第1章 基础知识

1.1 如何选择Python版本

```
>>> import sys
>>> sys.version   #查看Python版本信息
'3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit
(AMD64)]'
>>> sys.winver    #查看Python版本号
'3.9'
>>> sys.version_info    #查看Python详情信息
sys.version_info(major=3, minor=9, micro=2, releaselevel='final', serial=0)
```

1.2 Python安装与简单实用

利用IDLE调试

创建文件：File → New File，保存为.py / .pyw(对于GUI程序)

检查程序：Run → Check Module

运行程序：Run → Run Module

cmd运行Python

cd 进入地址： python 文件名.py

IDLE常用快捷键 书p4

1.3 使用pip管理Python扩展库

常用pip命令使用方法

pip命令	说明
pip install SomePackage	安装SomePackage模块
pip list	列出当前已安装的所有模块
pip install --upgrade SomePackage	升级安装SomePackage模块
pip uninstall SomePackage	卸载安装SomePackage模块
pip install somePackage	whl使用whl文件直接安装安装SomePackage模块

1.4 Python基础知识

1.4.1 Python对象模型

1.4.2 Python变量

Python 是强类型编程语言：Python解释器会根据赋值或运算来自动推断变量类型。

Python 是动态类型语言：变量的类型是可以随时变化的。

type() 和 isinstance()

```
x = 3
print(type(x))      #查看变量类型
isinstance(3,int)   #判断变量类型，返回bool类型
```

字符串和元组属于不可变序列，其余是可变序列。

Python允许多个变量指向同一个值。

id():返回变量地址

del x :删除变量

查看关键字

```
import keyword
keyword.kwlist
```

dir(builins):查看内置模块，类型和函数。

1.4.3 数字

复数运算

```
>>> a = 5 + 2j
>>> b = 3 - 1j
>>> c = a - b
>>> c
(2+3j)
>>> c.real
2.0
>>> c.imag
3.0
>>> c.conjugate()
(2-3j)
>>> a * b
(17+1j)
>>> a / b
(1.2999999999999998+1.0999999999999999j)
```

1.4.4 字符串

''' ''': 支持换行，表示较长的注释

格式化:

raw-string: 在字符串前面加r或R表示原始字符串，其中特殊字符不进行转义，但字符串的最后一个字符不能是\。（主要用于正则表达，也开业用来简化路径或url输入）

转义字符 \n 换行符 \" 双引号 \t 制表符 \ 一个
\r 回车 \ddd 3位八进制数对应的字符 \' 单引号 \xhh 2为十六进制数对应的字符

1.4.5 运算符与表达式

x in y;x not in y 成员测试运算符 x is y;x is not y 对象实体词同一性测试（地址是否一致：有的值相同，但是地址不相同）

位运算符

| ^ & << >> ~

集合交集、并集、对称差集

Python中用','分割，会产生一个元组

Python

1.4.7 对象的删除 del x

可以使用del命令来显示的删除对象，并解除与值之间的指向关系。

del也可以删除列表或其他可变序列中的指定元素，也可以删除整个列表或者其他类型序列对象。

```
x = (1,2,3)
del x[1]
del x
```

1.4.8 基本输入输出

输入 input()接受用户的键盘输入。

Python2.x中，该函数返回结果的类型由输入值使用的界定符来决定；而raw_input()返回类型一律为字符串。

```
>>> x = input("input:")
input: [1,2,3]
>>> print type(x)
<type 'list'>

>>> x = raw_input("input:")
input: [1,2,3]
>>> print type(x)
<type 'str'>
```

Python3.x中，所有返回类型均是str类型。

输出 把文件输出到指定文件

Python2.x

```
>>> fp = open(r'url','a+')
>>> print>>'Hello'
>>> fp.close
```

Python3.x

```
>>> fp = open(r'url','a+')
>>> print('Hello',file = fp)
>>> fp.close
```

加“，”不换行的性质 Python2.x

```
for i in range(10):
    print i,
0 1 2 3 4 5 6 7 8 9
```

Python3.x

```
for i in range(10,20):
    print(i,end=' ')
10 11 12 13 14 15 16 17 18 19
```

1.4.9 模块导入与使用

sys.modules.items()显示所有预加载模块的相关信息。

import 模块名[as别名]

```
>>> import math as m
>>> m.sqrt(9)
3.0
```

from 模块名 import 对象名[as别名]

```
>>> from math import sqrt as f
>>> f(9)
3.0
```

重新导入模块 Python2.x中可以使用reload()；Python3.x中可以使用imp模块或importlib模块的reload()函数。

前提：该模块已经被正确加载，即第一次导入和加载模块时不能使用reload方法。

导入模块时，会优先导入相应的.pyc文件，如果相应的.pyc文件与.py文件时间不相符或不存在对应的.pyc文件，则导入.py文件并重新将该模块编译为.pyc文件。

模块导入顺序

- 导入Python标准款模块，os,sys,re
- 导入第三方扩展库,如PIL，numpy,scipy
- 导入自己定义和开发的本地模块。

第2章 Python序列

2.1 列表 []

2.1.1 列表创建与删除

```
list_a = []
list_b = ['a',2,'sirng']
```

可以利用 range 来创建list.

range函数

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

参数1： 起点（默认0）

参数2： 终点

参数3： 步长

删除列表，使用**del + list**

2.1.2 列表元素的增加

1）用"+"增加元素

```
>>> aList = [3,4,5]
>>> aList = aList + [6]
>>> aList
[3, 4, 5, 6]
```

2）append()方法

原地修改：内存地址不变，内存单元增加

```
>>> aList.append(9)
>>> aList
[3, 4, 5, 6, 9]
```

比较两个方法

```
import time

# 使用+方法，增加list 成员
result = []
start = time.time()
for i in range(10000):
    result = result + [i]
print(len(result), ',', time.time() - start)

# 使用append方法，增加list成员
result = []
start = time.time()
for i in range(10000):
    result.append(i)
print(len(result), ',', time.time() - start)
```

append更快，因为是直接原地增加元素，而不是更改指针指向。

基于内存的管理方式

对于list而言，list中包含的元素值的引用，而不是直接包含元素本身。

- 若直接修改序列变量的值，则地址改变
- 若通过下标修改，地址不变，元素地址改变。

```
a = [1,2,4]
b = [1,2,3]
a == b
False
a.append(4)
id (a) # 不变
```

```
a.remove(4)
id(a) # 不变
```

3）extend()方法

将另一个迭代对象的所有元素添加至该列表对象尾部。

不改变内存首地址，属于**原地操作**。

```
>>> a = [5,2,3]
>>> id(a)
1862358752576
>>> a.extend([7,8,9])
>>> id(a)
1862358752576
```

4）insert()方法

在列表的任意位置插入元素，但由于列表的自动内存管理功能，该方法会涉及到插入位置之后元素的移动，会影响处理速度。

尽量避免在列表中间插入和删除元素，建议优先使用append()和pop()方法

比较append()和insert处理速度的差别：

```
import time

# 测试：insert插入第一位和append插入最后一位的时间差
# 使用+方法，增加list 成员
def Insert():
    a = []
    for i in range(10000):
        a.insert(0, i)
    print(a)

def Append():
    a = []
    for i in range(10000):
        a.append(i)


start = time.time()
# 做十次
for i in range(10):
    Insert()
print('Insert:', time.time() - start)


start = time.time()
for i in range(10):
    Append()
```

```
print('Append:', time.time() - start)

# Insert: 0.370502233505249
# Append: 0.010970830917358398
```

