
```

## 5.7 break 语句

`break`语句可以跳出当前所在层的循环。

【例子】

```
import random
secret = random.randint(1, 10) #[1,10]之间的随机数

while True:
    temp = input("不妨猜一下小哥哥现在心里想的是那个数字：")
    guess = int(temp)
    if guess > secret:
        print("大了，大了")
    else:
        if guess == secret:
            print("你是小哥哥心里的蛔虫吗？")
            print("哼，猜对也没有奖励！")
            break
        else:
            print("小了，小了")
print("游戏结束，不玩儿啦！")
```

## 5.8 continue 语句

`continue` 终止本轮循环并开始下一轮循环。

【例子】

```
for i in range(10):
    if i % 2 != 0:
        print(i)
        continue
    i += 2
    print(i)

# 2
# 1
# 4
# 3
# 6
# 5
# 8
# 7
# 10
# 9
```

## 5.9 pass 语句

`pass` 语句的意思是“不做任何事”，如果你在需要有语句的地方不写任何语句，那么解释器会提示出错，而 `pass` 语句就是用来解决这些问题的。

【例子】

```
def a_func():

# SyntaxError: unexpected EOF while parsing
```

【例子】

```
def a_func():
    pass
```

`pass` 是空语句，不做任何操作，只起到占位的作用，其作用是为了保持程序结构的完整性。尽管 `pass` 语句不做任何操作，但如果暂时不确定要在一个位置放上什么样的代码，可以先放置一个 `pass` 语句，让代码可以正常运行。

# 5.10 推导式

**列表推导式**

```
[ expr for value in collection [if condition] ]
```

【例子】

```
x = [-4, -2, 0, 2, 4]
y = [a * 2 for a in x]
print(y)
# [-8, -4, 0, 4, 8]
```

【例子】

```
x = [i ** 2 for i in range(1, 10)]
print(x)
# [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

【例子】

```
x = [(i, i ** 2) for i in range(6)]
print(x)

# [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

【例子】

```
x = [i for i in range(100) if (i % 2) != 0 and (i % 3) == 0]
print(x)

# [3, 9, 15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75, 81, 87, 93, 99]
```

【例子】

```
a = [(i, j) for i in range(0, 3) for j in range(0, 3)]
print(a)

# [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

【例子】

```
x = [[i, j] for i in range(0, 3) for j in range(0, 3)]
print(x)
# [[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]

x[0][0] = 10
print(x)
# [[10, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
```

【例子】

```
a = [(i, j) for i in range(0, 3) if i < 1 for j in range(0, 3) if j > 1]
print(a)

# [(0, 2)]
```

**元组推导式**

```
( expr for value in collection [if condition] )
```

【例子】

```
a = (x for x in range(10))
print(a)

# <generator object <genexpr> at 0x0000025BE511CC48>

print(tuple(a))

# (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

**字典推导式**

```
{ key_expr: value_expr for value in collection [if condition] }
```

【例子】

```
b = {i: i % 2 == 0 for i in range(10) if i % 3 == 0}
print(b)
# {0: True, 3: False, 6: True, 9: False}
```

**集合推导式**

```
{ expr for value in collection [if condition] }
```

【例子】

```
c = {i for i in [1, 2, 3, 4, 5, 5, 6, 4, 3, 2, 1]}
print(c)
# {1, 2, 3, 4, 5, 6}
```

**其它**

```
d = 'i for i in "I Love Lsgogroup"'
print(d)
# i for i in "I Love Lsgogroup"

e = (i for i in range(10))
print(e)
# <generator object <genexpr> at 0x0000007A0B8D01B0>

print(next(e))  # 0
print(next(e))  # 1

```

```python
for each in e:
    print(each, end=' ')

# 2 3 4 5 6 7 8 9

s = sum([i for i in range(101)])
print(s)  # 5050
s = sum((i for i in range(101)))
print(s)  # 5050
```

## 5.11 综合例子

```python
passwdList = ['123', '345', '890']
valid = False
count = 3
while count > 0:
    password = input('enter password:')
    for item in passwdList:
        if password == item:
            valid = True
            break

    if not valid:
        print('invalid input')
        count -= 1
        continue
    else:
        break
```

**参考文献：**


1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443
3. https://mp.weixin.qq.com/s/DZ589xEbOQ2QLtiq8mP1qQ


**练习题：**

1. 编写一个Python程序来查找那些可以被7除以5的整数的数字，介于1500和2700之间。

```
# your code here
```

2. 龟兔赛跑游戏

题目描述

话说这个世界上有各种各样的兔子和乌龟，但是 研究发现，所有的兔子和乌龟都有一个共同的特点——喜欢赛跑。于是世界上各个角落都不断在发生着乌龟和兔子的比赛，小华对此很感兴趣，于是决定研究不同兔 子和乌龟的赛跑。他发现，兔子虽然跑比乌龟快，但它们有众所周知的毛病——骄傲且懒惰，于是在与乌龟的比赛中，一旦任一秒结束后兔子发现自己领先t米或以 上，它们就会停下来休息s秒。对于不同的兔子，t，s的数值是不同的，但是所有的乌龟却是一致——它们不到终点决不停止。

然而有些比赛相当漫长，全程观看会耗费大量时间，而小华发现只要在每场比赛开始后记录下兔子和乌龟的数据——兔子的速度v1（表示每秒兔子能跑v1 米），乌龟的速度v2，以及兔子对应的t，s值，以及赛道的长度l——就能预测出比赛的结果。但是小华很懒，不想通过手工计算推测出比赛的结果，于是他找 到了你——清华大学计算机系的高才生——请求帮助，请你写一个程序，对于输入的一场比赛的数据v1，v2，t，s，l，预测该场比赛的结果。

输入

输入只有一行，包含用空格隔开的五个正整数v1，v2，t，s，l，其中(v1,v2< =100;t< =300;s< =10;l< =10000且为v1,v2的公倍数)

输出

输出包含两行，第一行输出比赛结果——一个大写字母“T”或“R”或“D”，分别表示乌龟获胜，兔子获胜，或者两者同时到达终点。

第二行输出一个正整数，表示获胜者（或者双方同时）到达终点所耗费的时间（秒数）。

---

样例输入

10 5 5 2 20

样例输出

D

4

```
# your code here
```

# 6 异常处理

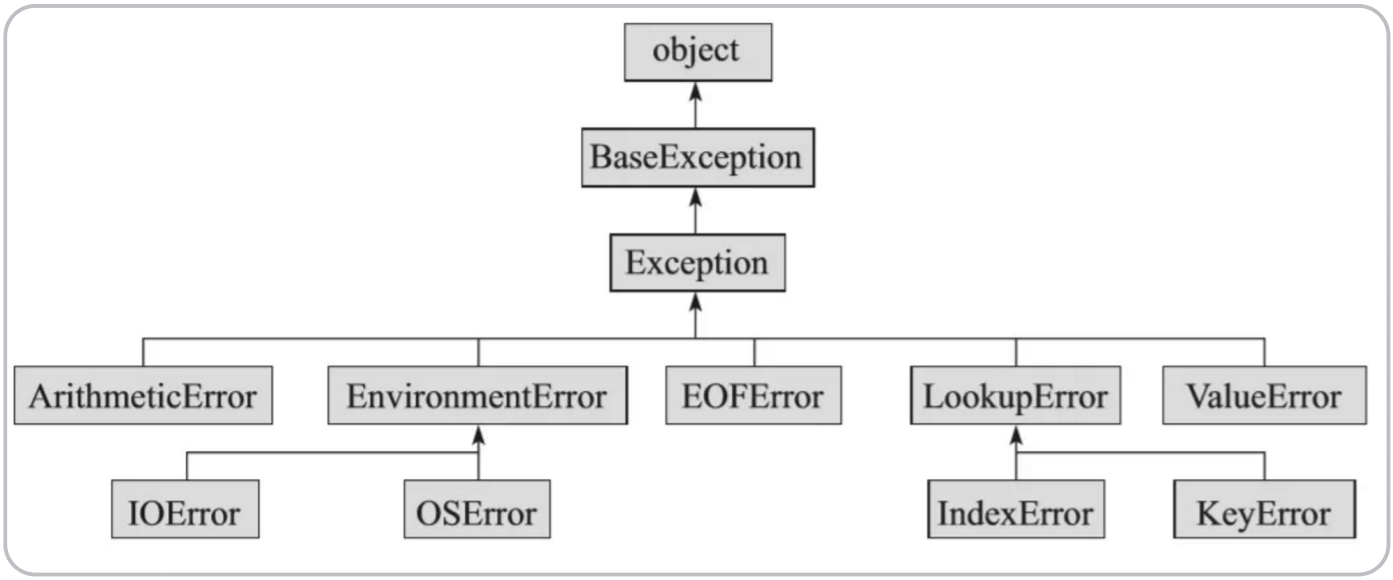
异常就是运行期检测到的错误。计算机语言针对可能出现的错误定义了异常类型，某种错误引发对应的异常时，异常处理程序将被启动，从而恢复程序的正常运行。

## 6.1 Python 标准异常总结

1. BaseException：所有异常的 **基类**
2. Exception：常规异常的 **基类**
3. StandardError：所有的内建标准异常的基类
4. ArithmeticError：所有数值计算异常的基类
5. FloatingPointError：浮点计算异常
6. OverflowError：数值运算超出最大限制
7. ZeroDivisionError：除数为零
8. AssertionError：断言语句（assert）失败
9. AttributeError：尝试访问未知的对象属性
10. EOFError：没有内建输入，到达EOF标记
11. EnvironmentError：操作系统异常的基类
12. IOError：输入/输出操作失败
13. OSError：操作系统产生的异常（例如打开一个不存在的文件）
14. WindowsError：系统调用失败
15. ImportError：导入模块失败的时候
16. KeyboardInterrupt：用户中断执行
17. LookupError：无效数据查询的基类
18. IndexError：索引超出序列的范围
19. KeyError：字典中查找一个不存在的关键字
20. MemoryError：内存溢出（可通过删除对象释放内存）
21. NameError：尝试访问一个不存在的变量
22. UnboundLocalError：访问未初始化的本地变量
23. ReferenceError：弱引用试图访问已经垃圾回收了的对象
24. RuntimeError：一般的运行时异常

25. NotImplementedError：尚未实现的方法
26. SyntaxError：语法错误导致的异常
27. IndentationError：缩进错误导致的异常
28. TabError：Tab和空格混用
29. SystemError：一般的解释器系统异常
30. TypeError：不同类型间的无效操作
31. ValueError：传入无效的参数
32. UnicodeError：Unicode相关的异常
33. UnicodeDecodeError：Unicode解码时的异常
34. UnicodeEncodeError：Unicode编码错误导致的异常
35. UnicodeTranslateError：Unicode转换错误导致的异常

异常体系内部有层次关系，Python异常体系中的部分关系如下所示：




## 6.2 Python标准警告总结

1. Warning：警告的基类
2. DeprecationWarning：关于被弃用的特征的警告
3. FutureWarning：关于构造将来语义会有改变的警告
4. UserWarning：用户代码生成的警告
5. PendingDeprecationWarning：关于特性将会被废弃的警告
6. RuntimeWarning：可疑的运行时行为(runtime behavior)的警告
7. SyntaxWarning：可疑语法的警告
8. ImportWarning：用于在导入模块过程中触发的警告
9. UnicodeWarning：与Unicode相关的警告
10. BytesWarning：与字节或字节码相关的警告

11. ResourceWarning：与资源使用相关的警告

## 6.3 try - except 语句

```
try:
    检测范围
except Exception[as reason]:
    出现异常后的处理代码
```

try 语句按照如下方式工作：

1. 首先，执行 `try` 子句（在关键字 `try` 和关键字 `except` 之间的语句）
2. 如果没有异常发生，忽略 `except` 子句， `try` 子句执行后结束。
3. 如果在执行 `try` 子句的过程中发生了异常，那么 `try` 子句余下的部分将被忽略。如果异常的类型和 `except` 之后的名称相符，那么对应的 `except` 子句将被执行。最后执行 `try` 语句之后的代码。
4. 如果一个异常没有与任何的 `except` 匹配，那么这个异常将会传递给上层的 `try` 中。

【例子】

```
try:
    f = open('test.txt')
    print(f.read())
    f.close()
except OSError:
    print('打开文件出错')

# 打开文件出错
```

【例子】

```
try:
    f = open('test.txt')
    print(f.read())
    f.close()
except OSError as error:
    print('打开文件出错\n原因是：' + str(error))

# 打开文件出错
# 原因是：[Errno 2] No such file or directory: 'test.txt'
```

一个 `try` 语句可能包含多个 `except` 子句，分别来处理不同的特定的异常。最多只有一个分支会被执行。

【例子】

```
try:
```

```
    int("abc")
    s = 1 + '1'
    f = open('test.txt')
    print(f.read())
    f.close()
except OSError as error:
    print('打开文件出错\n原因是: ' + str(error))
except TypeError as error:
    print('类型出错\n原因是: ' + str(error))
except ValueError as error:
    print('数值出错\n原因是: ' + str(error))

# 数值出错
# 原因是: invalid literal for int() with base 10: 'abc'
```

【例子】

```
dict1 = {'a': 1, 'b': 2, 'v': 22}
try:
    x = dict1['y']
except LookupError:
    print('查询错误')
except KeyError:
    print('键错误')
else:
    print(x)

# 查询错误
```

`try-except-else`语句尝试查询不在`dict`中的键值对，从而引发了异常。这一异常准确地说应属于`KeyError`，但由于`KeyError`是`LookupError`的子类，且将`LookupError`置于`KeyError`之前，因此程序优先执行该`except`代码块。所以，使用多个`except`代码块时，必须坚持对其规范排序，要从最具针对性的异常到最通用的异常。

【例子】

```
dict1 = {'a': 1, 'b': 2, 'v': 22}
try:
    x = dict1['y']
except KeyError:
    print('键错误')
except LookupError:
    print('查询错误')
else:
    print(x)

# 键错误
```

【例子】一个 `except` 子句可以同时处理多个异常，这些异常将被放在一个括号里成为一个元组。

```
try:
    s = 1 + '1'
    int("abc")
    f = open('test.txt')
    print(f.read())
    f.close()
except (OSError, TypeError, ValueError) as error:
    print('出错了！\n原因是：' + str(error))

# 出错了！
# 原因是：unsupported operand type(s) for +: 'int' and 'str'
```

## 6.4 try - except - finally 语句

```
try:
    检测范围
except Exception[as reason]:
    出现异常后的处理代码
finally:
    无论如何都会被执行的代码
```

不管 `try` 子句里面有没有发生异常， `finally` 子句都会执行。

如果一个异常在 `try` 子句里被抛出，而又没有任何的 `except` 把它截住，那么这个异常会在 `finally` 子句执行后被抛出。

【例子】

```
def divide(x, y):
    try:
        result = x / y
        print("result is", result)
    except ZeroDivisionError:
        print("division by zero!")
    finally:
        print("executing finally clause")


divide(2, 1)
# result is 2.0
# executing finally clause
```

```
divide(2, 0)
# division by zero!
# executing finally clause
divide("2", "1")
# executing finally clause
# TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

## 6.5 try - except - else 语句

如果在`try`子句执行时没有发生异常，Python将执行`else`语句后的语句。

```
try:
    检测范围
except:
    出现异常后的处理代码
else:
    如果没有异常执行这块代码
```

使用`except`而不带任何异常类型，这不是一个很好的方式，我们不能通过该程序识别出具体的异常信息，因为它捕获所有的异常。

```
try:
    检测范围
except(Exception1[, Exception2[,...ExceptionN]]):
   发生以上多个异常中的一个，执行这块代码
else:
    如果没有异常执行这块代码
```

【例子】

```
try:
    fh = open("testfile", "w")
    fh.write("这是一个测试文件，用于测试异常!!")
except IOError:
    print("Error: 没有找到文件或读取文件失败")
else:
    print("内容写入文件成功")
    fh.close()

# 内容写入文件成功
```

注意：`else`语句的存在必须以`except`语句的存在为前提，在没有`except`语句的`try`语句中使用`else`语句，会引发语法错误。

## 6.6 raise语句

Python 使用`raise`语句抛出一个指定的异常。

【例子】

```
try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')

# An exception flew by!
```

---

**练习题：**

1. 猜数字游戏

   题目描述:

   电脑产生一个零到100之间的随机数字，然后让用户来猜，如果用户猜的数字比这个数字大，提示太大，否则提示太小，当用户正好猜中电脑会提示，"恭喜你猜到了这个数是....."。在用户每次猜测之前程序会输出用户是第几次猜测，如果用户输入的根本不是一个数字，程序会告诉用户"输入无效".

   (尝试使用try catch异常处理结构对输入情况进行处理)

   获取随机数采用random模块。


```
# your code here
```

# 7 列表

简单数据类型

1. 整型`<class 'int'>`
2. 浮点型`<class 'float'>`
3. 布尔型`<class 'bool'>`

容器数据类型

1. 列表`<class 'list'>`
2. 元组`<class 'tuple'>`
3. 字典`<class 'dict'>`
4. 集合`<class 'set'>`
5. 字符串`<class 'str'>`

## 7.7 列表的定义

列表是有序集合，没有固定大小，能够保存任意数量任意类型的 Python 对象，语法为`[元素1，元素2，...，元素n]`。

1. 关键点是「中括号 []」和「逗号 ,」
2. 中括号 把所有元素绑在一起
3. 逗号 将每个元素一一分开

## 7.8 列表的创建

1. 创建一个普通列表

【例子】

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
print(x, type(x))
# ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'] <class 'list'>

x = [2, 3, 4, 5, 6, 7]
print(x, type(x))
# [2, 3, 4, 5, 6, 7] <class 'list'>

```

1. 利用`range()`创建列表

【例子】

```python
x = list(range(10))
print(x, type(x))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] <class 'list'>

x = list(range(1, 11, 2))
print(x, type(x))
# [1, 3, 5, 7, 9] <class 'list'>

x = list(range(10, 1, -2))
print(x, type(x))
# [10, 8, 6, 4, 2] <class 'list'>
```

1. 利用推导式创建列表

【例子】

```python
x = [0] * 5
print(x, type(x))
# [0, 0, 0, 0, 0] <class 'list'>

x = [0 for i in range(5)]
print(x, type(x))
# [0, 0, 0, 0, 0] <class 'list'>

x = [i for i in range(10)]
print(x, type(x))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] <class 'list'>

x = [i for i in range(1, 10, 2)]
print(x, type(x))
# [1, 3, 5, 7, 9] <class 'list'>

x = [i for i in range(10, 1, -2)]
print(x, type(x))
# [10, 8, 6, 4, 2] <class 'list'>

x = [i ** 2 for i in range(1, 10)]
print(x, type(x))
# [1, 4, 9, 16, 25, 36, 49, 64, 81] <class 'list'>

x = [i for i in range(100) if (i % 2) != 0 and (i % 3) == 0]
print(x, type(x))

# [3, 9, 15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75, 81, 87, 93, 99] <class 'list'>
```

1. 创建一个 4×3的二维数组

【例子】

```
x = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [0, 0, 0]]
print(x, type(x))
# [[1, 2, 3], [4, 5, 6], [7, 8, 9], [0, 0, 0]] <class 'list'>

for i in x:
    print(i, type(i))
# [1, 2, 3] <class 'list'>
# [4, 5, 6] <class 'list'>
# [7, 8, 9] <class 'list'>
# [0, 0, 0] <class 'list'>

x = [[0 for col in range(3)] for row in range(4)]
print(x, type(x))
# [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]] <class 'list'>

x[0][0] = 1
print(x, type(x))
# [[1, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]] <class 'list'>

x = [[0] * 3 for row in range(4)]
print(x, type(x))
# [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]] <class 'list'>

x[1][1] = 1
print(x, type(x))
# [[0, 0, 0], [0, 1, 0], [0, 0, 0], [0, 0, 0]] <class 'list'>
```

注意：

由于list的元素可以是任何对象，因此列表中所保存的是对象的指针。即使保存一个简单的`[1,2,3]`，也有3个指针和3个整数对象。

`x = [a] * 4`操作中，只是创建4个指向list的引用，所以一旦`a`改变，`x`中4个`a`也会随之改变。

【例子】

```
x = [[0] * 3] * 4
print(x, type(x))
# [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]] <class 'list'>

x[0][0] = 1
print(x, type(x))
# [[1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0]] <class 'list'>

a = [0] * 3
```

```
x = [a] * 4
print(x, type(x))
# [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]] <class 'list'>

x[0][0] = 1
print(x, type(x))
# [[1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0]] <class 'list'>
```

1. 创建一个混合列表

【例子】

```
mix = [1, 'lsgo', 3.14, [1, 2, 3]]
print(mix)  # [1, 'lsgo', 3.14, [1, 2, 3]]
```

1. 创建一个空列表

【例子】

```
empty = []
print(empty)  # []
```

列表不像元组，列表内容可更改 (mutable)，因此附加 (`append`, `extend`)、插入 (`insert`)、删除 (`remove`, `pop`) 这些操作都可以用在它身上。

## 7.9 向列表中添加元素

1. `list.append(obj)` 在列表末尾添加新的对象，只接受一个参数，参数可以是任何数据类型，被追加的元素在 list 中保持着原结构类型。

【例子】

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
x.append('Thursday')
print(x)
# ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Thursday']

print(len(x))  # 6
```

此元素如果是一个 list，那么这个 list 将作为一个整体进行追加，注意 `append()` 和 `extend()` 的区别。

【例子】

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
x.append(['Thursday', 'Sunday'])
print(x)
# ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', ['Thursday', 'Sunday']]

print(len(x))  # 6
```

1. `list.extend(seq)` 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）

【例子】

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
x.extend(['Thursday', 'Sunday'])
print(x)
# ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Thursday', 'Sunday']

print(len(x))  # 7
```

严格来说 `append` 是追加，把一个东西整体添加在列表后，而 `extend` 是扩展，把一个东西里的所有元素添加在列表后。

1. `list.insert(index, obj)` 在编号 `index` 位置前插入 `obj`。

【例子】

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
x.insert(2, 'Sunday')
print(x)
# ['Monday', 'Tuesday', 'Sunday', 'Wednesday', 'Thursday', 'Friday']

print(len(x))  # 6
```

## 7.10 删除列表中的元素

1. `list.remove(obj)` 移除列表中某个值的第一个匹配项

【例子】

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
x.remove('Monday')
print(x)  # ['Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

1. `list.pop([index=-1])` 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值

【例子】

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
y = x.pop()
print(y)  # Friday

y = x.pop(0)
print(y)  # Monday

y = x.pop(-2)
print(y)  # Wednesday
print(x)  # ['Tuesday', 'Thursday']
```