5）用乘法扩展列表对象(适用：list , string, tuple)

将列表整数相乘，生成一个新列表(地址变化)，新列表是原列表的重复。

```
>>> a = [3,4,5]
>>> b = a
>>> id(a)
2699380239104
>>> a = a * 3  # 注意 a = a * 3时，地址改变；而a * 3时地址不变化
>>> a
[3, 4, 5, 3, 4, 5, 3, 4, 5]
>>> b
[3, 4, 5]
>>> id(a)
2697266430784 #地址改变，创建新列表
>>> id(b)
2699380239104
```

使用*运算符，创建元素的引用，当修改某一个值的时候，也会被修改。

Tricky Part

```
# 先构成list: [None,None],再把list作为最外层list的元素，重复三次。
x = [[None] * 2] * 3
print(x)

# x[0][0]修改的是[[None, None], [None, None], [None, None]]中，第一个元素的第一个元素。
# 而该list是又*创建的，因此三个全是对于[None,None]的引用，更改任何一个中的一个元素，其他三个元素均会被修改。
x[0][0] = 5
print(x)

# 原理同上
x = [[1, 2, 3]] * 3
x[0][0] = 10
print(x)
```

2.1.3 列表元素的删除

1）del命令

- 可以删除一整个列表
- 可以删除指定位置上的元素（默认删除最后一个元素）

```
>>> a = [1,2,3]
>>> del a[2]
>>> a
[1, 2]
>>> del a
>>> a
Traceback (most recent call last):
  File "<pyshell#73>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

2）pop()方法

删除元素并返回指定（默认为最后一个）位置上的元素。下标越界，则抛出异常。

```
>>> a = [1,2,3,4,5]
# 默认最后
>>> a.pop()
5
>>> a
[1, 2, 3, 4]
# 指定下标
>>> a.pop(2)
3
```

3）remove()方法

删除首次出现的指定元素，若不存在，抛出异常。

```
>>> a = [3,3,5,2,3,7,11]
>>> a.remove(3)
>>> a
[3, 5, 2, 3, 7, 11]
```

remove 的tricky

```
# 样例数组中不含有连续的1
def remove_1():
    x = [1, 2, 1, 2, 1, 2, 1, 2, 1]
    for i in x:
        if i == 1:
            x.remove(i)
    print(x)


# 样例数组中含有连续的1
def remove_2():
```

```
    x = [1, 2, 1, 1, 1, 2, 1, 1]
    for i in x:
        if i == 1:
            x.remove(i)
    print(x)

if __name__ == "__main__":
    remove_1()
    # [2, 2, 2, 2]
    remove_2()
#     [2, 2, 1, 1]
'''1删除不干净的原因是：Python会自动对列表内存进行收缩并移动列表元素，以保证所以元素之间没有空隙。
增加列表元素的时候也会移动。
    当读到一个数组中有连续的1的时候，删除前一个1后，下一个1会收缩，向前移。导致下一个1没有被判断到。'''
```

优化

```
# 利用切片修正
der remove_c():
    x = [1, 2, 1, 1, 1, 2, 1, 1]
    for i in x[::]:
        if i == 1:
            x.remove(i)
```

第3章 选择与循环

3.1 条件表达式

```
"""Python中，if条件表达式的值只要不是False,0,空
    都可以认为与True等价"""


def if_while_test():
    if 3:
        print(5)
    a = [1, 2, 3]

    if a:
        print(6)
    b = []

    if b:
        print(b)
    else:
        print('empty')

    i = s = 0
    # 1~10求和
```

```
    while True:
        s += i
        i += 1
        if i > 10:
            break
    print(s)
    # 1~10求和，range范围：[0,11),步长为1
    s = 0
    for i in range(0, 11, 1):
        s += i
    print(s)


def several_compare():
    print(1 < 2 < 3)
    print(1 < 2 > 3)
    print(1 < 3 > 2)
```

短路运算/惰性求值

```
def short_route():
    """逻辑运算符 and 和 or 可以利用短路求值和惰性求值的特点（只计算必须计算的表达式的值），来提高运行效率。
       例：and 运算，只要出现一个0，则值为0，之后其他内容不用计算。
           or 运算，只要出现一,1，则值为1，之后其他内容不用计算。"""
    '''功能：使用用户指定的分隔符将多个字符串连接成一个字符串，如果用户没有指定，则默认使用','。
        join()函数

        语法：   'sep'.join(seq)

        参数说明：
        sep：分隔符。可以为空
        seq：要连接的元素序列、字符串、元组、字典
        上面的语法即：以sep作为分隔符，将seq所有的元素合并成一个新的字符串

        返回值：返回一个以分隔符sep连接各个元素后生成的字符串'''
    # 定义1.0
    def Join(chList, sep=None):
        return (sep or ',').join(chList)

    # 定义2.0，利用默认参数
    def Join(chList, sep=','):
        return sep.join(chList)
    chTest = ['1', '2', '3', '4', '5']
    print(Join(chTest))
    print(Join(chTest, ':'))

# if条件判断不能使用=
# if a = 3: 会报错
```

- Python条件表达式中不允许使用'='

3.2 选择结构

3.2.1 单分支选择结构

```
def single_if():
    x = input('Input two numbers:')
    a, b = map(int, x.split())
    if a > b:
        a, b = b, a
    print(a, b)
```

3.2.2 双分支选择结构

```
def if_else():
    chTest = ['1', '2', '3', '4', '5']
    if chTest:
        print(chTest)
    else:
        print('empty')
```

- value1 if condition else value2

```
def v1_if_else_v2():
    """ value1 if condition else value2
        condition的值与True等价时，表达式的值位value1;否则，值为value2
    """
    a = 5
    print(6) if a > 5 else print(5)
    print(6 if a > 3 else 5)
    b = 6 if a > 13 else 9
    print(b)

    # 测试if的惰性，此时没有引入random库，但是由于5 > 3 == True，所以可以输出结果为
3.0
    import math
    x = math.sqrt(9) if 5 > 3 else random.randint(1, 100)
    print(x)
    # 测试if的惰性，此时必须引入random库，因为2 > 3 == False，所以需要调用random输
出结果
    import random
    x = math.sqrt(9) if 2 > 3 else random.randint(1, 100)
    print(x)
```

3.2.3 多分支选择结构

if...: elif: else:

3.2.4 选择结构的嵌套

3.2.5 选择结构的应用案例

```
def interview():
    age = 24
    subject = "计算机"
    college = "非重点"
    if (age > 25 and subject == "电子信息工程") or (college == '重点' and
subject == "电子信息工程") or (
            age <= 28 and subject == "计算机"):
        print('Congratulations!')
    else:
        print('I am sorry.')