`remove` 和 `pop` 都可以删除元素，前者是指定具体要删除的元素，后者是指定一个索引。

1. `del var1[, var2 ……]` 删除单个或多个对象。

【例子】

如果知道要删除的元素在列表中的位置，可使用`del`语句。

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
del x[0:2]
print(x)  # ['Wednesday', 'Thursday', 'Friday']
```

如果你要从列表中删除一个元素，且不再以任何方式使用它，就使用`del`语句；如果你要在删除元素后还能继续使用它，就使用方法`pop()`。

## 7.11 获取列表中的元素

1. 通过元素的索引值，从列表获取单个元素，注意，列表索引值是从0开始的。
2. 通过将索引指定为-1，可让Python返回最后一个列表元素，索引 -2 返回倒数第二个列表元素，以此类推。

【例子】

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
print(x[0], type(x[0]))  # Monday <class 'str'>
print(x[-1], type(x[-1]))  # Friday <class 'str'>
print(x[-2], type(x[-2]))  # Thursday <class 'str'>
```

切片的通用写法是 `start : stop : step`

1. 情况 1 - "start :"
2. 以 `step` 为 1 (默认) 从编号 `start` 往列表尾部切片。

【例子】

```
x = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
print(x[3:])  # ['Thursday', 'Friday']
print(x[-3:])  # ['Wednesday', 'Thursday', 'Friday']
```

1. 情况 2 - ": stop"
2. 以 `step` 为 1 (默认) 从列表头部往编号 `stop` 切片。

【例子】

```
week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
print(week[:3])  # ['Monday', 'Tuesday', 'Wednesday']
print(week[:-3])  # ['Monday', 'Tuesday']
```

1. 情况 3 - "start : stop"
2. 以 `step` 为 1 (默认) 从编号 `start` 往编号 `stop` 切片。

【例子】

```
week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
print(week[1:3])  # ['Tuesday', 'Wednesday']
print(week[-3:-1])  # ['Wednesday', 'Thursday']
```

1. 情况 4 - "start : stop : step"
2. 以具体的 `step` 从编号 `start` 往编号 `stop` 切片。注意最后把 `step` 设为 -1，相当于将列表反向排列。

【例子】

```
week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
print(week[1:4:2])
# ['Tuesday', 'Thursday']

print(week[:4:2])
# ['Monday', 'Wednesday']

print(week[1::2])
# ['Tuesday', 'Thursday']

print(week[::-1])
# ['Friday', 'Thursday', 'Wednesday', 'Tuesday', 'Monday']
```

1. 情况 5 - " : "
2. 复制列表中的所有元素（浅拷贝）。

【例子】

```
week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

```
print(week[:])
# week的拷贝 ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

list1 = [123, 456, 789, 213]
list2 = list1
list3 = list1[:]
print(list2)  # [123, 456, 789, 213]
print(list3)  # [123, 456, 789, 213]
list1.sort()
print(list2)  # [123, 213, 456, 789]
print(list3)  # [123, 456, 789, 213]

list1 = [[123, 456], [789, 213]]
list2 = list1
list3 = list1[:]
print(list2)  # [[123, 456], [789, 213]]
print(list3)  # [[123, 456], [789, 213]]
list1[0][0] = 111
print(list2)  # [[111, 456], [789, 213]]
print(list3)  # [[111, 456], [789, 213]]
```

## 7.12 列表的常用操作符

1. 等号操作符：`==`
2. 连接操作符 `+`
3. 重复操作符 `*`
4. 成员关系操作符 `in`、`not in`

「等号 ==」，只有成员、成员位置都相同时才返回True。

和元组拼接一样， 列表拼接也有两种方式，用「加号 +」和「乘号 *」，前者首尾拼接，后者复制拼接。

【例子】

```
list1 = [123, 456]
list2 = [456, 123]
list3 = [123, 456]

print(list1 == list2)  # False
print(list1 == list3)  # True

```

```
list4 = list1 + list2  # extend()
print(list4)  # [123, 456, 456, 123]

list5 = list3 * 3
print(list5)  # [123, 456, 123, 456, 123, 456]

list3 *= 3
print(list3)  # [123, 456, 123, 456, 123, 456]

print(123 in list3)  # True
print(456 not in list3)  # False
```

前面三种方法（`append`, `extend`, `insert`）可对列表增加元素，它们没有返回值，是直接修改了原数据对象。

将两个list相加，需要创建新的 list 对象，从而需要消耗额外的内存，特别是当 list 较大时，尽量不要使用 “+” 来添加list。

## 7.13 列表的其它方法

`list.count(obj)` 统计某个元素在列表中出现的次数

【例子】

```
list1 = [123, 456] * 3
print(list1)  # [123, 456, 123, 456, 123, 456]
num = list1.count(123)
print(num)  # 3
```

`list.index(x[, start[, end]])` 从列表中找出某个值第一个匹配项的索引位置

【例子】

```
list1 = [123, 456] * 5
print(list1.index(123))  # 0
print(list1.index(123, 1))  # 2
print(list1.index(123, 3, 7))  # 4
```

`list.reverse()` 反向列表中元素

【例子】

```
x = [123, 456, 789]
x.reverse()
print(x)  # [789, 456, 123]
```

`list.sort(key=None, reverse=False)` 对原列表进行排序。

1. `key` -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。
2. `reverse` -- 排序规则， `reverse = True` 降序， `reverse = False` 升序（默认）。
3. 该方法没有返回值，但是会对列表的对象进行排序。

【例子】

```
x = [123, 456, 789, 213]
x.sort()
print(x)
# [123, 213, 456, 789]

x.sort(reverse=True)
print(x)
# [789, 456, 213, 123]


# 获取列表的第二个元素
def takeSecond(elem):
    return elem[1]


x = [(2, 2), (3, 4), (4, 1), (1, 3)]
x.sort(key=takeSecond)
print(x)
# [(4, 1), (2, 2), (1, 3), (3, 4)]

x.sort(key=lambda a: a[0])
print(x)
# [(1, 3), (2, 2), (3, 4), (4, 1)]
```

---

**参考文献：**


1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443
3. https://mp.weixin.qq.com/s/DZ589xEbOQ2QLtiq8mP1qQ


---

**练习题：**

1. 列表操作练习

   列表lst 内容如下

   lst = [2, 5, 6, 7, 8, 9, 2, 9, 9]

   请写程序完成下列操作：

1. 在列表的末尾增加元素15
2. 在列表的中间位置插入元素20
3. 将列表[2, 5, 6]合并到lst中
4. 移除列表中索引为3的元素
5. 翻转列表里的所有元素
6. 对列表里的元素进行排序，从小到大一次，从大到小一次

2. 修改列表

   问题描述：

   lst = [1, [4, 6], True]

   请将列表里所有数字修改成原来的两倍

3. leetcode 852题 山脉数组的峰顶索引

   如果一个数组k符合下面两个属性，则称之为山脉数组

   数组的长度大于等于3

   存在$i$，$i$>0 且$i < \text{len}(k) - 1$， 使得$\text{k[0]} < \text{k[1]} < \ldots < \text{k[i-1]} < \text{k[j]} > \text{k[i+1]} \ldots > \text{k[len(k)-1]}$

   这个$i$就是顶峰索引。

   现在，给定一个山脉数组，求顶峰索引。

   示例:

   输入: [1, 3, 4, 5, 3]

   输出: True

   输入：[1, 2, 4, 6, 4, 5]

   输出：False

```python
class Solution:
    def peakIndexInMountainArray(self, A: List[int]) -> int:

    # your code here
```

# 8 元组

「元组」定义语法为：`(元素1, 元素2, ..., 元素n)`

1. 小括号把所有元素绑在一起
2. 逗号将每个元素一一分开

## 8.1 创建和访问一个元组

1. Python 的元组与列表类似，不同之处在于tuple被创建后就不能对其进行修改，类似字符串。
2. 元组使用小括号，列表使用方括号。

```
t1 = (1, 10.31, 'python')
t2 = 1, 10.31, 'python'
print(t1, type(t1))
# (1, 10.31, 'python') <class 'tuple'>

print(t2, type(t2))
# (1, 10.31, 'python') <class 'tuple'>

tuple1 = (1, 2, 3, 4, 5, 6, 7, 8)
print(tuple1[1])  # 2
print(tuple1[5:])  # (6, 7, 8)
print(tuple1[:5])  # (1, 2, 3, 4, 5)
tuple2 = tuple1[:]
print(tuple2)  # (1, 2, 3, 4, 5, 6, 7, 8)
```

1. 创建元组可以用小括号 ()，也可以什么都不用，为了可读性，建议还是用 ()。
2. 元组中只包含一个元素时，需要在元素后面添加逗号，否则括号会被当作运算符使用：

【例子】

```
temp = (1)
print(type(temp))  # <class 'int'>
temp = 2, 3, 4, 5
print(type(temp))  # <class 'tuple'>
temp = []
print(type(temp))  # <class 'list'>
temp = ()
print(type(temp))  # <class 'tuple'>
temp = (1,)
print(type(temp))  # <class 'tuple'>
```

【例子】

```
print(8 * (8))  # 64
print(8 * (8,))  # (8, 8, 8, 8, 8, 8, 8, 8)
```

【例子】当然也可以创建二维元组：

```
nested = (1, 10.31, 'python'), ('data', 11)
print(nested)
# ((1, 10.31, 'python'), ('data', 11))
```

【例子】元组中可以用整数来对它进行索引 (indexing) 和切片 (slicing)，不严谨的讲，前者是获取单个元素，后者是获取一组元素。接着上面二维元组的例子，先看看索引的代码。

```
print(nested[0])
# (1, 10.31, 'python')
print(nested[0][0], nested[0][1], nested[0][2])
# 1 10.31 python
```

【例子】再看看切片的代码。

```
print(nested[0][0:2])
# (1, 10.31)
```

## 8.2 更新和删除一个元组

【例子】

```
week = ('Monday', 'Tuesday', 'Thursday', 'Friday')
week = week[:2] + ('Wednesday',) + week[2:]
print(week)  # ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
```

【例子】

```python
t1 = (1, 2, 3, [4, 5, 6])
print(t1)  # (1, 2, 3, [4, 5, 6])

t1[3][0] = 9
print(t1)  # (1, 2, 3, [9, 5, 6])
```

元组有不可更改 (immutable) 的性质，因此不能直接给元组的元素赋值，但是只要元组中的元素可更改 (mutable)，那么我们可以直接更改其元素，注意这跟赋值其元素不同。

## 8.3 元组相关的操作符

1. 比较操作符
2. 逻辑操作符
3. 连接操作符 `+`
4. 重复操作符 `*`
5. 成员关系操作符 `in`、`not in`

【例子】元组拼接 (concatenate) 有两种方式，用「加号 +」和「乘号 *」，前者首尾拼接，后者复制拼接。

```python
t1 = (2, 3, 4, 5)
t2 = ('老马的程序人生', '小马的程序人生')
t3 = t1 + t2
print(t3)
# (2, 3, 4, 5, '老马的程序人生', '小马的程序人生')

t4 = t2 * 2
print(t4)
# ('老马的程序人生', '小马的程序人生', '老马的程序人生', '小马的程序人生')
```

## 8.4 内置方法

元组大小和内容都不可更改，因此只有 `count` 和 `index` 两种方法。

【例子】

```python
t = (1, 10.31, 'python')
print(t.count('python'))  # 1
print(t.index(10.31))  # 1
```

1. `count('python')` 是记录在元组 `t` 中该元素出现几次，显然是 1 次
2. `index(10.31)` 是找到该元素在元组 `t` 的索引，显然是 1

## 8.5 解压元组

【例子】解压（unpack）一维元组（有几个元素左边括号定义几个变量）

```
t = (1, 10.31, 'python')
(a, b, c) = t
print(a, b, c)
# 1 10.31 python
```

【例子】解压二维元组（按照元组里的元组结构来定义变量）

```
t = (1, 10.31, ('OK', 'python'))
(a, b, (c, d)) = t
print(a, b, c, d)
# 1 10.31 OK python
```

【例子】如果你只想要元组其中几个元素，用通配符「*」，英文叫 wildcard，在计算机语言中代表一个或多个元素。下例就是把多个元素丢给了 `rest` 变量。

```
t = 1, 2, 3, 4, 5
a, b, *rest, c = t
print(a, b, c)  # 1 2 5
print(rest)  # [3, 4]
```

【例子】如果你根本不在乎 rest 变量，那么就用通配符「*」加上下划线「_」。

```
a, b, *_ = t
print(a, b)  # 1 2
```

---

**练习题：**

1. 元组概念

   写出下面代码的执行结果和最终结果的类型

   ```
   (1, 2)*2
   (1, )*2
   (1)*2
   ```

   分析为什么会出现这样的结果.

2. 拆包过程是什么？

   ```
   a, b = 1, 2
   ```

   上述过程属于拆包吗？

可迭代对象拆包时，怎么赋值给占位符？

# 9 字符串

## 9.1 字符串的定义

1. Python 中字符串被定义为引号之间的字符集合。
2. Python 支持使用成对的 单引号 或 双引号。

【例子】

```
t1 = 'i love Python!'
print(t1, type(t1))
# i love Python! <class 'str'>

t2 = "I love Python!"
print(t2, type(t2))
# I love Python! <class 'str'>

print(5 + 8)  # 13
print('5' + '8')  # 58
```

1. 如果字符串中需要出现单引号或双引号，可以使用转义符号 `\` 对字符串中的符号进行转义。

【例子】

```
print('let\'s go')  # let's go
print("let's go")  # let's go
print('C:\\now')  # C:\now
print("C:\\Program Files\\Intel\\Wifi\\Help")  # C:\Program Files\Intel\Wifi\Help
```

1. Python 的常用转义字符

| 转义字符 | 描述 |
| --- | --- |
| `\\` | 反斜杠符号 |
| `\'` | 单引号 |
| `\"` | 双引号 |
| `\n` | 换行 |

| | |
|---|---|
| \t | 横向制表符(TAB) |
| \r | 回车 |

1. 原始字符串只需要在字符串前边加一个英文字母 r 即可。

【例子】

```
print(r'C:\Program Files\Intel\Wifi\Help')
# C:\Program Files\Intel\Wifi\Help
```

1. python三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。

【例子】

```
para_str = """这是一个多行字符串的实例
多行字符串可以使用制表符
TAB ( \t )。
也可以使用换行符 [ \n ]。
"""
print (para_str)

'''
这是一个多行字符串的实例
多行字符串可以使用制表符
TAB (    )。
也可以使用换行符 [
 ]。
'''
```

## 9.2 字符串的切片与拼接

1. 类似于元组具有不可修改性
2. 从 0 开始 (和 C 一样)
3. 切片通常写成 `start:end` 这种形式，包括「`start` 索引」对应的元素，不包括「`end` 索引」对应的元素。
4. 索引值可正可负，正索引从 0 开始，从左往右；负索引从 -1 开始，从右往左。使用负数索引时，会从最后一个元素开始计数。最后一个元素的位置编号是 -1。

【例子】

```python
str1 = 'I Love LsgoGroup'
print(str1[:6])  # I Love
print(str1[5])  # e
print(str1[:6] + " 插入的字符串 " + str1[6:])
# I Love 插入的字符串  LsgoGroup

s = 'Python'
print(s)  # Python
print(s[2:4])  # th
print(s[-5:-2])  # yth
print(s[2])  # t
print(s[-1])  # n
```

## 9.3 字符串的常用内置方法

1. `capitalize()` 将字符串的第一个字符转换为大写。

【例子】

```python
str2 = 'xiaoxie'
print(str2.capitalize())  # Xiaoxie
```

1. `lower()` 转换字符串中所有大写字符为小写。
2. `upper()` 转换字符串中的小写字母为大写。
3. `swapcase()` 将字符串中大写转换为小写，小写转换为大写。

【例子】

```python
str2 = "DAXIExiaoxie"
print(str2.lower())  # daxiexiaoxie
print(str2.upper())  # DAXIEXIAOXIE
print(str2.swapcase())  # daxieXIAOXIE
```

1. `count(str, beg= 0,end=len(string))` 返回 `str` 在 string 里面出现的次数，如果 `beg` 或者 `end` 指定则返回指定范围内 `str` 出现的次数。

【例子】

```python
str2 = "DAXIExiaoxie"
print(str2.count('xi'))  # 2
```

1. `endswith(suffix, beg=0, end=len(string))` 检查字符串是否以指定子字符串 `suffix` 结束，如果是，返回 True,否则返回 False。如果 `beg` 和 `end` 指定值，则在指定范围内检查。
2. `startswith(substr, beg=0,end=len(string))` 检查字符串是否以指定子字符串 `substr` 开头，如果是，返回 True，否则返回 False。如果 `beg` 和 `end` 指定值，则在指定范围内检查。

【例子】

```
str2 = "DAXIExiaoxie"
print(str2.endswith('ie'))  # True
print(str2.endswith('xi'))  # False
print(str2.startswith('Da'))  # False
print(str2.startswith('DA'))  # True
```

1. `find(str, beg=0, end=len(string))` 检测 `str` 是否包含在字符串中，如果指定范围 `beg` 和 `end`，则检查是否包含在指定范围内，如果包含，返回开始的索引值，否则返回 -1。
2. `rfind(str, beg=0,end=len(string))` 类似于 `find()` 函数，不过是从右边开始查找。

【例子】

```
str2 = "DAXIExiaoxie"
print(str2.find('xi'))  # 5
print(str2.find('ix'))  # -1
print(str2.rfind('xi'))  # 9
```

1. `isnumeric()` 如果字符串中只包含数字字符，则返回 True，否则返回 False。

【例子】

```
str3 = '12345'
print(str3.isnumeric())  # True
str3 += 'a'
print(str3.isnumeric())  # False
```

1. `ljust(width[, fillchar])` 返回一个原字符串左对齐，并使用 `fillchar`（默认空格）填充至长度 `width` 的新字符串。
2. `rjust(width[, fillchar])` 返回一个原字符串右对齐，并使用 `fillchar`（默认空格）填充至长度 `width` 的新字符串。

【例子】

```
str4 = '1101'
print(str4.ljust(8, '0'))  # 11010000
print(str4.rjust(8, '0'))  # 00001101
```

1. `lstrip([chars])` 截掉字符串左边的空格或指定字符。
2. `rstrip([chars])` 删除字符串末尾的空格或指定字符。
3. `strip([chars])` 在字符串上执行 `lstrip()` 和 `rstrip()`。

【例子】

```
str5 = ' I Love LsgoGroup '
print(str5.lstrip())  # 'I Love LsgoGroup '
print(str5.lstrip().strip('I'))  # ' Love LsgoGroup '
print(str5.rstrip())  # ' I Love LsgoGroup'
print(str5.strip())  # 'I Love LsgoGroup'
print(str5.strip().strip('p'))  # 'I Love LsgoGrou'
```

1. `partition(sub)` 找到子字符串sub，把字符串分为一个三元组`(pre_sub,sub,fol_sub)`，如果字符串中不包含sub则返回`('原字符串','','')`。
2. `rpartition(sub)`类似于`partition()`方法，不过是从右边开始查找。

【例子】

```
str5 = ' I Love LsgoGroup '
print(str5.strip().partition('o'))  # ('I L', 'o', 've LsgoGroup')
print(str5.strip().partition('m'))  # ('I Love LsgoGroup', '', '')
print(str5.strip().rpartition('o'))  # ('I Love LsgoGr', 'o', 'up')
```

1. `replace(old, new [, max])` 把 将字符串中的`old`替换成`new`，如果`max`指定，则替换不超过`max`次。

【例子】

```
str5 = ' I Love LsgoGroup '
print(str5.strip().replace('I', 'We'))  # We Love LsgoGroup
```

1. `split(str="", num)` 不带参数默认是以空格为分隔符切片字符串，如果`num`参数有设置，则仅分隔`num`个子字符串，返回切片后的子字符串拼接的列表。

【例子】

```
str5 = ' I Love LsgoGroup '
print(str5.strip().split())  # ['I', 'Love', 'LsgoGroup']
print(str5.strip().split('o'))  # ['I L', 've Lsg', 'Gr', 'up']
```

【例子】

```
u = "www.baidu.com.cn"
# 使用默认分隔符
print(u.split())  # ['www.baidu.com.cn']

# 以"."为分隔符

print((u.split('.')))  # ['www', 'baidu', 'com', 'cn']

# 分割0次
print((u.split(".", 0)))  # ['www.baidu.com.cn']

```

```
# 分割一次
print((u.split(".", 1)))  # ['www', 'baidu.com.cn']

# 分割两次
print(u.split(".", 2))  # ['www', 'baidu', 'com.cn']

# 分割两次，并取序列为1的项
print((u.split(".", 2)[1]))  # baidu

# 分割两次，并把分割后的三个部分保存到三个变量
u1, u2, u3 = u.split(".", 2)
print(u1)  # www
print(u2)  # baidu
print(u3)  # com.cn
```

【例子】去掉换行符

```
c = '''say
hello
baby'''

print(c)
# say
# hello
# baby

print(c.split('\n'))  # ['say', 'hello', 'baby']
```

【例子】

```
string = "hello boy<[www.baidu.com]>byebye"
print(string.split('[')[1].split(']')[0])  # www.baidu.com
print(string.split('[')[1].split(']')[0].split('.'))  # ['www', 'baidu', 'com']
```

1. `splitlines([keepends])` 按照行('\r', '\r\n', \n')分隔，返回一个包含各行作为元素的列表，如果参数`keepends`为 False，不包含换行符，如果为 True，则保留换行符。

【例子】

```
str6 = 'I \n Love \n LsgoGroup'
print(str6.splitlines())  # ['I ', ' Love ', ' LsgoGroup']
print(str6.splitlines(True))  # ['I \n', ' Love \n', ' LsgoGroup']
```

1. `maketrans(intab, outtab)` 创建字符映射的转换表，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。
2. `translate(table, deletechars="")` 根据参数`table`给出的表，转换字符串的字符，要过滤掉的字符放到`deletechars`参数中。

【例子】

```
str = 'this is string example....wow!!!'
intab = 'aeiou'
outtab = '12345'
trantab = str.maketrans(intab, outtab)
print(trantab)  # {97: 49, 111: 52, 117: 53, 101: 50, 105: 51}
print(str.translate(trantab))  # th3s 3s str3ng 2x1mpl2....w4w!!!
```

## 9.4 字符串格式化

1. Python `format` 格式化函数

【例子】