def score_input():
    '''用户输入若干个成绩，求所有成绩的总和。每输入一个成绩后询问是否继续输入下一个成
绩，回答yes就继续输入下一个成绩，回答no停止输入成绩'''
    '''
    eval()函数
    描述
    eval() 函数用来执行一个字符串表达式，并返回表达式的值。

    语法
    以下是 eval() 方法的语法:

    eval(expression[, globals[, locals]])
    参数
    expression -- 表达式。
    globals -- 变量作用域，全局命名空间，如果被提供，则必须是一个字典对象。
    locals -- 变量作用域，局部命名空间，如果被提供，可以是任何映射对象。
    返回值
    返回表达式计算结果。

    raw_input() 和input的区别？？？
'''
    endFlag = 'yes'
    s = 0
    while endFlag.lower() == 'yes':
        x = input('请输入一个正整数: ')
        x = eval(x)
        if isinstance(x, int) and 0 <= x <= 100:
            s = s + x
        else:
            print('不是数字或者不符合要求')
        endFlag = raw_input('继续输入？(yes or no)')
        print("整数之和", s)


def CountDay(year, month, day):
    import time

    def demo(year, month, day):
        """编写程序，判断某天是某年第几天。
            本例要点是闰年判断时条件表达式的写法以及关系运算符、逻辑运算符和切片的运
用。"""
```

```
        day_month = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]  # 每月的
天数
        if year % 400 == 0 or (year % 4 == 0 and year % 100 != 0):  # 判断是否
为闰年
            day_month[1] = 29  # 闰年2月为29天
        if month == 1:
            return day
        else:
            return sum(day_month[:month - 1]) + day

    date = time.localtime()  # time.localtime()返回的是一个对象<class
'time.struct_time'>
    year, month, day = date[:3]  # date[:3]切片后返回的整体是一个元组，tuple;
    # 可以利用元组给"="左侧的变量赋值，赋值类型和元组中元素类型一致
    print(demo(year, month, day))
    '''date标准库提供了timedelta对象可以很方便的计算指定年，月，日，时，分，秒之前或
者之后的日期时间，
        还提供了返回结果包含“今天是今年第几天”，“今天是本周第几天”等答案的
timetuple()函数等等'''


def LearningTime():
    import datetime
    Today = datetime.date.today()  # 返回今天的日期 ： 年-月-日
    print(Today)
    ''' 原理： 今天日期 - 今年1月1日 + 一天'''
    delta = Today - datetime.date(Today.year, 1, 1) +
datetime.timedelta(days=1)  # 今天是今年的第几天
    print(delta)
    test1 = Today.timetuple().tm_yday  # 今天是今年的第几天
    print("输出今天是今年的第几天：", test1)
    test2 = Today.replace(year=2013)  # 替换日期中的年
    print("替换日期中的年：", test2)
    test3 = Today.replace(month=1)  # 替换日期中的月
    print("替换日期中的月：", test3)
    now = datetime.datetime.now()
    print("输出替换后的时间：", now)
    test4 = now.replace(second=30)  # 替换日期中的秒
    print("替换日期中的秒：", test4)
    test5 = now + datetime.timedelta(days=5)    # 计算5天后的日期时间
    print("计算5天后的日期时间", test5)
    test6 = now + datetime.timedelta(weeks=-5)    # 计算5周前的日期时间
    print("计算5周前的日期时间", test6)
```

input 和 raw_input的区别 time库

3.3 循环结构

- while循环一般用于循环次数难以提前确定的情况
- for循环一般用于循环次数可以提前确定的情况
- 优先考虑for循环
- 循环的else子句

- 循环自热结束时执行else结构中的else语句

```
"""3.3.2 循环结构优化"""
from pip._vendor.msgpack.fallback import xrange


def LoopAdvanced1():
    """尽量减少循环内部不必要的计算，将于循环变量无关的计算尽可能的提取到循环之外。
        尤其对于多重循环的情况。"""
    import time
    digits = (1, 2, 3, 4)

    start = time.time()
    for i in range(1000):
        result = []
        for i in digits:
            for j in digits:
                for k in digits:
                    result.append(i * 100 + j * 10 + k)
    print(time.time() - start)
    print(result)

    start = time.time()
    for i in range(1000):
        result = []
        for i in digits:
            i = i * 100
            for j in digits:  # 循环语句中的j并不受循环体中赋值的影响。
                j = j * 10
                for k in digits:
                    result.append(i + j + k)
    print(time.time() - start)
    print(result)


def LoopAdvanced2():
    """在循环中尽量引用局部变量，因为局部变量的查询和访问速度比全局变量略快。
        在使用模块中的方法时，可以通过将其转换为局部变量来提高运行速度。"""
    import time
    import math
    # method 1.1 直接使用
    start = time.time()  # 获取当前时间
    for i in range(10000000):
        math.sin(i)
    print('Time Used:', time.time() - start)  # 输出所用时间

    # method 1.2 利用局部变量转换
    loc_sin = math.sin  # 将模块的某一个方法变成局部变量
    start = time.time()
    """xrange() 函数用法与 range 完全相同，所不同的是生成的不是一个数组，而是一个生
成器。"""
    for i in xrange(10000000):
        loc_sin(i)
    print('Time Used:', time.time() - start)  # 输出所用时间
```

```
    """利用另一种导入和使用模块成员的方法,只导入所需要的模块也可以提升运行效率"""
    # method 2.1 直接使用
    import time
    from math import sin as sin
    start = time.time()
    for i in range(10000000):
        sin(i)
    print('Time Used:', time.time() - start)  # 输出所用时间

    # method 2.2 利用局部变量转换
    loc_sin = math.sin  # 将模块的某一个方法变成局部变量
    start = time.time()
    for i in range(10000000):
        loc_sin(i)
    print('Time Used:', time.time() - start)  # 输出所用时间
```

循环语句中的j并不受循环体中赋值的影响。

for语句中的i循环值为：1,2,3,4;

而for循环中的i值为100,200,...

```
    start = time.time()
    for i in range(1000):
        result = []
        for i in digits:
            i = i * 100
            for j in digits:        # 循环语句中的j并不受循环体中赋值的影响。
                j = j * 10
                for k in digits:
                    result.append(i + j + k)
    print(time.time() - start)
    print(result)
```

第4章 字符串与正则表达

4.0 编码标准

- 最早的字符串编码是美因标准信息交换码ASCII

4.1 字符串

- Python中字符串
  - 属于序列类型
  - 支持序列通用方法和切片操作
  - 支持特有的字符串操作方法
  - 字符串属于不可变序列类型
- Python字符串驻留机制
  - 短字符串赋给多个对象时，内存中只有一个副本

    - 长字符串不遵守驻留机制
- 判断Python字符串类型
    - 可以使用type()和instance()方法

4.1.1 字符串格式化

- 格式化方法
- chr() :unicode转换为char
- ord() :char转换为unicode

```
>>> chr(ord("3")+1)
'4'
>>> ord("3")
51
>>> ord("a")
97
>>> chr(97)
'a'
```

- 案例

```
# 进制间转换
>>> x = 1235
>>> so = "%o" %x
>>> so
'2323'
>>> sh = "%x" %x
>>> sh
'4d3'
>>> se = "%e" %x
>>> se
'1.235000e+03'
>>> chr(ord("3")+1)
'4'
>>> ord("3")
51
>>> ord("a")
97
>>> chr(97)
'a'
>>>
>>> "%s %65"
'%s %65'
>>> "%s" %65
'65'
>>> '%d,%c'%(65,65)
'65,A'
>>> '%d,%c'%(97,97)
'97,a'
>>> '%d'%555
'555'
>>> '%d'%'555'
```

```
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    '%d'%'555'
TypeError: %d format: a number is required, not str
>>> '%s'%[1,2,3]
'[1, 2, 3]'
>>>
```

- 使用format方法进行格式化
  - '{index:格式化方式}'.format(带入的字符)

```
def format():
    print("The number {0:,} in hex is : {0:#x}, the number {1} in oct is
{1:#o}".format(5555, 55))
    # The number 5,555 in hex is : 0x15b3, the number 55 in oct is 0o67
    print("The number {1:,} in hex is : {1:#x}, the number {0} in oct is
{0:#o}".format(5555, 55))
    # The number 55 in hex is : 0x37, the number 5555 in oct is 0o12663
    # 把元组中的元素作为列表中的一项，二维调用
    positon = (5, 8, 13)
    print("X:{0[0]};Y:{0[1]};Z:{0[2]}".format(positon))
#     X:5;Y:8;Z:13
```

4.1.2 字符串常用方法

- dir(""):查看所有字符串操作函数列表
- help()查看每个函数帮助
- find(),rfind(): 查找一个字符串在另一个字符串指定范围（默认是整个字符串）中首次和最后一次出现的位置，如果不存在则返回-1.
- index(),rindex()
- count()
- split(),rsplit()
- partition(),rpartition()
- 字符串连接join()
- lower(),upper(),capitalize(),title(),swapcase()
- replace()
- maketrans(),translate()
- strip(),rstrip(),lstrip()
- eval()
- 关键字in
- startswith(),endswith()
- isalnum(),isalpha(),isdigit(),isspace(),isupper(),islower()
- center(),ljust(),rjust()

4.1.3 字符串常量

- string.digits

- string.punctuation
- string.letters
- string.printable
- string.lowercase
- string.uppercase
- random库
    - random.getranbits(17)
    - random.choice()
    - random.randrange()
    - random.randint()
    - random.shuffle(list)

可变字符串

4.2 正则表达式

- 正则表达式
    - 字符串处理的有力工具和技术
- 正则表达式原理
    - 使用预定义模式去匹配一类具有共同特征的字符串
    - 处理字符串：快速、正确地完成复杂的查找、替换等
- Python中re模块提供了正则表达式操作所需要的功能。

4.2.1 正则表达式元字符

- 普通字符串可以匹配自身

4.2.2 re模块的主要方法（要重视这几个函数,参数,返回值,功能;判断时字符串内置还是re模块的方法）

- compile()
- search()

4.2.3 直接使用re模块方法

- 一般re模块方法的第一个是由通配符组成的模式串。

```
import re
text = 'alpha. beta....gamma delta'
re.split('[\.]+',text)
# ['alpha', 'beta', 'gamma', 'delta']
```

4.2.4 使用正则表达式对象

第5章 函数的设计与使用

5.1 函数定义与调用

定义函数需要注意：

- 函数形参不需要声明其类型，也不需要指定函数返回值类型。
- 即使该函数不需要接收任何参数，也必须保留一对空的圆括号。
- 括号后面的冒号必不可少。
- 函数体相对与def关键字必须保持一定的空格缩进。
- Python允许嵌套定义函数，并且所包含__call__()方法的类的对象均被认为是可调用的。 (详见5.8)
- 如果为函数的定义加上一段注释，可以为用户提供友好的提示和使用帮助。
    - 注释也要缩进。
    - 注释不是必须的。
    - 添加注释后，输入左侧圆括号之后，立刻就会得到该函数的使用说明。

5.2 形参与实参

- 函数定义时圆括号内时使用逗号分隔开的形参列表。
- 一个函数可以没有形参，但是定义时一对圆括号必须有，表示这是一个函数并且不接收参数。
- 函数调用时向其传递实参，根据不同的参数类型，将实参的值或引用传递给形参。
- 在定义函数时，对参数个数并没有限制，如果有多个形参，则需要使用逗号进行分隔。

```
'''接收两个参数，并输出其中的最大值。'''
def printMax(a,b):
    if a>b:
        print(a,'is the max')
    else:
        print(b,'is the max')
```

- 绝答多数情况下，在函数内部直接修改形参的值不会影响实参。

```
def addOne(a):
    print(a)
    a += 1
    print(a)


a = 3
addOne(a)
# 3
# 4
print(a)  # 实参a 没有被修改
# 3
```

- 有些情况下，可以通过特殊的方式在函数内部修改实参的值。

- 如果传递给函数的是Python可变序列，并且在函数内部使用下标或其他方式为可变序列增加，删除元素或修改元素值时，修改后的结果是可以反映到函数之外的。即实参也被修改。

```
def modify(v):  # 修改列表元素值
    v[0] = v[0] + 1


a = [2]
modify(a)
print(a)


# [3]

def modify(v, item):  # 为列表增加元素值
    v.append(item)


a = [2]
modify(a, 3)
print(a)


# [2, 3]

def modify(d):  # 修改字典元素值或者为字典添加元素
    d['age'] = 38


a = {'name': 'Dong', 'age': 37, 'sex': 'Male'}
modify(a)
print(a)
# {'name': 'Dong', 'age': 38, 'sex': 'Male'}
```

第6章 面向对象程序设计

6.1 类的定义与使用

6.1.1 类定义语法

- 创建一个类

```
class Car:  # 新式类必须有至少一个基类
    def infor(self):
        print("This is a car")
```

- 实例化对象

```
car = Car()
car.infor() # 通过对象名调用方法。
# This is a car
```

-内置方法`isinstance()`:测试一个对象是否为某个类的实例。

```
isinstance(car, Car)
isinstance(car, str)
# True
# False
```

- 关键字 pass:类似空语句，提供占位功能。

```
class A:
    pass


def demo():
    pass


if 5 > 3:
    pass
```

6.1.2 self参数

- 类的所有实例方法都必须至少有一个"self"的参数
- "self"参数必须是方法的第一个形参
- "self"参数代表将要创建的对象本身
- 在类的实例方法中访问实例属性是需要以self为前缀
- 在类外部用对象名调用对象方法时并不需要传递self参数
- 在类外部通过类名调用对象方法需要显式为self参数传值
- "self"参数名字是可以变化的

```
class A:
    def __init__(lalala, v):
        lalala.value = v

    def show(lalala):
        print(lalala.value)


a = A(3)
a.show()
# 3
```

要注意this指针和self指针的作用，指向的是实例自己

6.1.3 类成员与实例成员

- 类成员：指数据成员，或者广义上的属性
- 属性有两种：一种是实例属性；另一种是类属性
    - 实例属性
        - 一般是指在构造函数__init__()中定义的，使用时必须以self作为前缀
        - 实例属性属于实例（对象），只能通过对象名访问
    - 类属性
        - 在类中所有方法之外定义的数据成员
        - 类属性属于类，可以通过类名或对象名访问
- 类的方法中可调用类本身的其他方法，可访问类和类对象属性
- Python中可以动态地为类和对象增加成员