```
str = "{0} Love {1}".format('I', 'Lsgogroup')  # 位置参数
print(str)  # I Love Lsgogroup

str = "{a} Love {b}".format(a='I', b='Lsgogroup')  # 关键字参数
print(str)  # I Love Lsgogroup

str = "{0} Love {b}".format('I', b='Lsgogroup')  # 位置参数要在关键字参数之前
print(str)  # I Love Lsgogroup

str = '{0:.2f}{1}'.format(27.658, 'GB')  # 保留小数点后两位
print(str)  # 27.66GB
```

1. Python 字符串格式化符号

| 符 号 | 描述 |
| --- | --- |
| %c | 格式化字符及其ASCII码 |
| %s | 格式化字符串，用str()方法处理对象 |
| %r | 格式化字符串，用rper()方法处理对象 |
| %d | 格式化整数 |
| %o | 格式化无符号八进制数 |
| %x | 格式化无符号十六进制数 |
| %X | 格式化无符号十六进制数（大写） |
| %f | 格式化浮点数字，可指定小数点后的精度 |

| | |
|---|---|
| %e | 用科学计数法格式化浮点数 |
| %E | 作用同%e，用科学计数法格式化浮点数 |
| %g | 根据值的大小决定使用%f或%e |
| %G | 作用同%g，根据值的大小决定使用%f或%E |

【例子】

```
print('%c' % 97)  # a
print('%c %c %c' % (97, 98, 99))  # a b c
print('%d + %d = %d' % (4, 5, 9))  # 4 + 5 = 9
print("我叫 %s 今年 %d 岁!" % ('小明', 10))  # 我叫 小明 今年 10 岁!
print('%o' % 10)  # 12
print('%x' % 10)  # a
print('%X' % 10)  # A
print('%f' % 27.658)  # 27.658000
print('%e' % 27.658)  # 2.765800e+01
print('%E' % 27.658)  # 2.765800E+01
print('%g' % 27.658)  # 27.658
text = "I am %d years old." % 22
print("I said: %s." % text)  # I said: I am 22 years old..
print("I said: %r." % text)  # I said: 'I am 22 years old.'
```

1. 格式化操作符辅助指令

| 符号 | 功能 |
|---|---|
| m.n | m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话) |
| - | 用做左对齐 |
| + | 在正数前面显示加号( + ) |
| # | 在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X') |
| 0 | 显示的数字前面填充'0'而不是默认的空格 |

【例子】

```
print('%5.1f' % 27.658)  # ' 27.7'
print('%.2e' % 27.658)  # 2.77e+01
print('%10d' % 10)  # '        10'
print('%-10d' % 10)  # '10        '
print('%+d' % 10)  # +10
print('%#o' % 10)  # 0o12
print('%#x' % 108)  # 0x6c
print('%010d' % 5)  # 0000000005
```

**参考文献：**

1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443
3. https://mp.weixin.qq.com/s/DZ589xEbOQ2QLtiq8mP1qQ

---

**练习题：**

1. 字符串函数回顾
   a. 怎么批量替换字符串中的元素？
   b. 怎么把字符串按照空格进行拆分？
   c. 怎么去除字符串首位的空格？
2. 实现isdigit函数

   题目要求

   实现函数isdigit， 判断字符串里是否只包含数字0~9

```python
def isdigit(string):
    """
    判断字符串只包含数字
    :param string:
    :return:
    """
    # your code here
    pass
```

3. leetcode 5题 最长回文子串

   给定一个字符串 `s`，找到 `s` 中最长的回文子串。你可以假设 `s` 的最大长度为 1000。

   示例:

   输入: "babad"

   输出: "bab"

   输入: "cbbd"

   输出: "bb"

```python
class Solution:
    def longestPalindrome(self, s: str) -> str:

        # your code here
```

# 10 字典

## 10.1 可变类型与不可变类型

1. 序列是以连续的整数为索引，与此不同的是，字典以"关键字"为索引，关键字可以是任意不可变类型，通常用字符串或数值。
2. 字典是 Python 唯一的一个 映射类型，字符串、元组、列表属于序列类型。

那么如何快速判断一个数据类型 `X` 是不是可变类型的呢？两种方法：

1. 麻烦方法：用 `id(X)` 函数，对 X 进行某种操作，比较操作前后的 `id`，如果不一样，则 `X` 不可变，如果一样，则 `X` 可变。
2. 便捷方法：用 `hash(X)`，只要不报错，证明 `X` 可被哈希，即不可变，反过来不可被哈希，即可变。

```
i = 1
print(id(i))  # 140732167000896
i = i + 2
print(id(i))  # 140732167000960

l = [1, 2]
print(id(l))  # 4300825160
l.append('Python')
print(id(l))  # 4300825160
```

1. 整数 `i` 在加 1 之后的 `id` 和之前不一样，因此加完之后的这个 `i` (虽然名字没变)，但不是加之前的那个 `i` 了，因此整数是不可变类型。
2. 列表 `l` 在附加 `'Python'` 之后的 `id` 和之前一样，因此列表是可变类型。

```
print(hash('Name'))  # -9215951442099718823

print(hash((1, 2, 'Python')))  # 823623082077994 71

print(hash([1, 2, 'Python']))
# TypeError: unhashable type: 'list'

print(hash({1, 2, 3}))
# TypeError: unhashable type: 'set'
```

1. 数值、字符和元组 都能被哈希，因此它们是不可变类型。

2. 列表、集合、字典不能被哈希，因此它是可变类型。

## 10.2 字典的定义

字典 是无序的 键:值（`key:value`）对集合，键必须是互不相同的（在同一个字典之内）。

1. `dict` 内部存放的顺序和 `key` 放入的顺序是没有关系的。
2. `dict` 查找和插入的速度极快，不会随着 `key` 的增加而增加，但是需要占用大量的内存。

字典 定义语法为 `{元素1, 元素2, ..., 元素n}`

1. 其中每一个元素是一个「键值对」-- 键:值 (`key:value`)
2. 关键点是「大括号 {}」,「逗号 ,」和「冒号 :」
3. 大括号 -- 把所有元素绑在一起
4. 逗号 -- 将每个键值对分开
5. 冒号 -- 将键和值分开

## 10.3 创建和访问字典

【例子】

```
brand = ['李宁', '耐克', '阿迪达斯']
slogan = ['一切皆有可能', 'Just do it', 'Impossible is nothing']
print('耐克的口号是:', slogan[brand.index('耐克')])
# 耐克的口号是: Just do it

dic = {'李宁': '一切皆有可能', '耐克': 'Just do it', '阿迪达斯': 'Impossible is nothing'}
print('耐克的口号是:', dic['耐克'])
# 耐克的口号是: Just do it
```

通过字符串或数值作为`key`来创建字典。

注意：如果我们取的键在字典中不存在，会直接报错`KeyError`。

【例子】

```python
dic1 = {1: 'one', 2: 'two', 3: 'three'}
print(dic1)  # {1: 'one', 2: 'two', 3: 'three'}
print(dic1[1])  # one
print(dic1[4])  # KeyError: 4

dic2 = {'rice': 35, 'wheat': 101, 'corn': 67}
print(dic2)  # {'wheat': 101, 'corn': 67, 'rice': 35}
print(dic2['rice'])  # 35
```

【例子】通过元组作为`key`来创建字典，但一般不这样使用。

```python
dic = {(1, 2, 3): "Tom", "Age": 12, 3: [3, 5, 7]}
print(dic)  # {(1, 2, 3): 'Tom', 'Age': 12, 3: [3, 5, 7]}
print(type(dic))  # <class 'dict'>
```

通过构造函数`dict`来创建字典。

1. `dict()` -> 创建一个空的字典。

【例子】通过`key`直接把数据放入字典中，但一个`key`只能对应一个`value`，多次对一个`key`放入`value`，后面的值会把前面的值冲掉。

```python
dic = dict()
dic['a'] = 1
dic['b'] = 2
dic['c'] = 3

print(dic)
# {'a': 1, 'b': 2, 'c': 3}

dic['a'] = 11
print(dic)
# {'a': 11, 'b': 2, 'c': 3}

dic['d'] = 4
print(dic)
# {'a': 11, 'b': 2, 'c': 3, 'd': 4}
```

1. `dict(mapping)` -> new dictionary initialized from a mapping object's (key, value) pairs

【例子】

```python
dic1 = dict([('apple', 4139), ('peach', 4127), ('cherry', 4098)])
print(dic1)  # {'cherry': 4098, 'apple': 4139, 'peach': 4127}

dic2 = dict((('apple', 4139), ('peach', 4127), ('cherry', 4098)))
print(dic2)  # {'peach': 4127, 'cherry': 4098, 'apple': 4139}
```

1. `dict(**kwargs)` -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

【例子】这种情况下，键只能为字符串类型，并且创建的时候字符串不能加引号，加上就会直接报语法错误。

```
dic = dict(name='Tom', age=10)
print(dic)  # {'name': 'Tom', 'age': 10}
print(type(dic))  # <class 'dict'>
```

## 10.4 字典的内置方法

1. `dict.fromkeys(seq[, value])` 用于创建一个新字典，以序列 `seq` 中元素做字典的键，`value` 为字典所有键对应的初始值。

【例子】

```
seq = ('name', 'age', 'sex')
dic1 = dict.fromkeys(seq)
print("新的字典为 : %s" % str(dic1))
# 新的字典为 : {'name': None, 'age': None, 'sex': None}

dic2 = dict.fromkeys(seq, 10)
print("新的字典为 : %s" % str(dic2))
# 新的字典为 : {'name': 10, 'age': 10, 'sex': 10}

dic3 = dict.fromkeys(seq, ('小马', '8', '男'))
print("新的字典为 : %s" % str(dic3))
# 新的字典为 : {'name': ('小马', '8', '男'), 'age': ('小马', '8', '男'), 'sex': ('小马', '8', '男')}
```

1. `dict.keys()` 返回一个可迭代对象，可以使用 `list()` 来转换为列表，列表为字典中的所有键。

【例子】

```
dic = {'Name': 'lsgogroup', 'Age': 7}
print(dic.keys())  # dict_keys(['Name', 'Age'])
lst = list(dic.keys())  # 转换为列表
print(lst)  # ['Name', 'Age']
```

1. `dict.values()` 返回一个迭代器，可以使用 `list()` 来转换为列表，列表为字典中的所有值。

【例子】

```python
dic = {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}
print("字典所有值为 : ", list(dic.values()))
# 字典所有值为 :  [7, 'female', 'Zara']
```

1. `dict.items()` 以列表返回可遍历的 (键, 值) 元组数组。

【例子】

```python
dic = {'Name': 'Lsgogroup', 'Age': 7}
print("Value : %s" % dic.items())
# Value : dict_items([('Name', 'Lsgogroup'), ('Age', 7)])

print(tuple(dic.items()))
# (('Name', 'Lsgogroup'), ('Age', 7))
```

1. `dict.get(key, default=None)` 返回指定键的值，如果值不在字典中返回默认值。

【例子】

```python
dic = {'Name': 'Lsgogroup', 'Age': 27}
print("Age 值为 : %s" % dic.get('Age'))  # Age 值为 : 27
print("Sex 值为 : %s" % dic.get('Sex', "NA"))  # Sex 值为 : NA
```

1. `dict.setdefault(key, default=None)` 和 `get()` 方法 类似, 如果键不存在于字典中，将会添加键并将值设为默认值。

【例子】

```python
dic = {'Name': 'Lsgogroup', 'Age': 7}
print("Age 键的值为 : %s" % dic.setdefault('Age', None))  # Age 键的值为 : 7
print("Sex 键的值为 : %s" % dic.setdefault('Sex', None))  # Sex 键的值为 : None
print("新字典为：", dic)
# 新字典为：  {'Age': 7, 'Name': 'Lsgogroup', 'Sex': None}
```

1. `key in dict` `in` 操作符用于判断键是否存在于字典中，如果键在字典 dict 里返回 `true`，否则返回 `false`。而 `not in` 操作符刚好相反，如果键在字典 dict 里返回 `false`，否则返回 `true`。

【例子】

```python
dic = {'Name': 'Lsgogroup', 'Age': 7}

# in 检测键 Age 是否存在
if 'Age' in dic:
    print("键 Age 存在")
else:
    print("键 Age 不存在")

# 检测键 Sex 是否存在
```

```
if 'Sex' in dic:
    print("键 Sex 存在")
else:
    print("键 Sex 不存在")

# not in 检测键 Age 是否存在
if 'Age' not in dic:
    print("键 Age 不存在")
else:
    print("键 Age 存在")

# 键 Age 存在
# 键 Sex 不存在
# 键 Age 存在
```

1. `dict.pop(key[,default])` 删除字典给定键 `key` 所对应的值，返回值为被删除的值。`key` 值必须给出。若 `key` 不存在，则返回 `default` 值。
2. `del dict[key]` 删除字典给定键 `key` 所对应的值。

【例子】

```
dic1 = {1: "a", 2: [1, 2]}
print(dic1.pop(1), dic1)  # a {2: [1, 2]}

# 设置默认值，必须添加，否则报错
print(dic1.pop(3, "nokey"), dic1)  # nokey {2: [1, 2]}

del dic1[2]
print(dic1)  # {}
```

1. `dict.popitem()` 随机返回并删除字典中的一对键和值，如果字典已经为空，却调用了此方法，就报出KeyError异常。

【例子】

```
dic1 = {1: "a", 2: [1, 2]}
print(dic1.popitem())  # (1, 'a')
print(dic1)  # {2: [1, 2]}
```

1. `dict.clear()` 用于删除字典内所有元素。

【例子】

```
dic = {'Name': 'Zara', 'Age': 7}
print("字典长度 : %d" % len(dic))  # 字典长度 : 2
dict.clear()
print("字典删除后长度 : %d" % len(dic))  # 字典删除后长度 : 0
```

1. `dict.copy()` 返回一个字典的浅复制。

【例子】

```
dic1 = {'Name': 'Lsgogroup', 'Age': 7, 'Class': 'First'}
dic2 = dic1.copy()
print("新复制的字典为 : ", dic2)
# 新复制的字典为 :  {'Age': 7, 'Name': 'Lsgogroup', 'Class': 'First'}
```

【例子】直接赋值和 copy 的区别

```
dic1 = {'user': 'lsgogroup', 'num': [1, 2, 3]}

# 引用对象
dic2 = dic1
# 深拷贝父对象（一级目录），子对象（二级目录）不拷贝，还是引用
dic3 = dic1.copy()

print(id(dic1))  # 148635574728
print(id(dic2))  # 148635574728
print(id(dic3))  # 148635574344

# 修改 data 数据
dic1['user'] = 'root'
dic1['num'].remove(1)

# 输出结果
print(dic1)  # {'user': 'root', 'num': [2, 3]}
print(dic2)  # {'user': 'root', 'num': [2, 3]}
print(dic3)  # {'user': 'runoob', 'num': [2, 3]}
```

1. `dict.update(dict2)` 把字典参数 `dict2` 的 `key:value` 对 更新到字典 `dict` 里。

【例子】

```
dic = {'Name': 'Lsgogroup', 'Age': 7}
dic2 = {'Sex': 'female', 'Age': 8}
dic.update(dic2)
print("更新字典 dict : ", dic)
# 更新字典 dict :  {'Sex': 'female', 'Age': 8, 'Name': 'Lsgogroup'}
```

---

**练习题：**

1. 字典基本操作

   字典内容如下:

```
dic = {
    'python': 95,
    'java': 99,
    'c': 100
}
```

用程序解答下面的题目

字典的长度是多少

a. 请修改'java' 这个key对应的value值为98

b. 删除 c 这个key

c. 增加一个key-value对，key值为 php, value是90

d. 获取所有的key值，存储在列表里

e. 获取所有的value值，存储在列表里

f. 判断 javascript 是否在字典中

g. 获得字典里所有value 的和

h. 获取字典里最大的value

i. 获取字典里最小的value

j. 字典 dic1 = {'php': 97}， 将dic1的数据更新到dic中

2. 字典中的value

有一个字典，保存的是学生各个编程语言的成绩，内容如下

```
data = {
        'python': {'上学期': '90', '下学期': '95'},
        'c++': ['95', '96', '97'],
        'java': [{'月考':'90', '期中考试': '94', '期末考试': '98'}]
    }
```

各门课程的考试成绩存储方式并不相同，有的用字典，有的用列表，但是分数都是字符串类型，请实现函数`transfer_score(score_dict)`，将分数修改成int类型

```

def transfer_score(data):
    # your code here
```

# 11 集合

python 中`set`与`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`。

注意，`key`为不可变类型，即可哈希的值。

【例子】

```
num = {}
print(type(num))  # <class 'dict'>
num = {1, 2, 3, 4}
print(type(num))  # <class 'set'>
```

## 11.1 集合的创建

1. 先创建对象再加入元素。
2. 在创建空集合的时候只能使用`s = set()`，因为`s = {}`创建的是空字典。

【例子】

```
basket = set()
basket.add('apple')
basket.add('banana')
print(basket)  # {'banana', 'apple'}
```

1. 直接把一堆元素用花括号括起来`{元素1, 元素2, ..., 元素n}`。
2. 重复元素在`set`中会被自动被过滤。

【例子】

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)  # {'banana', 'apple', 'pear', 'orange'}
```

1. 使用`set(value)`工厂函数，把列表或元组转换成集合。

【例子】

```python
a = set('abracadabra')
print(a)
# {'r', 'b', 'd', 'c', 'a'}

b = set(("Google", "Lsgogroup", "Taobao", "Taobao"))
print(b)
# {'Taobao', 'Lsgogroup', 'Google'}

c = set(["Google", "Lsgogroup", "Taobao", "Google"])
print(c)
# {'Taobao', 'Lsgogroup', 'Google'}
```

1. 去掉列表中重复的元素

【例子】

```python
lst = [0, 1, 2, 3, 4, 5, 5, 3, 1]

temp = []
for item in lst:
    if item not in temp:
        temp.append(item)

print(temp)  # [0, 1, 2, 3, 4, 5]

a = set(lst)
print(list(a))  # [0, 1, 2, 3, 4, 5]
```

从结果发现集合的两个特点：无序 (unordered) 和唯一 (unique)。

由于 `set` 存储的是无序集合，所以我们不可以为集合创建索引或执行切片(slice)操作，也没有键(keys)可用来获取集合中元素的值，但是可以判断一个元素是否在集合中。

## 11.2 访问集合中的值

1. 可以使用`len()`内建函数得到集合的大小。

【例子】

```python
thisset = set(['Google', 'Baidu', 'Taobao'])
print(len(thisset))  # 3
```

1. 可以使用`for`把集合中的数据一个个读取出来。

【例子】

```
thisset = set(['Google', 'Baidu', 'Taobao'])
for item in thisset:
    print(item)

# Baidu
# Google
# Taobao
```

1. 可以通过`in`或`not in`判断一个元素是否在集合中已经存在

【例子】

```
thisset = set(['Google', 'Baidu', 'Taobao'])
print('Taobao' in thisset)  # True
print('Facebook' not in thisset)  # True
```

## 11.3 集合的内置方法

1. `set.add(elmnt)`用于给集合添加元素，如果添加的元素在集合中已存在，则不执行任何操作。

【例子】

```
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")
print(fruits)
# {'orange', 'cherry', 'banana', 'apple'}

fruits.add("apple")
print(fruits)
# {'orange', 'cherry', 'banana', 'apple'}
```

1. `set.update(set)`用于修改当前集合，可以添加新的元素或集合到当前集合中，如果添加的元素在集合中已存在，则该元素只会出现一次，重复的会忽略。

【例子】

```
x = {"apple", "banana", "cherry"}
y = {"google", "baidu", "apple"}
x.update(y)
print(x)
# {'cherry', 'banana', 'apple', 'google', 'baidu'}

y.update(["lsgo", "dreamtech"])
print(y)
# {'lsgo', 'baidu', 'dreamtech', 'apple', 'google'}
```

1. `set.remove(item)` 用于移除集合中的指定元素。如果元素不存在，则会发生错误。

【例子】

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")
print(fruits)  # {'apple', 'cherry'}
```

1. `set.discard(value)` 用于移除指定的集合元素。`remove()` 方法在移除一个不存在的元素时会发生错误，而 `discard()` 方法不会。

【例子】

```
fruits = {"apple", "banana", "cherry"}
fruits.discard("banana")
print(fruits)  # {'apple', 'cherry'}
```

1. `set.pop()` 用于随机移除一个元素。

【例子】

```
fruits = {"apple", "banana", "cherry"}
x = fruits.pop()
print(fruits)  # {'cherry', 'apple'}
print(x)  # banana
```

由于 set 是无序和无重复元素的集合，所以两个或多个 set 可以做数学意义上的集合操作。

1. `set.intersection(set1, set2 ...)` 返回两个集合的交集。
2. `set1 & set2` 返回两个集合的交集。
3. `set.intersection_update(set1, set2 ...)` 交集，在原始的集合上移除不重叠的元素。

【例子】

```
a = set('abracadabra')
b = set('alacazam')
print(a)  # {'r', 'a', 'c', 'b', 'd'}
print(b)  # {'c', 'a', 'l', 'm', 'z'}

c = a.intersection(b)
print(c)  # {'a', 'c'}
print(a & b)  # {'c', 'a'}
print(a)  # {'a', 'r', 'c', 'b', 'd'}

a.intersection_update(b)
print(a)  # {'a', 'c'}
```

1. `set.union(set1, set2...)` 返回两个集合的并集。
2. `set1 | set2` 返回两个集合的并集。

【例子】

```
a = set('abracadabra')
b = set('alacazam')
print(a)  # {'r', 'a', 'c', 'b', 'd'}
print(b)  # {'c', 'a', 'l', 'm', 'z'}

print(a | b)  # {'l', 'd', 'm', 'b', 'a', 'r', 'z', 'c'}
c = a.union(b)
print(c)  # {'c', 'a', 'd', 'm', 'r', 'b', 'z', 'l'}
```

1. `set.difference(set)` 返回集合的差集。
2. `set1 - set2` 返回集合的差集。
3. `set.difference_update(set)` 集合的差集，直接在原来的集合中移除元素，没有返回值。

【例子】

```
a = set('abracadabra')
b = set('alacazam')
print(a)  # {'r', 'a', 'c', 'b', 'd'}
print(b)  # {'c', 'a', 'l', 'm', 'z'}

c = a.difference(b)
print(c)  # {'b', 'd', 'r'}
print(a - b)  # {'d', 'b', 'r'}

print(a)  # {'r', 'd', 'c', 'a', 'b'}
a.difference_update(b)
print(a)  # {'d', 'r', 'b'}
```

1. `set.symmetric_difference(set)` 返回集合的异或。
2. `set1 ^ set2` 返回集合的异或。
3. `set.symmetric_difference_update(set)` 移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。

【例子】

```
a = set('abracadabra')
b = set('alacazam')
print(a)  # {'r', 'a', 'c', 'b', 'd'}
print(b)  # {'c', 'a', 'l', 'm', 'z'}

c = a.symmetric_difference(b)
print(c)  # {'m', 'r', 'l', 'b', 'z', 'd'}
print(a ^ b)  # {'m', 'r', 'l', 'b', 'z', 'd'}

print(a)  # {'r', 'd', 'c', 'a', 'b'}
a.symmetric_difference_update(b)
print(a)  # {'r', 'b', 'm', 'l', 'z', 'd'}
```

1. `set.issubset(set)` 判断集合是不是被其他集合包含，如果是则返回 True，否则返回 False。
2. `set1 <= set2` 判断集合是不是被其他集合包含，如果是则返回 True，否则返回 False。

【例子】

```
x = {"a", "b", "c"}
y = {"f", "e", "d", "c", "b", "a"}
z = x.issubset(y)
print(z)  # True
print(x <= y)  # True

x = {"a", "b", "c"}
y = {"f", "e", "d", "c", "b"}
z = x.issubset(y)
print(z)  # False
print(x <= y)  # False
```

1. `set.issuperset(set)` 用于判断集合是不是包含其他集合，如果是则返回 True，否则返回 False。
2. `set1 >= set2` 判断集合是不是包含其他集合，如果是则返回 True，否则返回 False。

【例子】

```
x = {"f", "e", "d", "c", "b", "a"}
y = {"a", "b", "c"}
z = x.issuperset(y)
print(z)  # True
print(x >= y)  # True

x = {"f", "e", "d", "c", "b"}
y = {"a", "b", "c"}
z = x.issuperset(y)
print(z)  # False
print(x >= y)  # False
```

1. `set.isdisjoint(set)` 用于判断两个集合是不是不相交，如果是返回 True，否则返回 False。

【例子】

```
x = {"f", "e", "d", "c", "b"}
y = {"a", "b", "c"}
z = x.isdisjoint(y)
print(z)  # False

x = {"f", "e", "d", "m", "g"}
y = {"a", "b", "c"}
z = x.isdisjoint(y)
print(z)  # True
```

## 11.4 集合的转换

【例子】

```
se = set(range(4))
li = list(se)
tu = tuple(se)

print(se, type(se))  # {0, 1, 2, 3} <class 'set'>
print(li, type(li))  # [0, 1, 2, 3] <class 'list'>
print(tu, type(tu))  # (0, 1, 2, 3) <class 'tuple'>
```

# 11.5 不可变集合

Python 提供了不能改变元素的集合的实现版本，即不能增加或删除元素，类型名叫`frozenset`。需要注意的是`frozenset`仍然可以进行集合操作，只是不能用带有`update`的方法。

1. `frozenset([iterable])` 返回一个冻结的集合，冻结后集合不能再添加或删除任何元素。

【例子】

```
a = frozenset(range(10))  # 生成一个新的不可变集合
print(a)
# frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

b = frozenset('lsgogroup')
print(b)
# frozenset({'g', 's', 'p', 'r', 'u', 'o', 'l'})
```

---

**参考文献：**


1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443
3. https://mp.weixin.qq.com/s/DZ589xEbOQ2QLtiq8mP1qQ


---

**练习题：**

1. 怎么表示只包含一个数字1的元组。
2. 创建一个空集合，增加 {‘x’,‘y’,‘z’} 三个元素。
3. 列表['A', 'B', 'A', 'B']去重。
4. 求两个集合{6, 7, 8}，{7, 8, 9}中不重复的元素
   差集指的是两个集合交集外的部分。
5. 求{'A', 'B', 'C'}中元素在 {'B', 'C', 'D'}中出现的次数。

# 12 序列

## 12.1 针对序列的内置函数

1. `list(sub)` 把一个可迭代对象转换为列表。

【例子】

```
a = list()
print(a)  # []

b = 'I Love LsgoGroup'
b = list(b)
print(b)
# ['I', ' ', 'L', 'o', 'v', 'e', ' ', 'L', 's', 'g', 'o', 'G', 'r', 'o', 'u', 'p']

c = (1, 1, 2, 3, 5, 8)
c = list(c)
print(c)  # [1, 1, 2, 3, 5, 8]
```

1. `tuple(sub)` 把一个可迭代对象转换为元组。

【例子】

```
a = tuple()
print(a)  # ()

b = 'I Love LsgoGroup'
b = tuple(b)
print(b)
# ('I', ' ', 'L', 'o', 'v', 'e', ' ', 'L', 's', 'g', 'o', 'G', 'r', 'o', 'u', 'p')

c = [1, 1, 2, 3, 5, 8]
c = tuple(c)
print(c)  # (1, 1, 2, 3, 5, 8)
```

1. `str(obj)` 把obj对象转换为字符串

【例子】

```python
a = 123
a = str(a)
print(a)  # 123
```

1. `len(s)` 返回对象（字符、列表、元组等）长度或元素个数。
    a. `s` -- 对象。

【例子】

```python
a = list()
print(len(a))  # 0

b = ('I', ' ', 'L', 'o', 'v', 'e', ' ', 'L', 's', 'g', 'o', 'G', 'r', 'o', 'u', 'p')
print(len(b))  # 16

c = 'I Love LsgoGroup'
print(len(c))  # 16
```

1. `max(sub)` 返回序列或者参数集合中的最大值

【例子】

```python
print(max(1, 2, 3, 4, 5))  # 5
print(max([-8, 99, 3, 7, 83]))  # 99
print(max('IloveLsgoGroup'))  # v
```

1. `min(sub)` 返回序列或参数集合中的最小值

【例子】

```python
print(min(1, 2, 3, 4, 5))  # 1
print(min([-8, 99, 3, 7, 83]))  # -8
print(min('IloveLsgoGroup'))  # G
```

1. `sum(iterable[, start=0])` 返回序列 `iterable` 与可选参数 `start` 的总和。

【例子】

```python
print(sum([1, 3, 5, 7, 9]))  # 25
print(sum([1, 3, 5, 7, 9], 10))  # 35
print(sum((1, 3, 5, 7, 9)))  # 25
print(sum((1, 3, 5, 7, 9), 20))  # 45
```

1. `sorted(iterable, key=None, reverse=False)` 对所有可迭代的对象进行排序操作。
    a. `iterable` -- 可迭代对象。
    b. `key` -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。
    c. `reverse` -- 排序规则， `reverse = True` 降序， `reverse = False` 升序（默认）。

d. 返回重新排序的列表。

【例子】

```
x = [-8, 99, 3, 7, 83]
print(sorted(x))  # [-8, 3, 7, 83, 99]
print(sorted(x, reverse=True))  # [99, 83, 7, 3, -8]

t = ({"age": 20, "name": "a"}, {"age": 25, "name": "b"}, {"age": 10, "name": "c"})
x = sorted(t, key=lambda a: a["age"])
print(x)
# [{'age': 10, 'name': 'c'}, {'age': 20, 'name': 'a'}, {'age': 25, 'name': 'b'}]
```

1. `reversed(seq)` 函数返回一个反转的迭代器。
    a. `seq` -- 要转换的序列，可以是 tuple, string, list 或 range。

【例子】

```
s = 'lsgogroup'
x = reversed(s)
print(type(x))  # <class 'reversed'>
print(x)  # <reversed object at 0x000002507E8EC2C8>
print(list(x))
# ['p', 'u', 'o', 'r', 'g', 'o', 'g', 's', 'l']

t = ('l', 's', 'g', 'o', 'g', 'r', 'o', 'u', 'p')
print(list(reversed(t)))
# ['p', 'u', 'o', 'r', 'g', 'o', 'g', 's', 'l']

r = range(5, 9)
print(list(reversed(r)))
# [8, 7, 6, 5]

x = [-8, 99, 3, 7, 83]
print(list(reversed(x)))
# [83, 7, 3, 99, -8]
```

1. `enumerate(sequence, [start=0])`

【例子】用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中。

```python
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
a = list(enumerate(seasons))
print(a)
# [(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]

b = list(enumerate(seasons, 1))
print(b)
# [(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]

for i, element in a:
    print('{0},{1}'.format(i, element))
```

1. `zip(iter1 [,iter2 [...]])`
   a. 用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的对象，这样做的好处是节约了不少的内存。
   b. 我们可以使用 `list()` 转换来输出列表。
   c. 如果各个迭代器的元素个数不一致，则返回列表长度与最短的对象相同，利用 `*` 号操作符，可以将元组解压为列表。

【例子】

```python
a = [1, 2, 3]
b = [4, 5, 6]
c = [4, 5, 6, 7, 8]

zipped = zip(a, b)
print(zipped)  # <zip object at 0x000000C5D89EDD88>
print(list(zipped))  # [(1, 4), (2, 5), (3, 6)]
zipped = zip(a, c)
print(list(zipped))  # [(1, 4), (2, 5), (3, 6)]

a1, a2 = zip(*zip(a, b))
print(list(a1))  # [1, 2, 3]
print(list(a2))  # [4, 5, 6]
```

---

**练习题：**

1. 怎么找出序列中的最大、小值？
2. sort() 和 sorted() 区别
3. 怎么快速求 1 到 100 所有整数相加之和？
4. 求列表 [2,3,4,5] 中每个元素的立方根。
5. 将[‘x’,‘y’,‘z’] 和 [1,2,3] 转成 [(‘x’,1),(‘y’,2),(‘z’,3)] 的形式。

# 13 函数与Lambda表达式

## 13.1 函数

还记得 Python 里面“万物皆对象”么？Python 把函数也当成对象，可以从另一个函数中返回出来而去构建高阶函数，比如：

1. 参数是函数
2. 返回值是函数

### 13.1.1 函数的定义

1. 函数以 `def` 关键词开头，后接函数名和圆括号()。
2. 函数执行的代码以冒号起始，并且缩进。
3. return [表达式] 结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 `None` 。

```
def functionname(parameters):
    "函数_文档字符串"
    function_suite
    return [expression]
```

### 13.1.2 函数的调用

【例子】

```
def printme(str):
    print(str)


printme("我要调用用户自定义函数!")  # 我要调用用户自定义函数!
printme("再次调用同一函数")  # 再次调用同一函数
temp = printme('hello') # hello
print(temp)  # None
```

【例子】

```python
def add(a, b):
    print(a + b)


add(1, 2)  # 3
add([1, 2, 3], [4, 5, 6]) # [1, 2, 3, 4, 5, 6]
```

## 13.1.3 函数文档

```python
def MyFirstFunction(name):
    "函数定义过程中name是形参"
    # 因为Ta只是一个形式，表示占据一个参数位置
    print('传递进来的{0}叫做实参，因为Ta是具体的参数值！'.format(name))


MyFirstFunction('老马的程序人生')
# 传递进来的老马的程序人生叫做实参，因为Ta是具体的参数值！

print(MyFirstFunction.__doc__)
# 函数定义过程中name是形参

help(MyFirstFunction)
# Help on function MyFirstFunction in module __main__:
# MyFirstFunction(name)
#     函数定义过程中name是形参
```

## 13.1.4 函数参数

Python 的函数具有非常灵活多样的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。从简到繁的参数形态如下：

1. 位置参数 (positional argument)
2. 默认参数 (default argument)
3. 可变参数 (variable argument)
4. 关键字参数 (keyword argument)
5. 命名关键字参数 (name keyword argument)
6. 参数组合

**1. 位置参数**

```
def functionname(arg1):
    "函数_文档字符串"
    function_suite
    return [expression]
```

1. `arg1` - 位置参数，这些参数在调用函数 (call function) 时位置要固定。

**2. 默认参数**

```
def functionname(arg1, arg2=v):
    "函数_文档字符串"
    function_suite
    return [expression]
```

1. `arg2 = v` - 默认参数 = 默认值，调用函数时，默认参数的值如果没有传入，则被认为是默认值。
2. 默认参数一定要放在位置参数 后面，不然程序会报错。

【例子】

```
def printinfo(name, age=8):
    print('Name:{0},Age:{1}'.format(name, age))


printinfo('小马')  # Name:小马,Age:8
printinfo('小马', 10)  # Name:小马,Age:10
```

1. Python 允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

【例子】

```
def printinfo(name, age):
    print('Name:{0},Age:{1}'.format(name, age))


printinfo(age=8, name='小马')  # Name:小马,Age:8
```

**3. 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 0, 1, 2 到任意个，是不定长的参数。

```
def functionname(arg1, arg2=v, *args):
    "函数_文档字符串"
    function_suite
    return [expression]
```

1. `*args` - 可变参数，可以是从零个到任意个，自动组装成元组。
2. 加了星号（*）的变量名会存放所有未命名的变量参数。

【例子】

```python
def printinfo(arg1, *args):
    print(arg1)
    for var in args:
        print(var)


printinfo(10)  # 10
printinfo(70, 60, 50)

# 70
# 60
# 50
```

### 4. 关键字参数

```python
def functionname(arg1, arg2=v, *args, **kw):
    "函数_文档字符串"
    function_suite
    return [expression]
```

1. `**kw` - 关键字参数，可以是从零个到任意个，自动组装成字典。

【例子】

```python
def printinfo(arg1, *args, **kwargs):
    print(arg1)
    print(args)
    print(kwargs)


printinfo(70, 60, 50)
# 70
# (60, 50)
# {}
printinfo(70, 60, 50, a=1, b=2)
# 70
# (60, 50)
# {'a': 1, 'b': 2}
```

「可变参数」和「关键字参数」的同异总结如下：

1. 可变参数允许传入零个到任意个参数，它们在函数调用时自动组装为一个元组 (tuple)。
2. 关键字参数允许传入零个到任意个参数，它们在函数内部自动组装为一个字典 (dict)。

### 5. 命名关键字参数

```
def functionname(arg1, arg2=v, *args, *, nkw, **kw):
    "函数_文档字符串"
    function_suite
    return [expression]
```

1. `*, nkw` - 命名关键字参数，用户想要输入的关键字参数，定义方式是在nkw 前面加个分隔符 *。
2. 如果要限制关键字参数的名字，就可以用「命名关键字参数」
3. 使用命名关键字参数时，要特别注意不能缺少参数名。

【例子】

```
def printinfo(arg1, *, nkw, **kwargs):
    print(arg1)
    print(nkw)
    print(kwargs)


printinfo(70, nkw=10, a=1, b=2)
# 70
# 10
# {'a': 1, 'b': 2}

printinfo(70, 10, a=1, b=2)
# TypeError: printinfo() takes 1 positional argument but 2 were given
```

1. 没有写参数名`nwk`，因此 10 被当成「位置参数」，而原函数只有 1 个位置函数，现在调用了 2 个，因此程序会报错。

### 6. 参数组合

在 Python 中定义函数，可以用位置参数、默认参数、可变参数、命名关键字参数和关键字参数，这 5 种参数中的 4 个都可以一起使用，但是注意，参数定义的顺序必须是：

1. 位置参数、默认参数、可变参数和关键字参数。
2. 位置参数、默认参数、命名关键字参数和关键字参数。

要注意定义可变参数和关键字参数的语法：

1. `*args` 是可变参数，`args` 接收的是一个 `tuple`
2. `**kw` 是关键字参数，`kw` 接收的是一个 `dict`

命名关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。定义命名关键字参数不要忘了写分隔符 `*`，否则定义的是位置参数。

警告：虽然可以组合多达 5 种参数，但不要同时使用太多的组合，否则函数很难懂。

## 13.1.5 函数的返回值

【例子】

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


print(add(1, 2))  # 3
print(add([1, 2, 3], [4, 5, 6]))  # [1, 2, 3, 4, 5, 6]
```

【例子】

```
def back():
    return [1, '小马的程序人生', 3.14]


print(back())  # [1, '小马的程序人生', 3.14]
```

【例子】

```
def back():
    return 1, '小马的程序人生', 3.14


print(back())  # (1, '小马的程序人生', 3.14)
```

【例子】

```
def printme(str):
    print(str)

temp = printme('hello') # hello
print(temp) # None
print(type(temp))  # <class 'NoneType'>
```

## 13.1.6 变量作用域

1. Python 中，程序的变量并不是在哪个位置都可以访问的，访问权限决定于这个变量是在哪里赋值的。
2. 定义在函数内部的变量拥有局部作用域，该变量称为局部变量。
3. 定义在函数外部的变量拥有全局作用域，该变量称为全局变量。
4. 局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。

【例子】

```python
def discounts(price, rate):
    final_price = price * rate
    return final_price


old_price = float(input('请输入原价:'))  # 98
rate = float(input('请输入折扣率:'))  # 0.9
new_price = discounts(old_price, rate)
print('打折后价格是:%.2f' % new_price)  # 88.20
```

1. 当内部作用域想修改外部作用域的变量时，就要用到`global`和`nonlocal`关键字了。

【例子】

```python
num = 1


def fun1():
    global num  # 需要使用 global 关键字声明
    print(num)  # 1
    num = 123
    print(num)  # 123


fun1()
print(num)  # 123
```

**内嵌函数**

【例子】

```python
def outer():
    print('outer函数在这被调用')

    def inner():
        print('inner函数在这被调用')

    inner()  # 该函数只能在outer函数内部被调用


outer()
# outer函数在这被调用
# inner函数在这被调用
```

**闭包**

1. 是函数式编程的一个重要的语法结构，是一种特殊的内嵌函数。
2. 如果在一个内部函数里对外层非全局作用域的变量进行引用，那么内部函数就被认为是闭包。

3. 通过闭包可以访问外层非全局作用域的变量，这个作用域称为 **闭包作用域**。

【例子】

```python
def funX(x):
    def funY(y):
        return x * y

    return funY


i = funX(8)
print(type(i))  # <class 'function'>
print(i(5))  # 40
```

【例子】闭包的返回值通常是函数。

```python
def make_counter(init):
    counter = [init]

    def inc(): counter[0] += 1

    def dec(): counter[0] -= 1

    def get(): return counter[0]

    def reset(): counter[0] = init

    return inc, dec, get, reset


inc, dec, get, reset = make_counter(0)
inc()
inc()
inc()
print(get())  # 3
dec()
print(get())  # 2
reset()
print(get())  # 0
```

【例子】如果要修改闭包作用域中的变量则需要 `nonlocal` 关键字

```python
def outer():
    num = 10

    def inner():
        nonlocal num  # nonlocal关键字声明
```

```
        num = 100
        print(num)

    inner()
    print(num)


outer()

# 100
# 100
```

**递归**

1. 如果一个函数在内部调用自身本身，这个函数就是递归函数。

【例子】`n! = 1 x 2 x 3 x ... x n`

++循环++

```
n = 5
for k in range(1, 5):
    n = n * k
print(n)  # 120
```

++递归++

```
def factorial(n):
    if n == 1:
        return 1
    return n * fact(n - 1)


print(factorial(5)) # 120
```

【例子】斐波那契数列 `f(n)=f(n-1)+f(n-2), f(0)=0 f(1)=1`

++循环++

```
i = 0
j = 1
lst = list([i, j])
for k in range(2, 11):
    k = i + j
    lst.append(k)
    i = j
    j = k
print(lst)
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

++递归++

```
def recur_fibo(n):
    if n <= 1:
        return n
    return recur_fibo(n - 1) + recur_fibo(n - 2)


lst = list()
for k in range(11):
    lst.append(recur_fibo(k))
print(lst)
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

【例子】设置递归的层数，Python默认递归层数为 100

```
import sys

sys.setrecursionlimit(1000)
```

---

# 13.2 Lambda 表达式

## 13.2.1 匿名函数的定义

在 Python 里有两类函数：

1. 第一类：用 `def` 关键词定义的正规函数
2. 第二类：用 `lambda` 关键词定义的匿名函数

python 使用 `lambda` 关键词来创建匿名函数，而非 `def` 关键词，它没有函数名，其语法结构如下：

```
lambda argument_list: expression
```

1. `lambda` - 定义匿名函数的关键词。
2. `argument_list` - 函数参数，它们可以是位置参数、默认参数、关键字参数，和正规函数里的参数类型一样。
3. `:` - 冒号，在函数参数和表达式中间要加个冒号。
4. `expression` - 只是一个表达式，输入函数参数，输出一些值。

注意：

1. `expression` 中没有 return 语句，因为 lambda 不需要它来返回，表达式本身结果就是返回值。
2. 匿名函数拥有自己的命名空间，且不能访问自己参数列表之外或全局命名空间里的参数。

【例子】

```
def sqr(x):
    return x ** 2


print(sqr)
# <function sqr at 0x000000BABD3A4400>

y = [sqr(x) for x in range(10)]
print(y)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

lbd_sqr = lambda x: x ** 2
print(lbd_sqr)
# <function <lambda> at 0x000000BABB6AC1E0>

y = [lbd_sqr(x) for x in range(10)]
print(y)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]


sumary = lambda arg1, arg2: arg1 + arg2
print(sumary(10, 20))  # 30

func = lambda *args: sum(args)
print(func(1, 2, 3, 4, 5))  # 15
```

## 13.2.2 匿名函数的应用

函数式编程 是指代码中每一块都是不可变的，都由纯函数的形式组成。这里的纯函数，是指函数本身相互独立、互不影响，对于相同的输入，总会有相同的输出，没有任何副作用。

【例子】非函数式编程

```python
def f(x):
    for i in range(0, len(x)):
        x[i] += 10
    return x


x = [1, 2, 3]
f(x)
print(x)
# [11, 12, 13]
```

【例子】函数式编程

```python
def f(x):
    y = []
    for item in x:
        y.append(item + 10)
    return y


x = [1, 2, 3]
f(x)
print(x)
# [1, 2, 3]
```

匿名函数 常常应用于函数式编程的高阶函数 (high-order function)中，主要有两种形式：

1. 参数是函数 (filter, map)
2. 返回值是函数 (closure)

如，在 `filter` 和 `map` 函数中的应用：

1. `filter(function, iterable)` 过滤序列，过滤掉不符合条件的元素，返回一个迭代器对象，如果要转换为列表，可以使用 `list()` 来转换。

【例子】

```python
odd = lambda x: x % 2 == 1
templist = filter(odd, [1, 2, 3, 4, 5, 6, 7, 8, 9])
print(list(templist))  # [1, 3, 5, 7, 9]
```

1. `map(function, *iterables)` 根据提供的函数对指定序列做映射。

【例子】

```python
m1 = map(lambda x: x ** 2, [1, 2, 3, 4, 5])
print(list(m1))
# [1, 4, 9, 16, 25]

m2 = map(lambda x, y: x + y, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])
print(list(m2))
# [3, 7, 11, 15, 19]
```