```
import types


class Car:
    price = 100000  # 定义类属性

    def __init__(self, c):
        self.color = c  # 定义实例属性


car1 = Car('Red')
car2 = Car('Blue')
print(car1.color, Car.price)
# Red 100000 创建实例成功
Car.price = 110000  # 修改类属性，通过类名访问
Car.name = "QQ"  # 增加类属性
car1.color = "Yellow"  # 修改实例属性，通过对象名访问
print(car2.color, Car.price, Car.name)
# Blue 110000 QQ 可以通过对象名.属性名/类名.属性名的形式访问数据成员
print(car1.color, Car.price, Car.name)
# Yellow 110000 QQ
print(car1.color, car1.price, car1.name)
# Yellow 110000 QQ
```

- Python中可以动态地为类和对象增加成员

```
def setSpeed(self, s):
    self.speed = s


car1.setSpeed = types.MethodType(setSpeed, Car)  # 动态为对象增加成员方法
car1.setSpeed(50)
print(car1.speed)
# 50 可以动态的为对象增加成员方法要使用types的MethodType方法
```

- 只有MethodType才能为类添加类的方法，否则只是显式的调用函数本身。

```
def addMethod():
    car2.setSpeed = setSpeed  # 把setSpeed函数赋给car2的一个成员变量
    print(car2.setSpeed)  # 此时setSpeed只是在内存中的一个函数
    # <function setSpeed at 0x0000019EE6724700>
    car2.setSpeed(car2, 50)  # 调用setSpeed(self,s)函数需要传入self指针和s数值
    print(car2.speed)
    # 50
    car2.setSpeed = types.MethodType(setSpeed, Car)  # 动态为对象增加成员方法
    print(car2.setSpeed)  # 此时setSpeed函数是类内部的一个方法
    # <bound method setSpeed of <class '__main__.Car'>>
    car2.setSpeed(50)  # 通过对象名设置数据成员的数值
    print(car2.speed)
    # 50
```

6.1.4 私有成员与公有成员

- Python没有对私有成员提供严格的访问保护机制。
    - 在定义类的属性时，如果属性名以两个下划线"__"开头则表示是私有属性。
    - 私有属性在类的外部不能直接访问，需要通过调用对象的公有成员方法来访问，或者通过Python支持的特殊方法来访问。
    - Python提供了访问私有属性的特殊方法，可用于程序的测试和调试，对于成员方法也具有同样的性质。
    - 私有属性是为了数据封装和保密而设的属性，一般只能在类的成员方法（类的内部）中使用访问。
        - 虽然Python支持一种特殊的方法来从外部直接访问类的私有成员，但是并不推荐这样做。
    - 公有属性是可以公开使用的，既可以在类的内部进行访问，也可以在外部程序中使用。
- 对于私有成员的方法要通过：对象名._类名__私有成员名 访问

```
class A:
    def __init__(self, value1=0, value2=0):
        self._value1 = value1
        self.__value2 = value2

    def setValue(self, value1, value2):
        self._value1 = value1
        self.__value2 = value2

    def show(self):
        print(self._value1)
        print(self.__value2)


a = A()
print(a._value1)    # _value1只是保护成员
```

```
print(a._A__value2)  # __value2是私有成员，不能直接访问。在外部访问对象的私有成员
                     #  特殊方法：对象名._类名__私有成员名
```

- Python 中，以下划线开头的变量名和方法名有特殊的含义，尤其是在类的定义中。用下划线作为变量名和方法名前缀和后缀表示类的特殊成员。
  - _xxx: 这样的对象叫做保护成员，不能用"from module import *"导入，只有类对象和子类对象能访问这些成员。
  - **xxx**: 系统定义的特殊成员。
  - __xxx: 类中的私有成员，只有类对象自己能访问，子类对象也不能访问到这个成员，但在对象外部可以通过“对象名._类名__私有成员名”这样的特殊方式类访问。
  - **Python中不存在严格意义上的私有成员。**
  - 在IDLE交互模式下，一个下划线'_'表示解释器中最后一次显示的内容或最后一次语句正确执行的输出结果。

```
>>> 3 + 5
8
>>> _ + 2
10
>>> _ * 3
30
>>> _ / 5
6.0
>>> 1 / 0
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    1 / 0
ZeroDivisionError: division by zero
>>> _
6.0
```

- 特殊成员定义和访问的方法

```
class Fruit:
    def __init__(self):
        self.__color = 'Red'
        self.price = 1


apple = Fruit()
print(apple.price)  # 显示对象公开数据成员的值
# 1
apple.price = 2  # 修改对象公开数据成员的值
print(apple.price)
# 2 apple的公开数据成员被修改
print(apple.price, apple._Fruit__color)  # 显示对象私有数据成员的值
# 2 Red 显示apple的私有数据成员
apple._Fruit__color = 'Blue'  # 修改对象私有数据成员的值
print(apple.price, apple._Fruit__color)
# 2 Blue 私有成员被修改
print(apple.__color)
```

```
# AttributeError: 'Fruit' object has no attribute '__color'
# 私有成员不能直接被访问
peach = Fruit()
print(peach.price,peach._Fruit__color)
# 1 Red 对于对象私有成员和共有成员的修改只是针对该对象，而不影响类的私有/共有成员函数
```

6.2 方法

类中定义的方法可以粗略分为四大类：

- 公有方法
- 私有方法
- 静态方法
- 类方法
- 公有方法
    - 属于对象，可以访问属于类和对象的成员，公有方法通过对象名直接调用，通过类名来调用属于对象的公有方法，需要显式为该方法"self"参数传递一个对象名，用来确定访问哪个对象的数据成员。
- 私有方法
    - 属于对象，私有方法的名字以两个下划线“__”开始，可以访问属于类和对象的成员，私有方法在属于对象的方法中通过“self”调用或在外部通过Python支持的特殊方法来调用
- 静态方法
    - 通过类名和对象名调用，不能直接访问属于对象的成员，只能访问属于类的成员
- 类方法
    - 通过类名和对象名调用，不能直接访问属于对象的成员，只能访问属于类的成员，一般将“cls”作为类方法的第一个参数名称，但也可以使用其他的名字作为参数，并且在调用类方法时不需要为“cls”参数传递值
- 调用:
    - 对象名.公有方法； 对象名.类方法；对象名.静态方法
    - 类名.公有方法(对象名)；类名.类方法；类名.静态方法
    - 私有方法的调用
        - 对象方法中self.**init**(self,v)
        - 外部特殊方法

```
class Root:
    __total = 0  # 定义类的私有数据成员

    def __init__(self, v):  # 定义私有方法（构造函数）
        self.__value = v
```

```
        Root.__total += 1

    def show(self):  # 定义共有方法，self传递一个对象名，确定操作对象
        print('self.__value:', self.__value)
        print('Root.__total:', Root.__total)

    @classmethod  # 类方法的标记
    def classShowTotal(cls):  # 类方法，cls作为第一个参数名
        print(cls.__total)

    @staticmethod
    def staticShowTotal():  # 静态方法
        print(Root.__total)


r = Root(3)
r.classShowTotal()  # 通过对象来调用类方法
# 1
r.staticShowTotal()  # 通过对象来调用静态方法
# 1
r.show()  # 通过对象来调用公有方法
# self.__value: 3
# Root.__total: 1
rr = Root(5)
Root.classShowTotal()  # 通过类名调用类方法
# 2
Root.staticShowTotal()  # 通过类名调用类方法
# 2
# Root.show()  # 试图通过类名直接调用实例方法，失败
# TypeError: show() missing 1 required positional argument: 'self'
Root.show(r)  # 可以通过这种方法来调用方法并访问实例成员
# self.__value: 3
# Root.__total: 2
r.show()
# self.__value: 3
# Root.__total: 2
Root.show(rr)       # 通过类名调用实例方法时为self参数显示传递对象名
# self.__value: 5
# Root.__total: 2
rr.show()
# self.__value: 5
# Root.__total: 2
```

6.3 属性

Python 2.x和Python3.x对属性的实现和处理方式不一样，内部实现有较大的差异，使用时应注意二者之间的区别。

属性的作用

```
# 廖雪峰 @property的使用 补充
class Student(object):
    def __init__(self, name):
```

```
        self._name = name
    '''属性的作用：我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。'''
# 设置name的get属性
    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        if isinstance(name, str):
            self._name = name
        else:
            print('type error!!!')

    def show(self):
        print(self._name)


s = Student('wendy')
s.show()
s.name = 'lily'  # 相当于s._name = 'lily' ,但是调用的name的set函数;增加了对于输入值的判断
print(s.name)    # 相当于s._name，调用的name的get函数。
s.name = 1      # 通过name的set函数判断输入类型不合理，所以没有进行对象成员的修改。
print(s.name)
print(dir(s))
```

6.3.1 Python 2.x 中的属性

- 使用@property或property()函数来声明一个属性
- 然而属性并没有得到真正意义的实现，也没有提供应有的访问保护机制。
- 在Python 2.x中，为对象增加新的数据成员时，将隐藏同名的已有属性。

```
class Test:
    def__init__(self,value):
        self.__value = value
    @property
        def value(self):
            return self.__value

a = Test(3)
print(a. value)
# 3
a.value = 5 # 动态添加了新成员，隐藏了定义的属性
            # 并不是对于Test类的私有变量value的修改，而是添加了一个同名的变量value，并且赋值为5
print(a. value)
# 5
t._Test__Value  # 原来的私有变量没有改变
# 3
```

- 除了动态增加成员时隐藏已有属性，下面的代码从表面来看是修改属性的值，而实际上也是增加了新成员，从而隐藏了已有属性。

```
class Test:
    def __init__(self,value):
        self.__value = value

    def __get(self):
        return self.__value

    def __set(self,v):
        self.__value = v

    value = property(__get,__set)    # 表示对于私有成员value赋予get和set的权限

    def show(self):
        print self.__value

t = Test(3)
print(t.value)
# 3
t.value+=2  # 动态添加新成员
print(t.value)  # 这里访问的是新成员
# 5
t.show()    # 访问原来定义的私有数据成员
# 3
del t.value # 这里删除的是刚才添加的新成员
print(t.value) # 访问原来的属性
# 3
del t.value # 试图删除属性
AttributeError: Test instance has no attribute 'value'
```

- Python 2.x 中私有成员和普通成员之间的关系

```
class Test:
    def show(self):
        print self.value
        print self.__v

t = Test()
t.show()    # 没有给实例传初始化的值
AttributeError: Test instance has no attribute 'value'
t.value = 3 # 通过对象增加新成员3
t.show()
# 3
AttributeError: Test instance has no attribute '_Test__v'
t.__v = 5   # 不可以通过.语法来访问对象的私有成员
t.show()
# 3
AttributeError: Test instance has no attribute '_Test__v'
t._Test__v = 5 # 可以通过_类名__私有成员名的方式，访问对象的私有成员
t.show()
```

```
# 3
# 5
```

6.3.2 Python 3.x 中的属性

- 属性得到了较为完整的实现，支持更加全面的保护机制。
- 如果设置属性为只读，则无法修改其值，也无法为对象增加与属性同名的新成员。也无法删除对象属性。

```
class Test:
    def __init__(self, value):
        self.__value = value

    @property
    def value(self):  # 只读，无法修改和删除
        return self.__value


t = Test(3)
print(t.value)
# 3
t.value = 5  # 只读属性不允许修改值
# AttributeError: can't set attribute
t.v = 5  # 动态增加新成员
print(t.v)
del t.v  # 动态删除成员
print(t.v)  # 成功删除成员
# AttributeError: 'Test' object has no attribute 'v'
del t.value  # 试图删除对象属性，失败
# AttributeError: can't delete attribute
print(t.value)   # 显示属性值
# 3
```

- 把属性设置为可读，可修改，而不允许删除。
    - `value = property(__get, __set) # 通过property()分配权限`

```
''' 把属性设置为可读，可修改，而不允许删除。'''
class Test:
    def __init__(self,value):
        self.__value = value

    def __get(self):
        return self.__value

    def __set(self, v):
        self.__value = v

    value = property(__get, __set)  # 通过property()分配权限

    def show(self):
        print(self.__value)
```

```
t = Test(3)
print(t.value)  # 允许读取属性值
# 3
t.value = 5  # 允许修改属性值
print(t.value)
# 5
t.show()  # 属性对应的私有变量也得到了相应的修改
# 5
'''del t.value  # 试图删除属性，失败
# AttributeError: can't delete attribute'''
```

- 把属性设置为可读，可修改，可删除。
    - value = property(__get, __set, __del) # 通过property()分配权限

```
''' 把属性设置为可读，可修改，可删除。'''


class Test:
    def __init__(self, value):
        self.__value = value

    def __get(self):
        return self.__value

    def __set(self, v):
        self.__value = v

    def __del(self):
        del self.__value

    value = property(__get, __set, __del)

    def show(self):
        print(self.__value)


t = Test(3)
t.show()
# 3
t.value = 5
t.show()
# 5
del t.value
''' # 删除了对象的属性
print(t.value)
#AttributeError: 'Test' object has no attribute '_Test__value'
t.show()
#AttributeError: 'Test' object has no attribute '_Test__value'
'''
t.value = 1  # 为对象动态增加属性和对应的私有数据成员
t.show()
# 1
```