除了 Python 这些内置函数，我们也可以自己定义高阶函数。

【例子】

```python
def apply_to_list(fun, some_list):
    return fun(some_list)

lst = [1, 2, 3, 4, 5]
print(apply_to_list(sum, lst))
# 15

print(apply_to_list(len, lst))
# 5

print(apply_to_list(lambda x: sum(x) / len(x), lst))
# 3.0
```

---

**参考文献：**


1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443
3. https://mp.weixin.qq.com/s/gKhXS8JVU8dZBHJF7slFsw

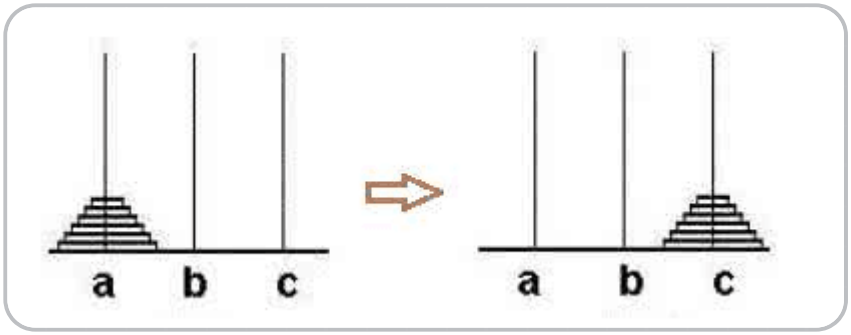

---

**练习题：**

1. 怎么给函数编写文档？
2. 怎么给函数参数和返回值注解？
3. 闭包中，怎么对数字、字符串、元组等不可变元素更新
4. 分别根据每一行的首元素和尾元素大小对二维列表 [[6, 5], [3, 7], [2, 8]] 排序。(利用lambda表达式)

   a = [[6, 5], [3, 7], [2, 8]]
5. 利用python解决汉诺塔问题？

   有a、b、c三根柱子，在a柱子上从下往上按照大小顺序摞着64片圆盘，把圆盘从下面开始按大小顺序重新摆放在c柱子上，尝试用函数来模拟解决的过程。（提示：将问题简化为已经成功地将a柱上面的63个盘子移到了b柱）


a
b
c
a
b
c

# 14 类与对象

## 14.1 对象 = 属性 + 方法

对象是类的实例。换句话说，类主要定义对象的结构，然后我们以类为模板创建对象。类不但包含方法定义，而且还包含所有实例共享的数据。

1. 封装：信息隐蔽技术

我们可以使用关键字 `class` 定义 Python 类，关键字后面紧跟类的名称、分号和类的实现。

【例子】

```
class Turtle:  # Python中的类名约定以大写字母开头
    """关于类的一个简单例子"""
    # 属性
    color = 'green'
    weight = 10
    legs = 4
    shell = True
    mouth = '大嘴'

    # 方法
    def climb(self):
        print('我正在很努力的向前爬...')

    def run(self):
        print('我正在飞快的向前跑...')

    def bite(self):
        print('咬死你咬死你!!')

    def eat(self):
        print('有得吃，真满足...')

    def sleep(self):
        print('困了，睡了，晚安，zzz')


```

```
tt = Turtle()
print(tt)
# <__main__.Turtle object at 0x0000007C32D67F98>

print(type(tt))
# <class '__main__.Turtle'>

print(tt.__class__)
# <class '__main__.Turtle'>

print(tt.__class__.__name__)
# Turtle

tt.climb()
# 我正在很努力的向前爬...

tt.run()
# 我正在飞快的向前跑...

tt.bite()
# 咬死你咬死你!!

# Python类也是对象。它们是type的实例
print(type(Turtle))
# <class 'type'>
```

1. 继承：子类自动共享父类之间数据和方法的机制

【例子】

```
class MyList(list):
    pass


lst = MyList([1, 5, 2, 7, 8])
lst.append(9)
lst.sort()
print(lst)

# [1, 2, 5, 7, 8, 9]
```

1. 多态：不同对象对同一方法响应不同的行动

【例子】

```
class Animal:
    def run(self):
```

```
        raise AttributeError('子类必须实现这个方法')


class People(Animal):
    def run(self):
        print('人正在走')


class Pig(Animal):
    def run(self):
        print('pig is walking')


class Dog(Animal):
    def run(self):
        print('dog is running')


def func(animal):
    animal.run()


func(Pig())
# pig is walking
```

## 14.2 self 是什么？

Python 的 `self` 相当于 C++ 的 `this` 指针。

【例子】

```
class Test:
    def prt(self):
        print(self)
        print(self.__class__)


t = Test()
t.prt()
# <__main__.Test object at 0x000000BC5A351208>
# <class '__main__.Test'>
```

类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数名称（对应于该实例，即该对象本身），按照惯例它的名称是 `self`。在调用方法时，我们无需明确提供与参数 `self` 相对应的参数。

【例子】

```
class Ball:
    def setName(self, name):
        self.name = name

    def kick(self):
        print("我叫%s,该死的，谁踢我..." % self.name)


a = Ball()
a.setName("球A")
b = Ball()
b.setName("球B")
c = Ball()
c.setName("球C")
a.kick()
# 我叫球A,该死的，谁踢我...
b.kick()
# 我叫球B,该死的，谁踢我...
```

## 14.3 Python 的魔法方法

据说，Python 的对象天生拥有一些神奇的方法，它们是面向对象的 Python 的一切...

它们是可以给你的类增加魔力的特殊方法...

如果你的对象实现了这些方法中的某一个，那么这个方法就会在特殊的情况下被 Python 所调用，而这一切都是自动发生的...

类有一个名为`__init__(self[, param1, param2...])`的魔法方法，该方法在类实例化时会自动调用。

【例子】

```

class Ball:
    def __init__(self, name):
        self.name = name

    def kick(self):
```

```
        print("我叫%s,该死的，谁踢我..." % self.name)


a = Ball("球A")
b = Ball("球B")
c = Ball("球C")
a.kick()
# 我叫球A,该死的，谁踢我...
b.kick()
# 我叫球B,该死的，谁踢我...
```

## 14.4 公有和私有

在 Python 中定义私有变量只需要在变量名或函数名前加上“__”两个下划线，那么这个函数或变量就会为私有的了。

【例子】类的私有属性实例

```
class JustCounter:
    __secretCount = 0  # 私有变量
    publicCount = 0  # 公开变量

    def count(self):
        self.__secretCount += 1
        self.publicCount += 1
        print(self.__secretCount)


counter = JustCounter()
counter.count()  # 1
counter.count()  # 2
print(counter.publicCount)  # 2

print(counter._JustCounter__secretCount)  # 2 Python的私有为伪私有
print(counter.__secretCount)
# AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

【例子】类的私有方法实例

```
class Site:
    def __init__(self, name, url):
        self.name = name  # public
        self.__url = url  # private

```

```
    def who(self):
        print('name  : ', self.name)
        print('url : ', self.__url)

    def __foo(self):  # 私有方法
        print('这是私有方法')

    def foo(self):  # 公共方法
        print('这是公共方法')
        self.__foo()


x = Site('老马的程序人生', 'https://blog.csdn.net/LSGO_MYP')
x.who()
# name  :  老马的程序人生
# url :  https://blog.csdn.net/LSGO_MYP

x.foo()
# 这是公共方法
# 这是私有方法

x.__foo()
# AttributeError: 'Site' object has no attribute '__foo'
```

## 14.5 继承

Python 同样支持类的继承，派生类的定义如下所示：

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

`BaseClassName`（示例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用：

```
class DerivedClassName(modname.BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

【例子】如果子类中定义与父类同名的方法或属性，则会自动覆盖父类对应的方法或属性。

```
# 类定义
class people:
    # 定义基本属性
    name = ''
    age = 0
    # 定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0

    # 定义构造方法
    def __init__(self, n, a, w):
        self.name = n
        self.age = a
        self.__weight = w

    def speak(self):
        print("%s 说: 我 %d 岁。" % (self.name, self.age))


# 单继承示例
class student(people):
    grade = ''

    def __init__(self, n, a, w, g):
        # 调用父类的构函
        people.__init__(self, n, a, w)
        self.grade = g

    # 覆写父类的方法
    def speak(self):
        print("%s 说: 我 %d 岁了, 我在读 %d 年级" % (self.name, self.age, self.grade))


s = student('小马的程序人生', 10, 60, 3)
s.speak()
# 小马的程序人生 说: 我 10 岁了, 我在读 3 年级
```

注意：如果上面的程序去掉：`people.__init__(self, n, a, w)`，则输出：`说: 我 0 岁了, 我在读 3 年级`，因为子类的构造方法把父类的构造方法覆盖了。

【例子】

```
class Fish:
    def __init__(self):
        self.x = r.randint(0, 10)
        self.y = r.randint(0, 10)

    def move(self):
        self.x -= 1
        print("我的位置", self.x, self.y)


class GoldFish(Fish):  # 金鱼
    pass


class Carp(Fish):  # 鲤鱼
    pass


class Salmon(Fish):  # 三文鱼
    pass


class Shark(Fish):  # 鲨鱼
    def __init__(self):
        self.hungry = True

    def eat(self):
        if self.hungry:
            print("吃货的梦想就是天天有得吃！")
            self.hungry = False
        else:
            print("太撑了，吃不下了！")
            self.hungry = True


g = GoldFish()
g.move()  # 我的位置 9 4
s = Shark()
s.eat() # 吃货的梦想就是天天有得吃！
s.move()
# AttributeError: 'Shark' object has no attribute 'x'
```

解决该问题可用以下两种方式：

1. 调用未绑定的父类方法`Fish.__init__(self)`

```
class Shark(Fish):  # 鲨鱼
    def __init__(self):
        Fish.__init__(self)
        self.hungry = True

    def eat(self):
        if self.hungry:
            print("吃货的梦想就是天天有得吃！")
            self.hungry = False
        else:
            print("太撑了，吃不下了！")
            self.hungry = True
```

1. 使用super函数`super().__init__()`

```
class Shark(Fish):  # 鲨鱼
    def __init__(self):
        super().__init__()
        self.hungry = True

    def eat(self):
        if self.hungry:
            print("吃货的梦想就是天天有得吃！")
            self.hungry = False
        else:
            print("太撑了，吃不下了！")
            self.hungry = True
```

Python 虽然支持多继承的形式，但我们一般不使用多继承，因为容易引起混乱。

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

需要注意圆括号中父类的顺序，若是父类中有相同的方法名，而在子类使用时未指定，Python 从左至右搜索，即方法在子类中未找到时，从左到右查找父类中是否包含方法。

```
# 类定义
class People:
    # 定义基本属性
    name = ''
    age = 0
    # 定义私有属性,私有属性在类外部无法直接进行访问
```

```
    __weight = 0

    # 定义构造方法
    def __init__(self, n, a, w):
        self.name = n
        self.age = a
        self.__weight = w

    def speak(self):
        print("%s 说: 我 %d 岁。" % (self.name, self.age))


# 单继承示例
class Student(People):
    grade = ''

    def __init__(self, n, a, w, g):
        # 调用父类的构函
        People.__init__(self, n, a, w)
        self.grade = g

    # 覆写父类的方法
    def speak(self):
        print("%s 说: 我 %d 岁了，我在读 %d 年级" % (self.name, self.age, self.grade))


# 另一个类，多重继承之前的准备
class Speaker:
    topic = ''
    name = ''

    def __init__(self, n, t):
        self.name = n
        self.topic = t

    def speak(self):
        print("我叫 %s，我是一个演说家，我演讲的主题是 %s" % (self.name, self.topic))


# 多重继承
class Sample01(Speaker, Student):
    a = ''

    def __init__(self, n, a, w, g, t):
        Student.__init__(self, n, a, w, g)
```

```
        Speaker.__init__(self, n, t)


test = Sample01("Tim", 25, 80, 4, "Python")
test.speak()  # 方法名同，默认调用的是在括号中排前地父类的方法


# 我叫 Tim，我是一个演说家，我演讲的主题是 Python

class Sample02(Student, Speaker):
    a = ''

    def __init__(self, n, a, w, g, t):
        Student.__init__(self, n, a, w, g)
        Speaker.__init__(self, n, t)


test = Sample02("Tim", 25, 80, 4, "Python")
test.speak()  # 方法名同，默认调用的是在括号中排前地父类的方法

# Tim 说：我 25 岁了，我在读 4 年级
```

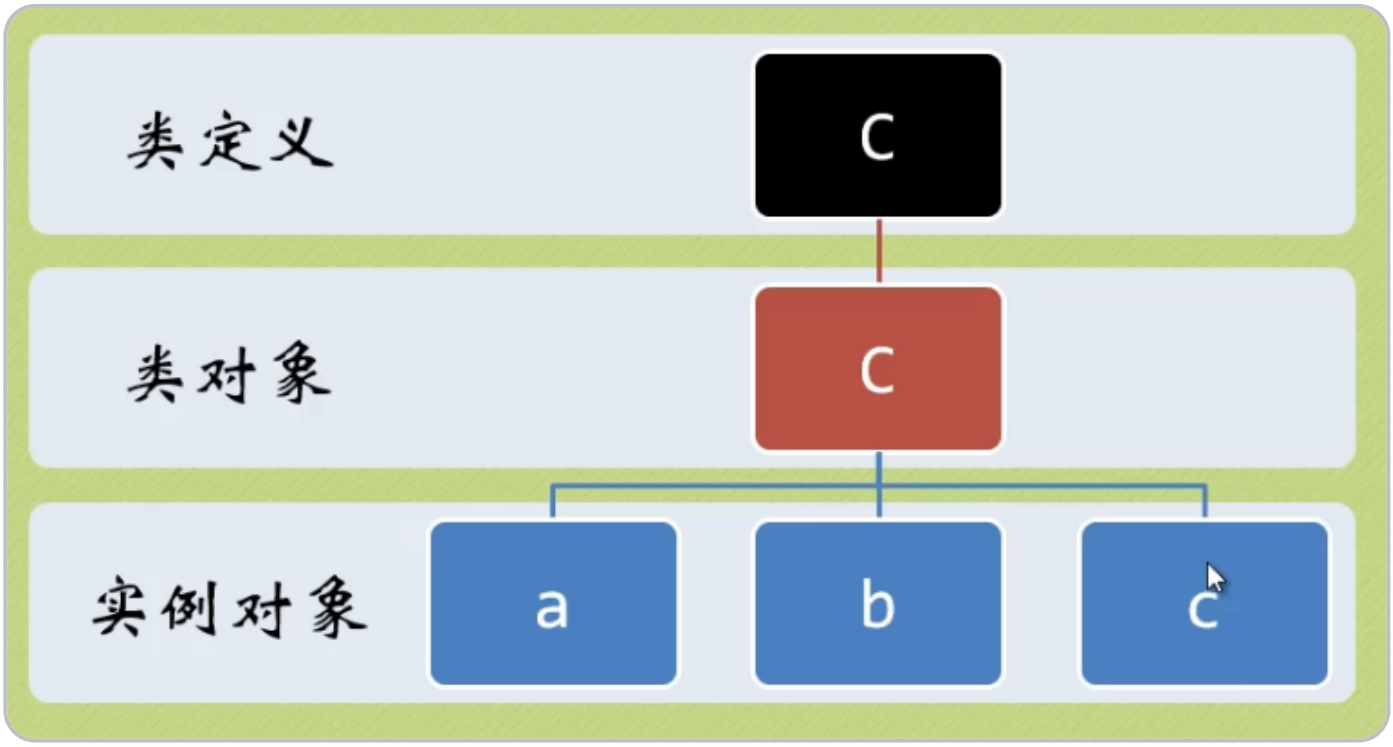
## 14.6 组合

【例子】

```
class Turtle:
    def __init__(self, x):
        self.num = x


class Fish:
    def __init__(self, x):
        self.num = x


class Pool:
    def __init__(self, x, y):
        self.turtle = Turtle(x)
        self.fish = Fish(y)

```

```
    def print_num(self):
        print("水池里面有乌龟%s只，小鱼%s条" % (self.turtle.num, self.fish.num))


p = Pool(2, 3)
p.print_num()
# 水池里面有乌龟2只，小鱼3条
```

## 14.7 类、类对象和实例对象




类对象：创建一个类，其实也是一个对象也在内存开辟了一块空间，称为类对象，类对象只有一个。

```
# 类对象
class A(object):
    pass
```

实例对象：就是通过实例化类创建的对象，称为实例对象，实例对象可以有多个。

【例子】

```
# 实例化对象 a、b、c都属于实例对象。
a = A()
b = A()
c = A()
```

类属性：类里面方法外面定义的变量称为类属性。类属性所属于类对象并且多个实例对象之间共享同一个类属性，说白了就是类属性所有的通过该类实例化的对象都能共享。

【例子】

```
class A():
    a = xx  #类属性
    def __init__(self):
        A.a = xx  #使用类属性可以通过 （类名.类属性）调用。
```

实例属性：实例属性和具体的某个实例对象有关系，并且一个实例对象和另外一个实例对象是不共享属性的，说白了实例属性只能在自己的对象里面使用，其他的对象不能直接使用，因为`self`是谁调用，它的值就属于该对象。

【例子】

```
class 类名():
    __init__(self):
        self.name = xx #实例属性
```

类属性和实例属性区别

1. 类属性：类外面，可以通过`实例对象.类属性`和`类名.类属性`进行调用。类里面，通过`self.类属性`和`类名.类属性`进行调用。
2. 实例属性：类外面，可以通过`实例对象.实例属性`调用。类里面，通过`self.实例属性`调用。
3. 实例属性就相当于局部变量。出了这个类或者这个类的实例对象，就没有作用了。
4. 类属性就相当于类里面的全局变量，可以和这个类的所有实例对象共享。

【例子】

```
# 创建类对象
class Test(object):
    class_attr = 100  # 类属性

    def __init__(self):
        self.sl_attr = 100  # 实例属性

    def func(self):
        print('类对象.类属性的值:', Test.class_attr)  # 调用类属性
        print('self.类属性的值', self.class_attr)  # 相当于把类属性 变成实例属性
        print('self.实例属性的值', self.sl_attr)  # 调用实例属性


a = Test()
a.func()

# 类对象.类属性的值: 100
# self.类属性的值 100
# self.实例属性的值 100

```

```
b = Test()
b.func()

# 类对象.类属性的值: 100
# self.类属性的值 100
# self.实例属性的值 100

a.class_attr = 200
a.sl_attr = 200
a.func()

# 类对象.类属性的值: 100
# self.类属性的值 200
# self.实例属性的值 200

b.func()

# 类对象.类属性的值: 100
# self.类属性的值 100
# self.实例属性的值 100

Test.class_attr = 300
a.func()

# 类对象.类属性的值: 300
# self.类属性的值 200
# self.实例属性的值 200

b.func()
# 类对象.类属性的值: 300
# self.类属性的值 300
# self.实例属性的值 100
```

注意：属性与方法名相同，属性会覆盖方法。

【例子】

```
class A:
    def x(self):
        print('x_man')


aa = A()
aa.x()  # x_man
aa.x = 1
print(aa.x)  # 1
aa.x()
# TypeError: 'int' object is not callable
```

## 14.8 什么是绑定？

Python 严格要求方法需要有实例才能被调用，这种限制其实就是 Python 所谓的绑定概念。

Python 对象的数据属性通常存储在名为`.__ dict__`的字典中，我们可以直接访问`__dict__`，或利用 Python 的内置函数`vars()`获取`.__ dict__`。

【例子】

```
class CC:
    def setXY(self, x, y):
        self.x = x
        self.y = y

    def printXY(self):
        print(self.x, self.y)


dd = CC()
print(dd.__dict__)
# {}

print(vars(dd))
# {}

print(CC.__dict__)
# {'__module__': '__main__', 'setXY': <function CC.setXY at 0x000000C3473DA048>, 'printXY':
<function CC.printXY at 0x000000C3473C4F28>, '__dict__': <attribute '__dict__' of 'CC' objects>,
'__weakref__': <attribute '__weakref__' of 'CC' objects>, '__doc__': None}

```

```
dd.setXY(4, 5)
print(dd.__dict__)
# {'x': 4, 'y': 5}

print(vars(CC))
# {'__module__': '__main__', 'setXY': <function CC.setXY at 0x000000632CA9B048>, 'printXY':
<function CC.printXY at 0x000000632CA83048>, '__dict__': <attribute '__dict__' of 'CC' objects>,
'__weakref__': <attribute '__weakref__' of 'CC' objects>, '__doc__': None}

print(CC.__dict__)
# {'__module__': '__main__', 'setXY': <function CC.setXY at 0x000000632CA9B048>, 'printXY':
<function CC.printXY at 0x000000632CA83048>, '__dict__': <attribute '__dict__' of 'CC' objects>,
'__weakref__': <attribute '__weakref__' of 'CC' objects>, '__doc__': None}
```

## 14.9 一些相关的内置函数（BIF）

1. `issubclass(class, classinfo)` 方法用于判断参数 class 是否是类型参数 classinfo 的子类。
2. 一个类被认为是其自身的子类。
3. `classinfo` 可以是类对象的元组，只要class是其中任何一个候选类的子类，则返回 `True` 。

【例子】

```
class A:
    pass


class B(A):
    pass


print(issubclass(B, A))  # True
print(issubclass(B, B))  # True
print(issubclass(A, B))  # False
print(issubclass(B, object))  # True
```

1. `isinstance(object, classinfo)` 方法用于判断一个对象是否是一个已知的类型，类似 `type()` 。
2. `type()` 不会认为子类是一种父类类型，不考虑继承关系。
3. `isinstance()` 会认为子类是一种父类类型，考虑继承关系。
4. 如果第一个参数不是对象，则永远返回 `False` 。
5. 如果第二个参数不是类或者由类对象组成的元组，会抛出一个 `TypeError` 异常。

【例子】

```
a = 2
print(isinstance(a, int))  # True
print(isinstance(a, str))  # False
print(isinstance(a, (str, int, list)))  # True


class A:
    pass


class B(A):
    pass


print(isinstance(A(), A))  # True
print(type(A()) == A)  # True
print(isinstance(B(), A))  # True
print(type(B()) == A)  # False
```

1. `hasattr(object, name)` 用于判断对象是否包含对应的属性。

【例子】

```
class Coordinate:
    x = 10
    y = -5
    z = 0


point1 = Coordinate()
print(hasattr(point1, 'x'))  # True
print(hasattr(point1, 'y'))  # True
print(hasattr(point1, 'z'))  # True
print(hasattr(point1, 'no'))  # False
```