```
print(t.value)
# 1
```

6.4 特殊方法与运算符重载

6.4.1 常用特殊方法

- Python中类的构造函数是__init__()
  - 一般用来为数据成员设置初值或进行其他必要的初始化工作。
  - 在创建对象时被自动调用和执行，可以通过为构造函数定义默认值参数类实现类似于其他语言中构造函数重载的目的。
  - 如果用户没有涉及构造函数，Python将提供一个默认的构造函数用来进行必要的初始化工作。
- Python中类的析构函数是__del__()
  - 一般用来释放对象占用的资源。
  - 在Python删除对象和收回对象空间时被自动调用和执行。
  - 如果用户没有编写析构函数，Python将提供一个默认的析构函数进行必要的清理工作。
  - 其他方法

表 6-1 Python 类特殊方法

方 法	功 能 说 明
__init__()	构造函数，生成对象时调用
__del__()	析构函数，释放对象时调用
__add__()、__radd__()	左+、右+
__sub__()	-
__mul__()	*
__div__()、__truediv__()	/，Python 2.x 使用__div__()，Python 3.x 使用__truediv__()
__floordiv__()	整除
__mod__()	%
__pow__()	**
__cmp__()	比较运算
__repr__()	打印、转换
__setitem__()	按照索引赋值
__getitem__()	按照索引获取值
__len__()	计算长度
__call__()	函数调用
__contains__()	测试是否包含某个元素
__eq__()、__ne__()、__lt__()、__le__()、__gt__()、__ge__()	==、!=、<、<=、>、>=
__str__()	转化为字符串
__lshift__()、__rshift__()	<<、>>
__and__()、__or__()、__invert__()	&、|、~
__iadd__()、__isub__()	+=、-=

6.4.2 案例精选

- 补充
  - 在Python中，可以同一行显示多条语句，只需用分号“;”隔开即可。
  - 在Python中，可以使用反斜杠()将一行语句分为多行解释。但是语句包含的{}、[]、()中的内容不需要使用多行连接符。

```
"""6-1 自定义一个数组类，支持数组与数字之间的四则运算数组之间的加法运算、内积运算和大小比较，数组元素访问和修改，以及成员测试等功能"""


class MyArray:
    """All the elements in this array must be numbers"""
    __value = []  # 类的私有对象
    __size = 0

    def __IsNumber(self, n):
        """
        在Python中，可以使用反斜杠(\)将一行语句分为多行解释。但是语句包含的{}、[]、()中的内容不需要使用多行连接符。
        int:整型; float:浮点数; complex:复数
        """
        if (not isinstance(n, int)) and \
                (not isinstance(n, float)) and \
                (not isinstance(n, complex)):
            return False
        return True

    def __init__(self, *args):
        if not args:
            self.__value = []
        else:
            for arg in args:
                if not self.__IsNumber(arg):
                    print('All elemtents must be numbers')
                    return
                self.__value = list(args)

    def __add__(self, n):  # 数组中每个元素都与数字n相加，或两个数组相加
        if self.__IsNumber(n):  # 若传入的参数n是数字
            b = MyArray()
            for v in self.__value:
                b.__value.append(v + n)
            return b
        elif isinstance(n, MyArray):  # 若传入的参数n是数组
            if len(n.__value) == len(self.__value):
                c = MyArray()
                for i, j in zip(self.__value, n.__value):
                    c.__value.append(i + j)
                return c
            else:
                print('Length not equal')
        else:
            print('Not supported')

    def __sub__(self, n):  # 数组中每个元素都与数字n相减，返回新数组
        if not self.__IsNumber(n):
```

```
            print('-operating with', type(n), 'and number type is not
supported.')
            return
        b = MyArray()
        for v in self.__value:
            b.__value.append(v - n)
        return b

    def __mul__(self, n):  # 数组中每个元素都与数字n相乘，返回新数组
        if not self.__IsNumber(n):
            print('* operating with', type(n), 'and number type is not
supported.')
            return
        b = MyArray()
        for v in self.__value:
            b.__value.append(v * n)
        return b

    def __truediv__(self, n):  # 数组中每个元素都与数字n相乘，返回新数组
        if not self.__IsNumber(n):
            print(r'/ operating with', type(n), 'and number type is not
supported.')
            return
        b = MyArray()
        for v in self.__value:
            b.__value.append(v / n)
        return b

    def __floordiv__(self, n):  # 数组中每个元素都与数字n整除，返回新数组
        if not isinstance(n, int):
            print(n, ' is not an integer')
            return
        b = MyArray()
        for v in self.__value:
            b.__value.append(v // n)
        return b

    def __mod__(self, n):  # 数组中每个元素都与数字n求余数，返回新数组
        if not self.__IsNumber(n):
            print(r'% operating with', type(n), 'and number type is not
supported.')
            return
        b = MyArray()
        for v in self.__value:
            b.__value.append(v % n)
        return b

    def __pow__(self, n):  # 数组中每个元素都与数字n进行幂计算，返回新数组
        if not self.__IsNumber(n):
            print('** operating with', type(n), 'and number type is not
supported.')
            return
        b = MyArray()
        for v in self.__value:
            b.__value.append(v ** n)
        return b
```

```
    def __len__(self):
        return len(self.__value)

    def __repr__(self):  # 直接使用对象作为语句时调用该函数
        # equibalent to return'self.__value)
        return repr(self.__value)

    def __str__(self):  # 使用print()函数输出对象时调用该函数
        return str(self.__value)

    def append(self, v):  # 追加元素
        if not self.__IsNumber(v):
            print('Only number can be appended.')
            return
        self.__value.append(v)

    def __getitem__(self, index):  # 获取指定位置的元素值
        if self.__IsNumber(index) and 0 <= index < len(self.__value):
            return self.__value[index]
        else:
            print('Index out of range.')

    def __setitem__(self, index, v):  # 设置指定位置的元素值
        if not self.__IsNumber(v):
            print(v, 'is not a number')
        elif (not isinstance(index, int) or index < 0 or index >=
len(self.__value)):
            print('Index type error or out of range.')
        else:
            self.__value = v

    def __contains__(self, v):  # 成员测试运算符in
        if v in self.__value:
            return True
        return False

    def dot(self, v):  # 模拟向量内积
        if not isinstance(v, MyArray):
            print(v, 'must be an instance of MyArray.')
            return
        if len(v) != len(self.__value):
            print('The size must be equal.')
            return
        b = MyArray()
        for m, n in zip(v.__value, self.__value):
            b.__value.append(m * n)
        return sum(b.__value)

    def __eq__(self, v):  # 关系运算符==
        if not isinstance(v, MyArray):
            print(v, 'must be an instance of MyArray.')
            return False
        if self.__value == v.__value:
            return True
        return False

    def __lt__(self, v):  # 关系运算符<
```

```
        if not isinstance(v, MyArray):
            print(v, 'must be an instance of MyArray.')
            return False
        if self.__value < v.__value:
            return True
        return False


if __name__ == '__main__':
    print('Please use me as a module.')
```

测试代码:

```
from MyArray import MyArray

x = MyArray(1, 2, 3, 4, 5, 6)
y = MyArray(6, 5, 4, 3, 2, 1)

print(len(x))
# 6

print(x + 5)
# [6, 7, 8, 9, 10, 11]

print(x * 3)
# [3, 6, 9, 12, 15, 18]

print(x.dot(y))
# 56

x.append(7)
print(x)
# [1, 2, 3, 4, 5, 6, 7]

print(x.dot(y))
# The size must be equal.