1. `getattr(object, name[, default])` 用于返回一个对象属性值。

【例子】

```python
class A(object):
    bar = 1



a = A()
print(getattr(a, 'bar'))  # 1
print(getattr(a, 'bar2', 3))  # 3
print(getattr(a, 'bar2'))
# AttributeError: 'A' object has no attribute 'bar2'
```

【例子】这个例子很酷！

```python
class A(object):
    def set(self, a, b):
        x = a
        a = b
        b = x
        print(a, b)



a = A()
c = getattr(a, 'set')
c(a='1', b='2')  # 2 1

```

1. `setattr(object, name, value)`对应函数 `getattr()`，用于设置属性值，该属性不一定是存在的。

【例子】

```python
class A(object):
    bar = 1



a = A()
print(getattr(a, 'bar'))  # 1
setattr(a, 'bar', 5)
print(a.bar)  # 5
setattr(a, "age", 28)
print(a.age)  # 28
```

1. `delattr(object, name)`用于删除属性。

【例子】

```python
class Coordinate:
    x = 10
    y = -5
    z = 0
```

```
5
6
7  point1 = Coordinate()
8
9  print('x = ', point1.x)  # x =  10
10 print('y = ', point1.y)  # y =  -5
11 print('z = ', point1.z)  # z =  0
12
13 delattr(Coordinate, 'z')
14
15 print('--删除 z 属性后--')  # --删除 z 属性后--
16 print('x = ', point1.x)  # x =  10
17 print('y = ', point1.y)  # y =  -5
18
19 # 触发错误
20 print('z = ', point1.z)
21 # AttributeError: 'Coordinate' object has no attribute 'z'
```

1. `class property([fget[, fset[, fdel[, doc]]]])`用于在新式类中返回属性值。
   a. `fget` -- 获取属性值的函数
   b. `fset` -- 设置属性值的函数
   c. `fdel` -- 删除属性值函数
   d. `doc` -- 属性描述信息

【例子】

```
1  class C(object):
2      def __init__(self):
3          self.__x = None
4
5      def getx(self):
6          return self.__x
7
8      def setx(self, value):
9          self.__x = value
10
11     def delx(self):
12         del self.__x
13
14     x = property(getx, setx, delx, "I'm the 'x' property.")
15
16
17 cc = C()
18 cc.x = 2
19 print(cc.x)  # 2
```

**参考文献：**


1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443
3. https://www.cnblogs.com/loved/p/8678919.html
4. https://www.runoob.com/python3/python3-class.html
5. https://www.jianshu.com/p/9fb316cbf42e


---

**练习题：**

1. 以下类定义中哪些是类属性，哪些是实例属性？
2. 怎么定义私有方法？
3. 尝试执行以下代码，并解释错误原因：

```
class C:
    def myFun():
        print('Hello!')
c = C()
c.myFun()
```

4. 按照以下要求定义一个游乐园门票的类，并尝试计算2个成人+1个小孩平日票价。

   要求:

   1. 平日票价100元
   2. 周末票价为平日的120%
   3. 儿童票半价

```
class Ticket():
    # your code here
```

# 15 魔法方法

魔法方法总是被双下划线包围，例如`__init__`。

魔法方法是面向对象的 Python 的一切，如果你不知道魔法方法，说明你还没能意识到面向对象的 Python 的强大。

魔法方法的“魔力”体现在它们总能够在适当的时候被自动调用。

魔法方法的第一个参数应为`cls`（类方法） 或者`self`（实例方法）。

1. `cls`：代表一个类的名称
2. `self`：代表一个实例对象的名称

## 15.1 基本的魔法方法

**`__init__(self[, ...])`**

1. 构造器，当一个实例被创建的时候调用的初始化方法

【例子】

```
class Rectangle:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def getPeri(self):
        return (self.x + self.y) * 2

    def getArea(self):
        return self.x * self.y


rect = Rectangle(4, 5)
print(rect.getPeri())  # 18
print(rect.getArea())  # 20
```

`__new__(cls[, ...])`

1. `__new__` 是在一个对象实例化的时候所调用的第一个方法，在调用 `__init__` 初始化前，先调用 `__new__`。
2. `__new__` 至少要有一个参数 `cls`，代表要实例化的类，此参数在实例化时由 Python 解释器自动提供，后面的参数直接传递给 `__init__`。
3. `__new__` 对当前类进行了实例化，并将实例返回，传给 `__init__` 的 `self`。但是，执行了 `__new__`，并不一定会进入 `__init__`，只有 `__new__` 返回了，当前类 `cls` 的实例，当前类的 `__init__` 才会进入。

【例子】

```
class A(object):
    def __init__(self, value):
        print("into A __init__")
        self.value = value

    def __new__(cls, *args, **kwargs):
        print("into A __new__")
        print(cls)
        return object.__new__(cls)


class B(A):
    def __init__(self, value):
        print("into B __init__")
        self.value = value

    def __new__(cls, *args, **kwargs):
        print("into B __new__")
        print(cls)
        return super().__new__(cls, *args, **kwargs)


b = B(10)

# 结果:
# into B __new__
# <class '__main__.B'>
# into A __new__
# <class '__main__.B'>
# into B __init__

class A(object):
    def __init__(self, value):
        print("into A __init__")
        self.value = value
```

```
36
37      def __new__(cls, *args, **kwargs):
38          print("into A __new__")
39          print(cls)
40          return object.__new__(cls)
41
42
43  class B(A):
44      def __init__(self, value):
45          print("into B __init__")
46          self.value = value
47
48      def __new__(cls, *args, **kwargs):
49          print("into B __new__")
50          print(cls)
51          return super().__new__(A, *args, **kwargs)  # 改动了cls变为A
52
53
54  b = B(10)
55
56  # 结果:
57  # into B __new__
58  # <class '__main__.B'>
59  # into A __new__
60  # <class '__main__.A'>
```

1. 若`__new__`没有正确返回当前类`cls`的实例，那`__init__`是不会被调用的，即使是父类的实例也不行，将没有`__init__`被调用。
2. 可利用`__new__`实现单例模式。

【例子】

```
1   class Earth:
2       pass
3
4
5   a = Earth()
6   print(id(a))  # 260728291456
7   b = Earth()
8   print(id(b))  # 260728291624
9
10  class Earth:
11      __instance = None  # 定义一个类属性做判断
12
13      def __new__(cls):
14          if cls.__instance is None:
```

```
            cls.__instance = object.__new__(cls)
            return cls.__instance
        else:
            return cls.__instance


a = Earth()
print(id(a))  # 512320401648
b = Earth()
print(id(b))  # 512320401648
```

1. `__new__` 方法主要是当你继承一些不可变的 class 时（比如 `int, str, tuple`）， 提供给你一个自定义这些类的实例化过程的途径。

【例子】

```
class CapStr(str):
    def __new__(cls, string):
        string = string.upper()
        return str.__new__(cls, string)


a = CapStr("i love lsgogroup")
print(a)  # I LOVE LSGOGROUP
```

`__del__(self)`

析构器，当一个对象将要被系统回收之时调用的方法。

Python 采用自动引用计数（ARC）方式来回收对象所占用的空间，当程序中有一个变量引用该 Python 对象时，Python 会自动保证该对象引用计数为 1；当程序中有两个变量引用该 Python 对象时，Python 会自动保证该对象引用计数为 2，依此类推，如果一个对象的引用计数变成了 0，则说明程序中不再有变量引用该对象，表明程序不再需要该对象，因此 Python 就会回收该对象。

大部分时候，Python 的 ARC 都能准确、高效地回收系统中的每个对象。但如果系统中出现循环引用的情况，比如对象 a 持有一个实例变量引用对象 b，而对象 b 又持有一个实例变量引用对象 a，此时两个对象的引用计数都是 1，而实际上程序已经不再有变量引用它们，系统应该回收它们，此时 Python 的垃圾回收器就可能没那么快，要等专门的循环垃圾回收器（Cyclic Garbage Collector）来检测并回收这种引用循环。

【例子】

```
class C(object):
    def __init__(self):
        print('into C __init__')

    def __del__(self):
        print('into C __del__')
```

```
7
8
9   c1 = C()
10  # into C __init__
11  c2 = c1
12  c3 = c2
13  del c3
14  del c2
15  del c1
16  # into C __del__
```

`__str__` 和 `__repr__`

`__str__(self)`:

1. 当你打印一个对象的时候，触发`__str__`
2. 当你使用`%s`格式化的时候，触发`__str__`
3. `str`强转数据类型的时候，触发`__str__`

`__repr__(self):`

1. `repr`是`str`的备胎
2. 有`__str__`的时候执行`__str__`,没有实现`__str__`的时候，执行`__repr__`
3. `repr(obj)`内置函数对应的结果是`__repr__`的返回值
4. 当你使用`%r`格式化的时候 触发`__repr__`

【例子】

```
1   class Cat:
2       """定义一个猫类"""
3
4       def __init__(self, new_name, new_age):
5           """在创建完对象之后 会自动调用，它完成对象的初始化的功能"""
6           self.name = new_name
7           self.age = new_age
8
9       def __str__(self):
10          """返回一个对象的描述信息"""
11          return "名字是:%s ，年龄是:%d" % (self.name, self.age)
12
13      def __repr__(self):
14          """返回一个对象的描述信息"""
15          return "Cat:(%s,%d)" % (self.name, self.age)
16
17      def eat(self):
18          print("%s在吃鱼...." % self.name)
19
```

```
    def drink(self):
        print("%s在喝可乐..." % self.name)

    def introduce(self):
        print("名字是:%s, 年龄是:%d" % (self.name, self.age))


# 创建了一个对象
tom = Cat("汤姆", 30)
print(tom)  # 名字是:汤姆 , 年龄是:30
print(str(tom)) # 名字是:汤姆 , 年龄是:30
print(repr(tom))  # Cat:(汤姆,30)
tom.eat()  # 汤姆在吃鱼....
tom.introduce()  # 名字是:汤姆, 年龄是:30
```

`__str__(self)` 的返回结果可读性强。也就是说， `__str__` 的意义是得到便于人们阅读的信息，就像下面的 '2019-10-11' 一样。

`__repr__(self)` 的返回结果应更准确。怎么说， `__repr__` 存在的目的在于调试，便于开发者使用。

【例子】

```
import datetime

today = datetime.date.today()
print(str(today))  # 2019-10-11
print(repr(today))  # datetime.date(2019, 10, 11)
print('%s' %today)  # 2019-10-11
print('%r' %today)  # datetime.date(2019, 10, 11)
```

## 15.2 算术运算符

类型工厂函数，指的是不通过类而是通过函数来创建对象。

【例子】

```
class C:
    pass


print(type(len))  # <class 'builtin_function_or_method'>
print(type(dir))  # <class 'builtin_function_or_method'>
print(type(int))  # <class 'type'>
```

```
print(type(list))  # <class 'type'>
print(type(tuple))  # <class 'type'>
print(type(C))  # <class 'type'>
print(int('123'))  # 123

# 这个例子中list工厂函数把一个元祖对象加工成了一个列表对象。
print(list((1, 2, 3)))  # [1, 2, 3]
```

1. `__add__(self, other)` 定义加法的行为：`+`
2. `__sub__(self, other)` 定义减法的行为：`-`

```
class MyClass:

    def __init__(self, height, weight):
        self.height = height
        self.weight = weight

    # 两个对象的长相加，宽不变.返回一个新的类
    def __add__(self, others):
        return MyClass(self.height + others.height, self.weight + others.weight)

    # 两个对象的宽相减，长不变.返回一个新的类
    def __sub__(self, others):
        return MyClass(self.height - others.height, self.weight - others.weight)

    # 说一下自己的参数
    def intro(self):
        print("高为", self.height, " 重为", self.weight)


def main():
    a = MyClass(height=10, weight=5)
    a.intro()

    b = MyClass(height=20, weight=10)
    b.intro()

    c = b - a
    c.intro()

    d = a + b
    d.intro()


if __name__ == '__main__':
```

```
    main()

# 高为 10  重为 5
# 高为 20  重为 10
# 高为 10  重为 5
# 高为 30  重为 15
```

1. `__mul__(self, other)` 定义乘法的行为：`*`
2. `__truediv__(self, other)` 定义真除法的行为：`/`
3. `__floordiv__(self, other)` 定义整数除法的行为：`//`
4. `__mod__(self, other)` 定义取模算法的行为：`%`
5. `__divmod__(self, other)` 定义当被 `divmod()` 调用时的行为
6. `divmod(a, b)` 把除数和余数运算结果结合起来，返回一个包含商和余数的元组 `(a // b, a % b)`。

【例子】

```
print(divmod(7, 2))  # (3, 1)
print(divmod(8, 2))  # (4, 0)
```

1. `__pow__(self, other[, module])` 定义当被 `power()` 调用或 `**` 运算时的行为
2. `__lshift__(self, other)` 定义按位左移位的行为：`<<`
3. `__rshift__(self, other)` 定义按位右移位的行为：`>>`
4. `__and__(self, other)` 定义按位与操作的行为：`&`
5. `__xor__(self, other)` 定义按位异或操作的行为：`^`
6. `__or__(self, other)` 定义按位或操作的行为：`|`

## 15.3 反算术运算符

反运算魔方方法，与算术运算符保持一一对应，不同之处就是反运算的魔法方法多了一个“r”。当文件左操作不支持相应的操作时被调用。

1. `__radd__(self, other)` 定义加法的行为：`+`
2. `__rsub__(self, other)` 定义减法的行为：`-`
3. `__rmul__(self, other)` 定义乘法的行为：`*`
4. `__rtruediv__(self, other)` 定义真除法的行为：`/`
5. `__rfloordiv__(self, other)` 定义整数除法的行为：`//`
6. `__rmod__(self, other)` 定义取模算法的行为：`%`
7. `__rdivmod__(self, other)` 定义当被 divmod() 调用时的行为
8. `__rpow__(self, other[, module])` 定义当被 power() 调用或 `**` 运算时的行为
9. `__rlshift__(self, other)` 定义按位左移位的行为：`<<`

10. `__rrshift__(self, other)` 定义按位右移位的行为：`>>`
11. `__rand__(self, other)` 定义按位与操作的行为：`&`
12. `__rxor__(self, other)` 定义按位异或操作的行为：`^`
13. `__ror__(self, other)` 定义按位或操作的行为：`|`

`a + b`

这里加数是`a`，被加数是`b`，因此是`a`主动，反运算就是如果`a`对象的`__add__()`方法没有实现或者不支持相应的操作，那么 Python 就会调用`b`的`__radd__()`方法。

【例子】

```
class Nint(int):
    def __radd__(self, other):
        return int.__sub__(other, self) # 注意 self 在后面


a = Nint(5)
b = Nint(3)
print(a + b)  # 8
print(1 + b)  # -2
```

## 15.4 增量赋值运算符

1. `__iadd__(self, other)` 定义赋值加法的行为：`+=`
2. `__isub__(self, other)` 定义赋值减法的行为：`-=`
3. `__imul__(self, other)` 定义赋值乘法的行为：`*=`
4. `__itruediv__(self, other)` 定义赋值真除法的行为：`/=`
5. `__ifloordiv__(self, other)` 定义赋值整数除法的行为：`//=`
6. `__imod__(self, other)` 定义赋值取模算法的行为：`%=`
7. `__ipow__(self, other[, modulo])` 定义赋值幂运算的行为：`**=`
8. `__ilshift__(self, other)` 定义赋值按位左移位的行为：`<<=`
9. `__irshift__(self, other)` 定义赋值按位右移位的行为：`>>=`
10. `__iand__(self, other)` 定义赋值按位与操作的行为：`&=`
11. `__ixor__(self, other)` 定义赋值按位异或操作的行为：`^=`
12. `__ior__(self, other)` 定义赋值按位或操作的行为：`|=`

## 15.5 一元运算符

1. `__neg__(self)` 定义正号的行为：`+x`
2. `__pos__(self)` 定义负号的行为：`-x`
3. `__abs__(self)` 定义当被 `abs()` 调用时的行为
4. `__invert__(self)` 定义按位求反的行为：`~x`

## 15.6 属性访问

`__getattr__`，`__getattribute__`，`__setattr__` 和 `__delattr__`

`__getattr__(self, name)`: 定义当用户试图获取一个不存在的属性时的行为。

`__getattribute__(self, name)`：定义当该类的属性被访问时的行为（先调用该方法，查看是否存在该属性，若不存在，接着去调用 `__getattr__`）。

`__setattr__(self, name, value)`：定义当一个属性被设置时的行为。

`__delattr__(self, name)`：定义当一个属性被删除时的行为。

【例子】

```
class C:
    def __getattribute__(self, item):
        print('__getattribute__')
        return super().__getattribute__(item)

    def __getattr__(self, item):
        print('__getattr__')

    def __setattr__(self, key, value):
        print('__setattr__')
        super().__setattr__(key, value)

    def __delattr__(self, item):
        print('__delattr__')
        super().__delattr__(item)


c = C()
c.x
```

```
# __getattribute__
# __getattr__

c.x = 1
# __setattr__

del c.x
# __delattr__
```

扩展参考：

1. 技术图文：Python魔法方法之属性访问详解

## 15.7 描述符

描述符就是将某种特殊类型的类的实例指派给另一个类的属性。

1. `__get__(self, instance, owner)` 用于访问属性，它返回属性的值。
2. `__set__(self, instance, value)` 将在属性分配操作中调用，不返回任何内容。
3. `__del__(self, instance)` 控制删除操作，不返回任何内容。

【例子】

```
class MyDecriptor:
    def __get__(self, instance, owner):
        print('__get__', self, instance, owner)

    def __set__(self, instance, value):
        print('__set__', self, instance, value)

    def __delete__(self, instance):
        print('__delete__', self, instance)


class Test:
    x = MyDecriptor()


t = Test()
t.x
# __get__ <__main__.MyDecriptor object at 0x000000CEAAEB6B00> <__main__.Test object at
0x000000CEABDC0898> <class '__main__.Test'>

```

```
t.x = 'x-man'
# __set__ <__main__.MyDecriptor object at 0x00000023687C6B00> <__main__.Test object at 0x00000023696B0940> x-man

del t.x
# __delete__ <__main__.MyDecriptor object at 0x000000EC9B160A90> <__main__.Test object at 0x000000EC9B160B38>
```

扩展参考：

1. 技术图文：什么是Python的描述符？

## 15.8 定制序列

协议（Protocols）与其它编程语言中的接口很相似，它规定你哪些方法必须要定义。然而，在 Python 中的协议就显得不那么正式。事实上，在 Python 中，协议更像是一种指南。

**容器类型的协议**

1. 如果说你希望定制的容器是不可变的话，你只需要定义`__len__()`和`__getitem__()`方法。
2. 如果你希望定制的容器是可变的话，除了`__len__()`和`__getitem__()`方法，你还需要定义`__setitem__()`和`__delitem__()`两个方法。

【例子】编写一个不可改变的自定义列表，要求记录列表中每个元素被访问的次数。

```
class CountList:
    def __init__(self, *args):
        self.values = [x for x in args]
        self.count = {}.fromkeys(range(len(self.values)), 0)

    def __len__(self):
        return len(self.values)

    def __getitem__(self, item):
        self.count[item] += 1
        return self.values[item]


c1 = CountList(1, 3, 5, 7, 9)
c2 = CountList(2, 4, 6, 8, 10)
print(c1[1])  # 3
print(c2[2])  # 6
print(c1[1] + c2[1])  # 7
```

```
19
20  print(c1.count)
21  # {0: 0, 1: 2, 2: 0, 3: 0, 4: 0}
22
23  print(c2.count)
24  # {0: 0, 1: 1, 2: 1, 3: 0, 4: 0}
```

1. `__len__(self)` 定义当被 `len()` 调用时的行为（返回容器中元素的个数）。
2. `__getitem__(self, key)` 定义获取容器中元素的行为，相当于 `self[key]`。
3. `__setitem__(self, key, value)` 定义设置容器中指定元素的行为，相当于 `self[key] = value`。
4. `__delitem__(self, key)` 定义删除容器中指定元素的行为，相当于 `del self[key]`。

【例子】编写一个可改变的自定义列表，要求记录列表中每个元素被访问的次数。

```
1   class CountList:
2       def __init__(self, *args):
3           self.values = [x for x in args]
4           self.count = {}.fromkeys(range(len(self.values)), 0)
5
6       def __len__(self):
7           return len(self.values)
8
9       def __getitem__(self, item):
10          self.count[item] += 1
11          return self.values[item]
12
13      def __setitem__(self, key, value):
14          self.values[key] = value
15
16      def __delitem__(self, key):
17          del self.values[key]
18          for i in range(0, len(self.values)):
19              if i >= key:
20                  self.count[i] = self.count[i + 1]
21          self.count.pop(len(self.values))
22
23
24  c1 = CountList(1, 3, 5, 7, 9)
25  c2 = CountList(2, 4, 6, 8, 10)
26  print(c1[1])  # 3
27  print(c2[2])  # 6
28  c2[2] = 12
29  print(c1[1] + c2[2])  # 15
30  print(c1.count)
31  # {0: 0, 1: 2, 2: 0, 3: 0, 4: 0}
```

```
print(c2.count)
# {0: 0, 1: 0, 2: 2, 3: 0, 4: 0}
del c1[1]
print(c1.count)
# {0: 0, 1: 0, 2: 0, 3: 0}
```

## 15.9 迭代器

1. 迭代是 Python 最强大的功能之一，是访问集合元素的一种方式。
2. 迭代器是一个可以记住遍历的位置的对象。
3. 迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。
4. 迭代器只能往前不会后退。
5. 字符串，列表或元组对象都可用于创建迭代器：

【例子】

```
string = 'lsgogroup'
for c in string:
    print(c)

'''
l
s
g
o
g
r
o
u
p
'''

for c in iter(string):
    print(c)
```

【例子】

```python
links = {'B': '百度', 'A': '阿里', 'T': '腾讯'}
for each in links:
    print('%s -> %s' % (each, links[each]))

'''
B -> 百度
A -> 阿里
T -> 腾讯
'''

for each in iter(links):
    print('%s -> %s' % (each, links[each]))
```

1. 迭代器有两个基本的方法：`iter()` 和 `next()`。
2. `iter(object)` 函数用来生成迭代器。
3. `next(iterator[, default])` 返回迭代器的下一个项目。
4. `iterator` -- 可迭代对象
5. `default` -- 可选，用于设置在没有下一个元素时返回该默认值，如果不设置，又没有下一个元素则会触发 `StopIteration` 异常。

【例子】

```python
links = {'B': '百度', 'A': '阿里', 'T': '腾讯'}
it = iter(links)
print(next(it))  # B
print(next(it))  # A
print(next(it))  # T
print(next(it))  # StopIteration


it = iter(links)
while True:
    try:
        each = next(it)
    except StopIteration:
        break
    print(each)

# B
# A
# T
```

把一个类作为一个迭代器使用需要在类中实现两个魔法方法 `__iter__()` 与 `__next__()` 。

1. `__iter__(self)` 定义当迭代容器中的元素的行为，返回一个特殊的迭代器对象， 这个迭代器对象实现了 `__next__()` 方法并通过 `StopIteration` 异常标识迭代的完成。

2. `__next__()` 返回下一个迭代器对象。
3. `StopIteration` 异常用于标识迭代的完成，防止出现无限循环的情况，在 `__next__()` 方法中我们可以设置在完成指定循环次数后触发 `StopIteration` 异常来结束迭代。

【例子】

```
class Fibs:
    def __init__(self, n=10):
        self.a = 0
        self.b = 1
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        if self.a > self.n:
            raise StopIteration
        return self.a


fibs = Fibs(100)
for each in fibs:
    print(each, end=' ')

# 1 1 2 3 5 8 13 21 34 55 89
```

## 15.10 生成器

1. 在 Python 中，使用了 `yield` 的函数被称为生成器（generator）。
2. 跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。
3. 在调用生成器运行的过程中，每次遇到 `yield` 时函数会暂停并保存当前所有的运行信息，返回 `yield` 的值, 并在下一次执行 `next()` 方法时从当前位置继续运行。
4. 调用一个生成器函数，返回的是一个迭代器对象。

【例子】

```
def myGen():
    print('生成器执行! ')
    yield 1
```

```
    yield 2


myG = myGen()
print(next(myG))
# 生成器执行!
# 1

print(next(myG))  # 2
print(next(myG))  # StopIteration

myG = myGen()
for each in myG:
    print(each)

'''
生成器执行!
1
2
'''
```

【例子】用生成器实现斐波那契数列。

```
def libs(n):
    a = 0
    b = 1
    while True:
        a, b = b, a + b
        if a > n:
            return
        yield a


for each in libs(100):
    print(each, end=' ')

# 1 1 2 3 5 8 13 21 34 55 89
```

---

**参考文献：**


1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443
3. http://c.biancheng.net/view/2371.html

4. https://www.cnblogs.com/seablog/p/7173107.html
5. https://www.cnblogs.com/Jimmy1988/p/6804095.html
6. https://blog.csdn.net/johnsonguo/article/details/585193

---

**练习题：**

1. 上面提到了许多魔法方法，如`__new__`,`__init__`,`__str__`,`__rstr__`,`__getitem__`,`__setitem__`等等，请总结它们各自的使用方法。
2. 利用python做一个简单的定时器类

   要求:

   a. 定制一个计时器的类。

   b. `start`和`stop`方法代表启动计时和停止计时。

   c. 假设计时器对象`t1`，`print(t1)`和直接调用`t1`均显示结果。

   d. 当计时器未启动或已经停止计时时，调用`stop`方法会给予温馨的提示。

   e. 两个计时器对象可以进行相加：`t1+t2`。

   f. 只能使用提供的有限资源完成。

# 16 模块

在前面我们脚本是用 Python 解释器来编程，如果你从 Python 解释器退出再进入，那么你定义的所有的方法和变量就都消失了。

为此 Python 提供了一个办法，把这些定义存放在文件中，为一些脚本或者交互式的解释器实例使用，这个文件被称为模块（Module）。

模块是一个包含所有你定义的函数和变量的文件，其后缀名是`.py`。模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用 Python 标准库的方法。

## 16.1 什么是模块

1. 容器 -> 数据的封装
2. 函数 -> 语句的封装
3. 类 -> 方法和属性的封装
4. 模块 -> 程序文件

【例子】创建一个 hello.py 文件

```
# hello.py
def hi():
    print('Hi everyone, I love lsgogroup!')
```

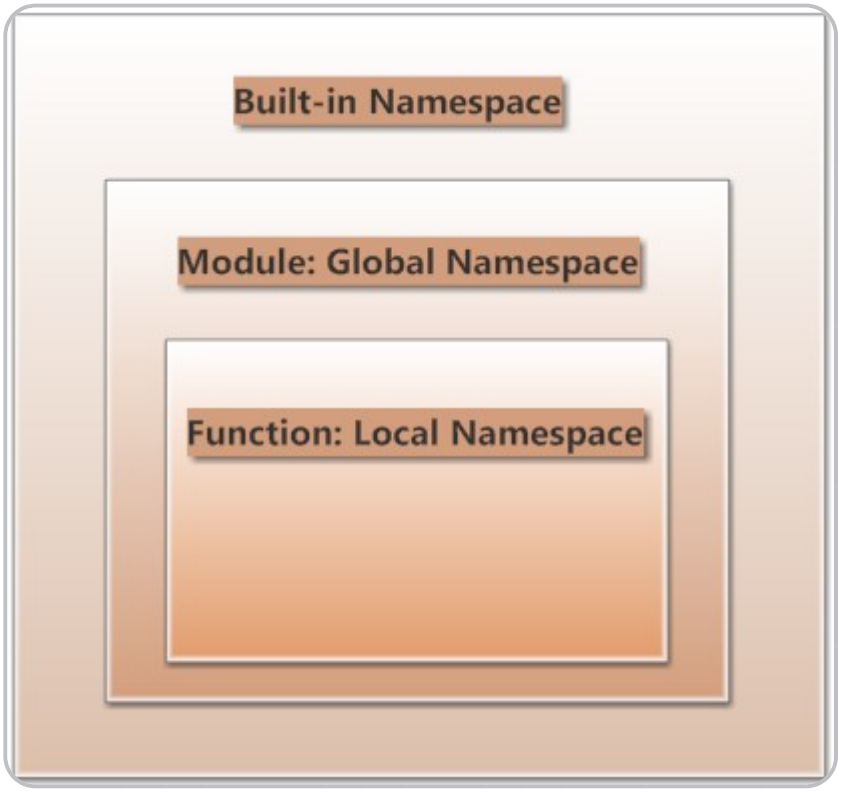
## 16.2 命名空间

命名空间因为对象的不同，也有所区别，可以分为如下几种：

1. 内置命名空间（Built-in Namespaces）：Python 运行起来，它们就存在了。内置函数的命名空间都属于内置命名空间，所以，我们可以在任何程序中直接运行它们，比如`id()`,不需要做什么操作，拿过来就直接使用了。

2. 全局命名空间（Module：Global Namespaces）：每个模块创建它自己所拥有的全局命名空间，不同模块的全局命名空间彼此独立，不同模块中相同名称的命名空间，也会因为模块的不同而不相互干扰。
3. 本地命名空间（Function & Class：Local Namespaces）：模块中有函数或者类，每个函数或者类所定义的命名空间就是本地命名空间。如果函数返回了结果或者抛出异常，则本地命名空间也结束了。

上述三种命名空间的关系




程序在查询上述三种命名空间的时候，就按照从里到外的顺序，即：Local Namespaces --> Global Namesspaces --> Built-in Namesspaces。

【例子】

```
import hello

hello.hi()  # Hi everyone, I love lsgogroup!
hi()  # NameError: name 'hi' is not defined
```

## 16.3 导入模块

【例子】创建一个模块 TemperatureConversion.py

```python
# TemperatureConversion.py
def c2f(cel):
    fah = cel * 1.8 + 32
    return fah


def f2c(fah):
    cel = (fah - 32) / 1.8
    return cel

```

1. 第一种：import 模块名

【例子】

```python
import TemperatureConversion

print('32摄氏度 = %.2f华氏度' % TemperatureConversion.c2f(32))
print('99华氏度 = %.2f摄氏度' % TemperatureConversion.f2c(99))

# 32摄氏度 = 89.60华氏度
# 99华氏度 = 37.22摄氏度
```

1. 第二种：from 模块名 import 函数名

【例子】

```python
from TemperatureConversion import c2f, f2c

print('32摄氏度 = %.2f华氏度' % c2f(32))
print('99华氏度 = %.2f摄氏度' % f2c(99))

# 32摄氏度 = 89.60华氏度
# 99华氏度 = 37.22摄氏度
```

下面的方式不推荐

【例子】

```python
from TemperatureConversion import *

print('32摄氏度 = %.2f华氏度' % c2f(32))
print('99华氏度 = %.2f摄氏度' % f2c(99))

# 32摄氏度 = 89.60华氏度
# 99华氏度 = 37.22摄氏度
```

1. 第三种：import 模块名 as 新名字

【例子】

```
import TemperatureConversion as tc

print('32摄氏度 = %.2f华氏度' % tc.c2f(32))
print('99华氏度 = %.2f摄氏度' % tc.f2c(99))

# 32摄氏度 = 89.60华氏度
# 99华氏度 = 37.22摄氏度
```

## 16.4 if __name__ == '__main__'

对于很多编程语言来说，程序都必须要有一个入口，而 Python 则不同，它属于脚本语言，不像编译型语言那样先将程序编译成二进制再运行，而是动态的逐行解释运行。也就是从脚本第一行开始运行，没有统一的入口。

假设我们有一个 const.py 文件，内容如下：

```
PI = 3.14


def main():
    print("PI:", PI)


main()

# PI: 3.14
```

现在，我们写一个用于计算圆面积的 area.py 文件，area.py 文件需要用到 const.py 文件中的 `PI` 变量。从 const.py 中，我们把 `PI` 变量导入 area.py：

```
from const import PI


def calc_round_area(radius):
    return PI * (radius ** 2)


def main():
    print("round area: ", calc_round_area(2))


```

```
main()

'''
PI: 3.14
round area:  12.56
'''
```

我们看到 const.py 中的 main 函数也被运行了，实际上我们不希望它被运行，因为 const.py 提供的 main 函数只是为了测试常量定义。这时`if __name__ == '__main__'`派上了用场，我们把 const.py 改一下，添加`if __name__ == "__main__"`：

```
PI = 3.14

def main():
    print("PI:", PI)

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

运行 const.py，输出如下：

```
PI: 3.14
```

运行 area.py，输出如下：

```
round area:  12.56
```

`__name__`：是内置变量，可用于表示当前模块的名字。

```
import const

print(__name__)
# __main__

print(const.__name__)
# const
```

由此我们可知：如果一个 .py 文件（模块）被直接运行时，其`__name__`值为`__main__`，即模块名为`__main__`。

所以，`if __name__ == '__main__'`的意思是：当 .py 文件被直接运行时，`if __name__ == '__main__'`之下的代码块将被运行；当 .py 文件以模块形式被导入时，`if __name__ == '__main__'`之下的代码块不被运行。

## 16.5 搜索路径

当解释器遇到 import 语句，如果模块在当前的搜索路径就会被导入。

【例子】

```
import sys

print(sys.path)

# ['C:\\ProgramData\\Anaconda3\\DLLs', 'C:\\ProgramData\\Anaconda3\\lib',
'C:\\ProgramData\\Anaconda3', 'C:\\ProgramData\\Anaconda3\\lib\\site-packages',...]
```

我们使用 import 语句的时候，Python 解释器是怎样找到对应的文件的呢？

这就涉及到 Python 的搜索路径，搜索路径是由一系列目录名组成的，Python 解释器就依次从这些目录中去寻找所引入的模块。

这看起来很像环境变量，事实上，也可以通过定义环境变量的方式来确定搜索路径。

搜索路径是在 Python 编译或安装的时候确定的，安装新的库应该也会修改。搜索路径被存储在 `sys` 模块中的 `path` 变量中。

## 16.6 包（package）

包是一种管理 Python 模块命名空间的形式，采用"点模块名称"。

创建包分为三个步骤：

1. 创建一个文件夹，用于存放相关的模块，文件夹的名字即包的名字。
2. 在文件夹中创建一个 `__init__.py` 的模块文件，内容可以为空。
3. 将相关的模块放入文件夹中。

不妨假设你想设计一套统一处理声音文件和数据的模块（或者称之为一个"包"）。

现存很多种不同的音频文件格式（基本上都是通过后缀名区分的，例如： .wav，.aiff，.au），所以你需要有一组不断增加的模块，用来在不同的格式之间转换。

并且针对这些音频数据，还有很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所以你还需要一组怎么也写不完的模块来处理这些操作。

这里给出了一种可能的包结构（在分层的文件系统中）：

```
sound/                          顶层包
      __init__.py               初始化 sound 包
      formats/                  文件格式转换子包
```

```
        __init__.py
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ...
    effects/                声音效果子包
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    filters/                filters 子包
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

在导入一个包的时候，Python 会根据 `sys.path` 中的目录来寻找这个包中包含的子目录。

目录只有包含一个叫做 `__init__.py` 的文件才会被认作是一个包，最简单的情况，放一个空的 `__init__.py` 就可以了。

```
import sound.effects.echo
```

这将会导入子模块 `sound.effects.echo`。 他必须使用全名去访问:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

还有一种导入子模块的方法是：

```
from sound.effects import echo
```

这同样会导入子模块: echo，并且他不需要那些冗长的前缀，所以他可以这样使用：

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

还有一种变化就是直接导入一个函数或者变量：

```
from sound.effects.echo import echofilter
```

同样的，这种方法会导入子模块: echo，并且可以直接使用他的 echofilter() 函数：

```
echofilter(input, output, delay=0.7, atten=4)
```

注意当使用 `from package import item` 这种形式的时候，对应的 `item` 既可以是包里面的子模块（子包），或者包里面定义的其他名称，比如函数，类或者变量。

设想一下，如果我们使用 `from sound.effects import *` 会发生什么？

Python 会进入文件系统，找到这个包里面所有的子模块，一个一个的把它们都导入进来。

导入语句遵循如下规则：如果包定义文件 `__init__.py` 存在一个叫做 `__all__` 的列表变量，那么在使用 `from package import *` 的时候就把这个列表中的所有名字作为包内容导入。

这里有一个例子，在 `sounds/effects/__init__.py` 中包含如下代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这表示当你使用 `from sound.effects import *` 这种用法时，你只会导入包里面这三个子模块。

如果 `__all__` 真的没有定义，那么使用 `from sound.effects import *` 这种语法的时候，就不会导入包 `sound.effects` 里的任何子模块。他只是把包 `sound.effects` 和它里面定义的所有内容导入进来（可能运行 `__init__.py` 里定义的初始化代码）。

这会把 `__init__.py` 里面定义的所有名字导入进来。并且他不会破坏掉我们在这句话之前导入的所有明确指定的模块。

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

这个例子中，在执行 `from...import` 前，包 `sound.effects` 中的 `echo` 和 `surround` 模块都被导入到当前的命名空间中了。

通常我们并不主张使用 `*` 这种方法来导入模块，因为这种方法经常会导致代码的可读性降低。

---

参考文献：


1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443
3. https://blog.csdn.net/u010820857/article/details/85330778


---

**练习题：**

1. 怎么查出通过 from xx import xx导入的可以直接调用的方法？
2. 了解Collection模块，编写程序以查询给定列表中最常见的元素。

   题目说明:

   输入: language = ['PHP', 'PHP', 'Python', 'PHP', 'Python', 'JS', 'Python', 'Python','PHP', 'Python']

   输出: Python

```
"""

Input file
language = ['PHP', 'PHP', 'Python', 'PHP', 'Python', 'JS', 'Python', 'Python','PHP', 'Python']

Output file
Python

"""
def most_element(language):
    """ Return a list of lines after inserting a word in a specific line. """

    # your code here
```

# 17 datetime模块

datetime 是 Python 中处理日期的标准模块，它提供了 4 种对日期和时间进行处理的类：**datetime**、**date**、**time** 和 **timedelta**。

## 17.1 datetime类

```
class datetime(date):
    def __init__(self, year, month, day, hour, minute, second, microsecond, tzinfo)
        pass
    def now(cls, tz=None):
        pass
    def timestamp(self):
        pass
    def fromtimestamp(cls, t, tz=None):
        pass
    def date(self):
        pass
    def time(self):
        pass
    def year(self):
        pass
    def month(self):
        pass
    def day(self):
        pass
    def hour(self):
        pass
    def minute(self):
        pass
    def second(self):
        pass
    def isoweekday(self):
        pass
```

```
    def strftime(self, fmt):
        pass
    def combine(cls, date, time, tzinfo=True):
        pass
```

1. `datetime.now(tz=None)` 获取当前的日期时间，输出顺序为：年、月、日、时、分、秒、微秒。
2. `datetime.timestamp()` 获取以 1970年1月1日为起点记录的秒数。
3. `datetime.fromtimestamp(tz=None)` 使用 unixtimestamp 创建一个 datetime。

【例子】如何创建一个 datetime 对象？

```
import datetime

dt = datetime.datetime(year=2020, month=6, day=25, hour=11, minute=23, second=59)
print(dt)  # 2020-06-25 11:23:59
print(dt.timestamp())  # 1593055439.0

dt = datetime.datetime.fromtimestamp(1593055439.0)
print(dt)  # 2020-06-25 11:23:59
print(type(dt)) # <class 'datetime.datetime'>

dt = datetime.datetime.now()
print(dt)  # 2020-06-25 11:11:03.877853
print(type(dt))  # <class 'datetime.datetime'>
```

1. `datetime.strftime(fmt)` 格式化 datetime 对象。

| 符号 | 说明 |
| --- | --- |
| %a | 本地简化星期名称（如星期一，返回 Mon） |
| %A | 本地完整星期名称（如星期一，返回 Monday） |
| %b | 本地简化的月份名称（如一月，返回 Jan） |
| %B | 本地完整的月份名称（如一月，返回 January） |
| %c | 本地相应的日期表示和时间表示 |
| %d | 月内中的一天（0-31） |
| %H | 24小时制小时数（0-23） |
| %I | 12小时制小时数（01-12） |
| %j | 年内的一天（001-366） |
| %m | 月份（01-12） |
| %M | 分钟数（00-59） |
| %p | 本地A.M.或P.M.的等价符 |

| | |
|---|---|
| %S | 秒（00-59） |
| %U | 一年中的星期数（00-53）星期天为星期的开始 |
| %w | 星期（0-6），星期天为星期的开始 |
| %W | 一年中的星期数（00-53）星期一为星期的开始 |
| %x | 本地相应的日期表示 |
| %X | 本地相应的时间表示 |
| %y | 两位数的年份表示（00-99） |
| %Y | 四位数的年份表示（0000-9999） |
| %Z | 当前时区的名称（如果是本地时间，返回空字符串） |
| %% | %号本身 |

【例子】如何将 datetime 对象转换为任何格式的日期？

```
import datetime

dt = datetime.datetime(year=2020, month=6, day=25, hour=11, minute=51, second=49)
s = dt.strftime("'%Y/%m/%d %H:%M:%S")
print(s)  # '2020/06/25 11:51:49

s = dt.strftime('%d %B, %Y, %A')
print(s)  # 25 June, 2020, Thursday
```

【练习】如何将给定日期转换为 "mmm-dd, YYYY" 的格式？

```
# 输入
d1 = datetime.date('2010-09-28')

# 输出
'Sep-28,2010'
```

【参考答案】

```
import datetime

d1 = datetime.date(2010, 9, 28)
print(d1.strftime('%b-%d,%Y'))
# Sep-28,2010
```

1. `datetime.date()` Return the date part.
2. `datetime.time()` Return the time part, with tzinfo None.
3. `datetime.year` 年
4. `datetime.month` 月

5. `datetime.day` 日
6. `datetime.hour` 小时
7. `datetime.minute` 分钟
8. `datetime.second` 秒
9. `datetime.isoweekday` 星期几

【例子】datetime 对象包含很多与日期时间相关的实用功能。

```
import datetime

dt = datetime.datetime(year=2020, month=6, day=25, hour=11, minute=51, second=49)
print(dt.date())  # 2020-06-25
print(type(dt.date()))  # <class 'datetime.date'>
print(dt.time())  # 11:51:49
print(type(dt.time()))  # <class 'datetime.time'>
print(dt.year)  # 2020
print(dt.month)  # 6
print(dt.day)  # 25
print(dt.hour)  # 11
print(dt.minute)  # 51
print(dt.second)  # 49
print(dt.isoweekday())  # 4
```

在处理含有字符串日期的数据集或表格时，我们需要一种自动解析字符串的方法，无论它是什么格式的，都可以将其转化为 datetime 对象。这时，就要使用到 dateutil 中的 parser 模块。

1. `parser.parse(timestr, parserinfo=None, **kwargs)`

【例子】如何在 python 中将字符串解析为 datetime对象？

```
from dateutil import parser

s = '2020-06-25'
dt = parser.parse(s)
print(dt)  # 2020-06-25 00:00:00
print(type(dt))  # <class 'datetime.datetime'>

s = 'March 31, 2010, 10:51pm'
dt = parser.parse(s)
print(dt)  # 2010-03-31 22:51:00
print(type(dt))  # <class 'datetime.datetime'>
```

【练习】如何将字符串日期解析为 datetime 对象？

```
# 输入
s1 = "2010 Jan 1"
s2 = '31-1-2000'
s3 = 'October10, 1996, 10:40pm'

# 输出
2010-01-01 00:00:00
2000-01-31 00:00:00
2019-10-10 22:40:00
```

【参考答案】

```
from dateutil import parser

s1 = "2010 Jan 1"
s2 = '31-1-2000'
s3 = 'October10, 1996, 10:40pm'

dt1 = parser.parse(s1)
dt2 = parser.parse(s2)
dt3 = parser.parse(s3)

print(dt1)  # 2010-01-01 00:00:00
print(dt2)  # 2000-01-31 00:00:00
print(dt3)  # 1996-10-10 22:40:00
```

【练习】计算以下列表中连续的天数。

```
# 输入
['Oct, 2, 1869', 'Oct, 10, 1869', 'Oct, 15, 1869', 'Oct, 20, 1869','Oct, 23, 1869']

# 输出
[8, 5, 5, 3]
```

【参考答案】

```
import numpy as np
from dateutil import parser

dateString = ['Oct, 2, 1869', 'Oct, 10, 1869', 'Oct, 15, 1869', 'Oct, 20, 1869', 'Oct, 23, 1869']
dates = [parser.parse(i) for i in dateString]
td = np.diff(dates)
print(td)
# [datetime.timedelta(days=8) datetime.timedelta(days=5)
#  datetime.timedelta(days=5) datetime.timedelta(days=3)]
d = [i.days for i in td]
print(d)  # [8, 5, 5, 3]
```

## 17.2 date类

```
class date:
    def __init__(self, year, month, day):
        pass
    def today(cls):
        pass
```

1. `date.today()` 获取当前日期信息。

【例子】如何在 Python 中获取当前日期和时间？

```
import datetime

d = datetime.date(2020, 6, 25)
print(d)  # 2020-06-25
print(type(d))  # <class 'datetime.date'>

d = datetime.date.today()
print(d)  # 2020-06-25
print(type(d))  # <class 'datetime.date'>
```