'''
x[9] = 8
# Index type error or out of range.
'''

print(x / 2)
# [0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5]

print(x // 2)
# [0, 1, 1, 2, 2, 3, 3]

print(x % 3)
# [1, 2, 0, 1, 2, 0, 1]

print(x[2])
# 3
```

```
print('a' in x)
# False

print(x < y)
# True

x = MyArray(1, 2, 3, 4, 5, 6)
print(x + y)
# [7, 7, 7, 7, 7, 7]
```

6.5 继承机制

- 继承的目的是代码复用和设计复用
- 继承关系中，已有的、设计好的类称为父类或基类，新设计的类称为子类或派生类。
- **派生类可以继承父类的公有成员，但是不能继承其私有成员。**
- 如果需要在派生类中调用基类的方法，**可以使用内置函数super()，或者通过“基类名.方法()”的方式来实现。**

```
        # 调用基类构造方法初始化基类的私有成员，注意使用格式
        # super(Teacher,self)返回的是基类的指针，推荐使用super()方法调用基类构造。
        super(Teacher, self).__init__(name, age, sex)
```

- **Python支持多继承**，如果父类中有相同的方法名，而在子类中使用是没有指定父类名，则Python解释器将从左向右按顺序搜索。

案例

```
''' 6.2 在派生类中调用基类方法
    首先设计Person类，然后以Person为基类派生Teacher类，分别创建Person类和Teacher类的对象，
    并在派生类对象中调用基类方法。'''


class Person:
    def __init__(self, name='', age=20, sex='man'):
        self.setName(name)
        self.setAge(age)
        self.setSex(sex)

    def setName(self, name):
        if not isinstance(name, str):
            print('name must be string.')
            return
        self.__name = name

    def setAge(self, age):
        if not isinstance(age, int):
            print('name must be integer.')
            return
        self.__age = age
```

```
    def setSex(self, sex):
        if sex != 'man' and sex != 'woman':
            print('sex must be "man" or "woman"')
            return
        self.__sex = sex

    def show(self):
        print('Name', self.__name)
        print('Age', self.__age)
        print('Sex', self.__sex)


class Teacher(Person):  # 派生类
    def __init__(self, name="", age=30, sex='man', department='Computer'):
        super(Teacher, self).__init__(name, age, sex)
        # #or, use another method like below:
        # Person.__init__(self, name, age, sex)
        self.setDepartment(department)

    def setDepartment(self, department):
        if not isinstance(department, str):
            print('department must be a string.')
            return
        self.__department = department

    def show(self):
        # Person.show()
        super(Teacher, self).show()
        print('Department:', self.__department)


if __name__ == '__main__':
    zhangsan = Person('Zhang San', 19, 'man')
    zhangsan.show()
    lisi = Teacher('Li Si', 32, 'man', 'Math')
    lisi.show()
    lisi.setAge(40) # 派生类调用基类的方法
    lisi.show()
```

更好的理解Python类的继承机制

```
class A(object):
    def __init__(self):
        self.__private()
        self.public()

    def __private(self):
        print('__private() method in A')

    def public(self):
        print('public() method in A')
```

```
class B(A):  # 注意，类B没有定义构造函数

    def __private(self):
        print('__private() method in B')

    def public(self):
        print('public() method in B')


b = B()
# __private() method in A # 调用了A类的私有函数的构造函数，不能访问派生类B的私有函数
# public() method in B    # 调用了A类的共有成员的构造函数
print(dir(b))


# ['_A__private', '_B__private', '__class__'...]

class C(A):
    def __init__(self):  # 显示定义构造函数
        self.__private()
        self.public()

    def __private(self):
        print('__private() method in C')

    def public(self):
        print('public() method in C')


c = C()
# __private() method in C   # 调用了C类的构造方法，由于被覆写
# public() method in C
print(dir(c))
# ['_A__private', '_C__private', '__class__', ...]
```

- 课外思考

```
class D:
    attr = 3

class B(D):
    pass

class E:
    attr = 2

class C(E):
    attr = 1

class A(B,C):
    pass

X = A()
```

```
print(X.attr)
# 3

class D:
    attr = 3

class B(D):
    pass

class E:
    attr = 2

class C(E):
    attr = 1

class A(B,C):
    pass

X = A()
print(X.attr)
# 1
```

- 类组合的使用
- 使用super()方法的作：
    - 如果对于基类进行增删，使用super()时，不需要对派生类进行修改。
    - 如果只是再派生类自己定义，则对于基类修改的时候，需要对派生类进行手动修改。

第9章 GUI编程

9.1 wxPython

创建GUI程序的三个主要步骤：

- 导入wxPython包
- 建立框架类：框架类父类为wx.Frame
- 建立主程序：
    - 创建应用程序对象
    - 创建框架类对象
    - 显示框架
    - 开始事件循环
- 执行frame.Show(True)
- 执行app.MainLoop(),框架处理事件

9.1.1 Frame

- Frame也成为框架或窗体，是所有框体的父类
- 创建GUI程序框架时，需要继承wx.Frame派生出子类，在派生类中调用基类构造数进行必要的初始化
- 它的构造函数格式为

```
__init__(self, Windoww parent, int id=-1, String title = EmptyString, Point
pos = DefaultPosition, Size size = DefaultSize, long style =
DEFAULT_FRAME_STYLE, String name = FrameNameStr)
```

```
class MyFrame(wx.Frame):
    def __init__(self, superior):
        # 调用了父类的Frame函数
        wx.Frame.__init__(self, parent=superior, title=u'My First Form', size=
(300, 300))
        self.Bind(wx.EVT_SIZE, self.OnSize)
        self.Bind(wx.EVT_MOVE, self.OnFrameMove)

        panel = wx.Panel(self, -1)
        label1 = wx.StaticText(panel, -1, "FrameSize:")
        label2 = wx.StaticText(panel, -1, "FramePos:")
        label3 = wx.StaticText(parent=panel, label='MousePos:')
        self.sizeFrame = wx.TextCtrl(panel, -1, '', style=wx.TE_READONLY)
        self.posFrame = wx.TextCtrl(panel, -1, '', style=wx.TE_READONLY)
        self.posMouse = wx.TextCtrl(panel, -1, '', style=wx.TE_READONLY)
        panel.Bind(wx.EVT_MOTION, self.OnMouseMove)
        self.panel = panel

        # Use some sizers for layout of the widgets
        sizer = wx.FlexGridSizer(3, 2, 5, 5)
        sizer.Add(label1)
        sizer.Add(self.sizeFrame)
        sizer.Add(label2)
        sizer.Add(self.posFrame)
        sizer.Add(label3)
        sizer.Add(self.posMouse)

        border = wx.BoxSizer()
        border.Add(sizer, 0, wx.ALL, 15)
        panel.SetSizerAndFit(border)
        self.Fit()

    def OnSize(self, event):
            size = event.GetSize()
            self.sizeFrame.SetValue("%s, %s" % (size.width, size.height))
            # tell the event system to continue looking for an event handler,
            # so the default handler will get called.
            event.Skip()

    def OnFrameMove(self, event):
            pos = event.GetPosition()
            self.posFrame.SetValue('%s, %s' % (pos.x, pos.y))

    def OnMouseMove(self, event):
            pos = event.GetPosition()
            self.posMouse.SetValue('%s, %s' % (pos.x, pos.y))
```

```
if __name__ == '__main__':
    app = wx.App()  # Create an instance of the application class
    frame = MyFrame(None)
    frame.Show(True)
    app.MainLoop()  # Tell it to start processing events
```