【练习】如何统计两个日期之间有多少个星期六？

```
# 输入
d1 = datetime.date(1869, 1, 2)
d2 = datetime.date(1869, 10, 2)

# 输出
40
```

【参考答案】

```
import datetime

d1 = datetime.date(1869, 1, 2)
d2 = datetime.date(1869, 10, 2)
dt = (d2 - d1).days
print(dt)
print(d1.isoweekday())  # 6
print(dt // 7 + 1)  # 40
```

## 17.3 time类

```
class time:
    def __init__(self, hour, minute, second, microsecond, tzinfo):
        pass
```

【例子】如何使用 datetime.time() 类？

```
import datetime

t = datetime.time(12, 9, 23, 12980)
print(t)  # 12:09:23.012980
print(type(t))  # <class 'datetime.time'>
```

注意：

1. 1秒 = 1000 毫秒（milliseconds）
2. 1毫秒 = 1000 微妙（microseconds）

【练习】如何将给定日期转换为当天开始的时间？

```
# 输入
import datetime
date = datetime.date(2019, 10, 2)

# 输出
2019-10-02 00:00:00
```

【参考答案】

```
import datetime

date = datetime.date(2019, 10, 2)
dt = datetime.datetime(date.year, date.month, date.day)
print(dt)  # 2019-10-02 00:00:00

dt = datetime.datetime.combine(date, datetime.time.min)
print(dt)  # 2019-10-02 00:00:00
```

## 17.4 timedelta类

`timedelta` 表示具体时间实例中的一段时间。你可以把它们简单想象成两个日期或时间之间的间隔。

它常常被用来从 `datetime` 对象中添加或移除一段特定的时间。

```
class timedelta(SupportsAbs[timedelta]):
    def __init__(self, days, seconds, microseconds, milliseconds, minutes, hours, weeks,):
        pass
    def days(self):
        pass
    def total_seconds(self):
        pass
```

【例子】如何使用 datetime.timedelta() 类？

```
import datetime

td = datetime.timedelta(days=30)
print(td)  # 30 days, 0:00:00
print(type(td))  # <class 'datetime.timedelta'>
print(datetime.date.today())  # 2020-07-01
print(datetime.date.today() + td)  # 2020-07-31

dt1 = datetime.datetime(2020, 1, 31, 10, 10, 0)
dt2 = datetime.datetime(2019, 1, 31, 10, 10, 0)
td = dt1 - dt2
print(td)  # 365 days, 0:00:00
print(type(td))  # <class 'datetime.timedelta'>

td1 = datetime.timedelta(days=30)  # 30 days
td2 = datetime.timedelta(weeks=1)  # 1 week
td = td1 - td2
print(td)  # 23 days, 0:00:00
print(type(td))  # <class 'datetime.timedelta'>
```

如果将两个 datetime 对象相减，就会得到表示该时间间隔的 timedelta 对象。

同样地，将两个时间间隔相减，可以得到另一个 timedelta 对象。

【练习】

1. 距离你出生那天过去多少天了？
2. 距离你今年的下一个生日还有多少天？
3. 将距离你今年的下一个生日的天数转换为秒数。

```
# 输入
bday = 'Oct 2, 1969'
```

【参考答案】

```
from dateutil import parser
import datetime
```

```
3
4  bDay = 'Oct 2, 1969'
5  dt1 = parser.parse(bDay).date()
6  dt2 = datetime.date.today()
7  dt3 = datetime.date(dt2.year, dt1.month, dt1.day)
8  print(dt1)  # 1969-10-02
9  print(dt2)  # 2020-07-01
10 print(dt3)  # 2020-10-02
11
12 td = dt2 - dt1
13 print(td.days)  # 18535
14 td = dt3 - dt2
15 print(td.days)  # 93
16 print(td.days * 24 * 60 * 60)  # 8035200
17 print(td.total_seconds())  # 8035200.0
```

**练习题：**

1. 假设你获取了用户输入的日期和时间如`2020-1-21 9:01:30`，以及一个时区信息如`UTC+5:00`，均是`str`，请编写一个函数将其转换为timestamp：

   题目说明:

```
"""

Input file
example1: dt_str='2020-6-1 08:10:30', tz_str='UTC+7:00'
example2: dt_str='2020-5-31 16:10:30', tz_str='UTC-09:00'

Output file
result1: 1590973830.0
result2: 1590973830.0
"""


def to_timestamp(dt_str, tz_str):
    # your code here
    pass
```

2. 编写Python程序以选择指定年份的所有星期日。

   题目说明:

```
"""

Input file
2020

Output file
```

```
2020-01-05

2020-01-12

2020-01-19

2020-01-26

2020-02-02
-----
2020-12-06

2020-12-13

2020-12-20

2020-12-27
"""

def all_sundays(year):
    # your code here
```

# 18 文件与文件系统

## 18.1 打开文件

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

   Open file and return a stream.  Raise OSError upon failure.

   a. `file`: 必需，文件路径（相对或者绝对路径）。
   b. `mode`: 可选，文件打开模式
   c. `buffering`: 设置缓冲
   d. `encoding`: 一般使用utf8
   e. `errors`: 报错级别
   f. `newline`: 区分换行符

常见的`mode`如下表所示：

| 打开模式 | 执行操作 |
| --- | --- |
| 'r' | 以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。 |
| 'w' | 打开一个文件只用于写入。<br>如果该文件已存在则打开文件，并从开头开始编辑。<br>即原有内容会被删除。<br>如果该文件不存在，创建新文件。 |
| 'x' | 写模式，新建一个文件，如果该文件已存在则会报错。 |
| 'a' | 追加模式，打开一个文件用于追加。<br>如果该文件已存在，文件指针将会放在文件的结尾。<br>也就是说，新的内容将会被写入到已有内容之后。<br>如果该文件不存在，创建新文件进行写入。 |
| 'b' | 以二进制模式打开文件。一般用于非文本文件，如：图片。 |
| 't' | 以文本模式打开（默认）。一般用于文本文件，如：txt。 |
| '+' | 可读写模式（可添加到其它模式中使用） |

【例】打开一个文件，并返回文件对象，如果该文件无法被打开，会抛出`OSError`。

```
f = open('将进酒.txt')
print(f)
```

```
# <_io.TextIOWrapper name='将进酒.txt' mode='r' encoding='cp936'>

for each in f:
    print(each)

# 君不见，黄河之水天上来，奔流到海不复回。
# 君不见，高堂明镜悲白发，朝如青丝暮成雪。
# 人生得意须尽欢，莫使金樽空对月。
# 天生我材必有用，千金散尽还复来。
# 烹羊宰牛且为乐，会须一饮三百杯。
# 岑夫子，丹丘生，将进酒，杯莫停。
# 与君歌一曲，请君为我倾耳听。
# 钟鼓馔玉不足贵，但愿长醉不复醒。
# 古来圣贤皆寂寞，惟有饮者留其名。
# 陈王昔时宴平乐，斗酒十千恣欢谑。
# 主人何为言少钱，径须沽取对君酌。
# 五花马，千金裘，呼儿将出换美酒，与尔同销万古愁。
```

## 18.2 文件对象方法

1. `fileObject.close()` 用于关闭一个已打开的文件。关闭后的文件不能再进行读写操作， 否则会触发`ValueError`错误。

【例】

```
f = open("将进酒.txt")
print('FileName:', f.name)  # FileName: 将进酒.txt
f.close()
```

1. `fileObject.read([size])` 用于从文件读取指定的字符数，如果未给定或为负则读取所有。

【例】

```
f = open('将进酒.txt', 'r')
line = f.read(20)
print("读取的字符串: %s" % line)
# 读取的字符串: 君不见，黄河之水天上来，奔流到海不复回。

f.close()
```

1. `fileObject.readline()` 读取整行，包括 "\n" 字符。

【例】

```
f = open('将进酒.txt', 'r')
line = f.readline()
print("读取的字符串：%s" % line)
# 读取的字符串：君不见，黄河之水天上来，奔流到海不复回。
f.close()
```

1. `fileObject.readlines()`用于读取所有行(直到结束符 EOF)并返回列表，该列表可以由 Python 的 `for... in ...` 结构进行处理。

【例】

```
f = open('将进酒.txt', 'r')
lines = f.readlines()
print(lines)

for each in lines:
    each.strip()
    print(each)

# 君不见，黄河之水天上来，奔流到海不复回。
# 君不见，高堂明镜悲白发，朝如青丝暮成雪。
# 人生得意须尽欢，莫使金樽空对月。
# 天生我材必有用，千金散尽还复来。
# 烹羊宰牛且为乐，会须一饮三百杯。
# 岑夫子，丹丘生，将进酒，杯莫停。
# 与君歌一曲，请君为我倾耳听。
# 钟鼓馔玉不足贵，但愿长醉不复醒。
# 古来圣贤皆寂寞，惟有饮者留其名。
# 陈王昔时宴平乐，斗酒十千恣欢谑。
# 主人何为言少钱，径须沽取对君酌。
# 五花马，千金裘，呼儿将出换美酒，与尔同销万古愁。

f.close()
```

1. `fileObject.tell()`返回文件的当前位置，即文件指针当前位置。

【例】

```
f = open('将进酒.txt', 'r')
line = f.readline()
print(line)
# 君不见，黄河之水天上来，奔流到海不复回。
pos = f.tell()
print(pos)  # 42
f.close()
```

1. `fileObject.seek(offset[, whence])` 用于移动文件读取指针到指定位置。
    a. `offset`：开始的偏移量，也就是代表需要移动偏移的字节数，如果是负数表示从倒数第几位开始。
    b. `whence`：可选，默认值为 0。给 `offset` 定义一个参数，表示要从哪个位置开始偏移；0 代表从文件开头开始算起，1 代表从当前位置开始算起，2 代表从文件末尾算起。

【例】

```
f = open('将进酒.txt', 'r')
line = f.readline()
print(line)
# 君不见，黄河之水天上来，奔流到海不复回。
line = f.readline()
print(line)
# 君不见，高堂明镜悲白发，朝如青丝暮成雪。
f.seek(0, 0)
line = f.readline()
print(line)
# 君不见，黄河之水天上来，奔流到海不复回。
f.close()
```

1. `fileObject.write(str)` 用于向文件中写入指定字符串，返回的是写入的字符长度。

【例】

```
f = open('workfile.txt', 'wb+')
print(f.write(b'0123456789abcdef'))  # 16
print(f.seek(5))  # 5
print(f.read(1))  # b'5'
print(f.seek(-3, 2))  # 13
print(f.read(1))  # b'd'
```

在文件关闭前或缓冲区刷新前，字符串内容存储在缓冲区中，这时你在文件中是看不到写入的内容的。

如果文件打开模式带 `b`，那写入文件内容时，`str`（参数）要用 `encode` 方法转为 `bytes` 形式，否则报错：`TypeError: a bytes-like object is required, not 'str'`。

【例】

```
str = '...'
# 文本 = Unicode字符序列
# 相当于 string 类型

str = b'...'
# 文本 = 八位序列(0到255之间的整数)
# 字节文字总是以‘b’或‘B’作为前缀；它们产生一个字节类型的实例，而不是str类型。
# 相当于 byte[]
```

【例】

```
f = open('将进酒.txt', 'r+')
str = '\n作者：李白'
f.seek(0, 2)
line = f.write(str)
f.seek(0, 0)
for each in f:
    print(each)

# 君不见，黄河之水天上来，奔流到海不复回。
# 君不见，高堂明镜悲白发，朝如青丝暮成雪。
# 人生得意须尽欢，莫使金樽空对月。
# 天生我材必有用，千金散尽还复来。
# 烹羊宰牛且为乐，会须一饮三百杯。
# 岑夫子，丹丘生，将进酒，杯莫停。
# 与君歌一曲，请君为我倾耳听。
# 钟鼓馔玉不足贵，但愿长醉不复醒。
# 古来圣贤皆寂寞，惟有饮者留其名。
# 陈王昔时宴平乐，斗酒十千恣欢谑。
# 主人何为言少钱，径须沽取对君酌。
# 五花马，千金裘，呼儿将出换美酒，与尔同销万古愁。
# 作者：李白

f.close()
```

1. `fileObject.writelines(sequence)` 向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符 `\n`。

【例】

```
f = open('test.txt', 'w+')
seq = ['小马的程序人生\n', '老马的程序人生']
f.writelines(seq)
f.seek(0, 0)
for each in f:
    print(each)

# 小马的程序人生
# 老马的程序人生
f.close()
```

## 18.3 简洁的 with 语句

一些对象定义了标准的清理行为，无论系统是否成功的使用了它，一旦不需要它了，那么这个标准的清理行为就会执行。

关键词 with 语句就可以保证诸如文件之类的对象在使用完之后一定会正确的执行它的清理方法。

【例】

```
try:
    f = open('myfile.txt', 'w')
    for line in f:
        print(line)
except OSError as error:
    print('出错啦!%s' % str(error))
finally:
    f.close()

# 出错啦!not readable
```

这段代码执行完毕后，就算在处理过程中出问题了，文件 f 总是会关闭。

【例】

```
try:
    with open('myfile.txt', 'w') as f:
        for line in f:
            print(line)
except OSError as error:
    print('出错啦!%s' % str(error))

# 出错啦!not readable
```

# 19 OS 模块中关于文件/目录常用的函数

我们所知道常用的操作系统就有：Windows，Mac OS，Linu，Unix等，这些操作系统底层对于文件系统的访问工作原理是不一样的，因此你可能就要针对不同的系统来考虑使用哪些文件系统模块……，这样的做法是非常不友好且麻烦的，因为这样就意味着当你的程序运行环境一改变，你就要相应的去修改大量的代码来应对。

有了OS（Operation System）模块，我们不需要关心什么操作系统下使用什么模块，OS模块会帮你选择正确的模块并调用。

1. `os.getcwd()` 用于返回当前工作目录。
2. `os.chdir(path)` 用于改变当前工作目录到指定的路径。

【例】

```
import os

path = 'C:\\'
print("当前工作目录 : %s" % os.getcwd())
# 当前工作目录 : C:\Users\Administrator\PycharmProjects\untitled1
os.chdir(path)
print("目录修改成功 : %s" % os.getcwd())
# 目录修改成功 : C:\
```

1. `listdir (path='.')` 返回 `path` 指定的文件夹包含的文件或文件夹的名字的列表。

【例】

```
import os

dirs = os.listdir()
for item in dirs:
    print(item)
```

1. `os.mkdir(path)` 创建单层目录，如果该目录已存在抛出异常。

【例】

```
import os

if os.path.isdir(r'.\b') is False:
    os.mkdir(r'.\B')
    os.mkdir(r'.\B\A')

os.mkdir(r'.\C\A') # FileNotFoundError
```

1. `os.makedirs(path)` 用于递归创建多层目录，如果该目录已存在抛出异常。

【例】

```
import os
os.makedirs(r'.\E\A')
```

1. `os.remove(path)` 用于删除指定路径的文件。如果指定的路径是一个目录，将抛出 `OSError` 。

【例】首先创建 `.\E\A\text.txt` 文件，然后进行删除。

```
import os

print("目录为: %s" % os.listdir(r'.\E\A'))
os.remove(r'.\E\A\test.txt')
print("目录为: %s" % os.listdir(r'.\E\A'))
```

1. `os.rmdir(path)` 用于删除单层目录。仅当这文件夹是空的才可以, 否则, 抛出 `OSError` 。

【例】首先创建 `.\E\A` 目录，然后进行删除。

```
import os

print("目录为: %s" % os.listdir(r'.\E'))
os.rmdir(r'.\E\A')
print("目录为: %s" % os.listdir(r'.\E'))
```

1. `os.removedirs(path)` 递归删除目录，从子目录到父目录逐层尝试删除，遇到目录非空则抛出异常。

【例】首先创建 `.\E\A` 目录，然后进行删除。

```
import os

print("目录为: %s" % os.listdir(os.getcwd()))
os.removedirs(r'.\E\A')  # 先删除A 然后删除E
print("目录为: %s" % os.listdir(os.getcwd()))
```

1. `os.rename(src, dst)` 方法用于命名文件或目录，从 `src` 到 `dst` ，如果 `dst` 是一个存在的目录, 将抛出 `OSError` 。

【例】把test.txt文件重命名为test2.txt。

```
import os

print("目录为: %s" % os.listdir(os.getcwd()))
os.rename("test.txt", "test2.txt")
print("重命名成功。")
print("目录为: %s" % os.listdir(os.getcwd()))
```

1. `os.system(command)` 运行系统的shell命令（将字符串转化成命令）

【例】先自行创建一个a.py的文件，然后由shell命令打开。

```
import os

path = os.getcwd() + '\\a.py'
a = os.system(r'python %s' % path)

os.system('calc')  # 打开计算器
```

1. `os.curdir` 指代当前目录（`.`）
2. `os.pardir` 指代上一级目录（`..`）
3. `os.sep` 输出操作系统特定的路径分隔符（win下为`\\`，Linux下为`/`）
4. `os.linesep` 当前平台使用的行终止符（win下为`\r\n`，Linux下为`\n`）
5. `os.name` 指代当前使用的操作系统（包括：'mac'，'nt'）

【例】

```
import os

print(os.curdir)  # .
print(os.pardir)  # ..
print(os.sep)  # \
print(os.linesep)
print(os.name)  # nt
```

1. `os.path.basename(path)` 去掉目录路径，单独返回文件名
2. `os.path.dirname(path)` 去掉文件名，单独返回目录路径
3. `os.path.join(path1[, path2[, ...]])` 将 `path1`， `path2` 各部分组合成一个路径名
4. `os.path.split(path)` 分割文件名与路径，返回 `(f_path,f_name)` 元组。如果完全使用目录，它会将最后一个目录作为文件名分离，且不会判断文件或者目录是否存在。
5. `os.path.splitext(path)` 分离文件名与扩展名，返回 `(f_path,f_name)` 元组。

【例】

```python
import os

# 返回文件名
print(os.path.basename(r'C:\test\lsgo.txt'))  # lsgo.txt
# 返回目录路径
print(os.path.dirname(r'C:\test\lsgo.txt'))  # C:\test
# 将目录和文件名合成一个路径
print(os.path.join('C:\\', 'test', 'lsgo.txt'))  # C:\test\lsgo.txt
# 分割文件名与路径
print(os.path.split(r'C:\test\lsgo.txt'))  # ('C:\\test', 'lsgo.txt')
# 分离文件名与扩展名
print(os.path.splitext(r'C:\test\lsgo.txt'))  # ('C:\\test\\lsgo', '.txt')
```

1. `os.path.getsize(file)` 返回指定文件大小，单位是字节。
2. `os.path.getatime(file)` 返回指定文件最近的访问时间
3. `os.path.getctime(file)` 返回指定文件的创建时间
4. `os.path.getmtime(file)` 返回指定文件的最新的修改时间
5. 浮点型秒数，可用time模块的 `gmtime()` 或 `localtime()` 函数换算

【例】

```python
import os
import time

file = r'.\lsgo.txt'
print(os.path.getsize(file))  # 30
print(os.path.getatime(file))  # 1565593737.347196
print(os.path.getctime(file))  # 1565593737.347196
print(os.path.getmtime(file))  # 1565593797.9298275
print(time.gmtime(os.path.getctime(file)))
# time.struct_time(tm_year=2019, tm_mon=8, tm_mday=12, tm_hour=7, tm_min=8, tm_sec=57, tm_wday=0, tm_yday=224, tm_isdst=0)
print(time.localtime(os.path.getctime(file)))
# time.struct_time(tm_year=2019, tm_mon=8, tm_mday=12, tm_hour=15, tm_min=8, tm_sec=57, tm_wday=0, tm_yday=224, tm_isdst=0)
```

1. `os.path.exists(path)` 判断指定路径（目录或文件）是否存在
2. `os.path.isabs(path)` 判断指定路径是否为绝对路径
3. `os.path.isdir(path)` 判断指定路径是否存在且是一个目录
4. `os.path.isfile(path)` 判断指定路径是否存在且是一个文件
5. `os.path.islink(path)` 判断指定路径是否存在且是一个符号链接
6. `os.path.ismount(path)` 判断指定路径是否存在且是一个悬挂点
7. `os.path.samefile(path1,path2)` 判断path1和path2两个路径是否指向同一个文件

【例】

```
import os

print(os.path.ismount('D:\\'))  # True
print(os.path.ismount('D:\\Test'))  # False
```

# 20 序列化与反序列化

Python 的 pickle 模块实现了基本的数据序列和反序列化。

1. 通过 pickle 模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。
2. 通过 pickle 模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

pickle模块中最常用的函数为：

`pickle.dump(obj, file, [,protocol])` 将`obj`对象序列化存入已经打开的`file`中。

1. `obj`：想要序列化的`obj`对象。
2. `file`:文件名称。
3. `protocol`：序列化使用的协议。如果该项省略，则默认为0。如果为负值或`HIGHEST_PROTOCOL`，则使用最高的协议版本。

`pickle.load(file)` 将`file`中的对象序列化读出。

1. `file`：文件名称。

【例】

```
import pickle

dataList = [[1, 1, 'yes'],
            [1, 1, 'yes'],
            [1, 0, 'no'],
            [0, 1, 'no'],
            [0, 1, 'no']]
dataDic = {0: [1, 2, 3, 4],
           1: ('a', 'b'),
           2: {'c': 'yes', 'd': 'no'}}

# 使用dump()将数据序列化到文件中
fw = open(r'.\dataFile.pkl', 'wb')

# Pickle the list using the highest protocol available.
pickle.dump(dataList, fw, -1)

# Pickle dictionary using protocol 0.
```

```
pickle.dump(dataDic, fw)
fw.close()

# 使用load()将数据从文件中序列化读出
fr = open('dataFile.pkl', 'rb')
data1 = pickle.load(fr)
print(data1)
data2 = pickle.load(fr)
print(data2)
fr.close()

# [[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]
# {0: [1, 2, 3, 4], 1: ('a', 'b'), 2: {'c': 'yes', 'd': 'no'}}
```

---

**参考文献：**


1. https://www.runoob.com/python3/python3-tutorial.html
2. https://www.bilibili.com/video/av4050443


---

**练习题：**

1. 打开中文字符的文档时,会出现乱码,python自带的打开文件是否可以指定文字编码?还是只能用相关函数?
2. 编写程序查找最长的单词

   输入文档: res/test.txt

   题目说明:

```
"""

Input file
test.txt

Output file
['general-purpose,', 'object-oriented,']

"""
def longest_word(filename):
    # your code here
 pass
```