

ch1

python语言特点：**简单、高级、面向对象、可扩展性**

ch2

python3中，**一切皆为对象**

- 标识(identity) 用于唯一标识一个对象，对应于对象在计算机内存中的位置。应用: id(obj)
- 类型(type) 用于表示对象所属的数据类型。应用: type(obj)
- 值(value) 用于表示对象的数据类型的值。应用: print(obj)

==运算符判断两个变量指向的对象的值是否相同

is运算符判断两个变量是否指向同一对象

不可变对象：一旦创建，其值不能修改。如int、str、complex等

可变对象：列表、字典等

标识符是变量、函数、类、模块和其他对象的名称。第一个字符必须是**字符、下划线**。

- 语句是python程序的过程构造块，用于**定义函数、定义类、创建对象、变量赋值、调用函数、控制分支、创建循环**等
- python语句分为**简单语句**和**复合语句**
简单语句：**表达式语句、赋值语句、assert语句、pass空语句、del语句、return语句、yield语句、raise语句、break语句、continue语句、import语句、global语句、nonlocal语句**等
复合语句：**if语句、while语句、for语句、try语句、with语句、函数定义、类定义**等

反斜杠 (\) 用于一个代码跨越多行的情况。如果语句太长，可以使用续行符 (\)

分号 (;) 用于在一行书写多条语句(每行结尾加;也可以)

ch3

可迭代对象 是指可以一次返回其中的一个成员的对象。可迭代对象包括所有序列类型（如str、tuple、bytes、list、bytearray等）和一些非序列类型（如range、dict、set、frozenset、file、用户定义的一些对象等）。可迭代对象通过内置iter()函数用来生成迭代器（iterator）。迭代器可以被next()函数调用,并不断返回下一个值（从而避免一次把整个数据调入内存）。同时，Iterator自己也是一种Iterable。

```
str.format('{0:1} * {1:1} = {2:<2} ', i, j, i*j)
# 第一个0, 1, 2指的是第几个参数，第二个1,1,<2指的是长度分别为1, 1, 小于2
```

```
for i in range(1,10):
    s = ''
    for j in range(10-i+1, 10):
        s += ' '
    for j in range(i,10):
        s += str.format('{0:1}*{1:1}={2:<2} ', i, j, i*j)
    print(s)
```

效果：

```
===== RESTART: C:/Users/Administrator/Desktop/test.py =====
1*1=1  1*2=2  1*3=3  1*4=4  1*5=5  1*6=6  1*7=7  1*8=8  1*9=9
      2*2=4  2*3=6  2*4=8  2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
            3*3=9  3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
                  4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
                        5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
                              6*6=36 6*7=42 6*8=48 6*9=54
                                    7*7=49 7*8=56 7*9=63
                                          8*8=64 8*9=72
                                              9*9=81

>>> !
```

enumerate() 函数用于将一个可遍历的数据对象组合为一个索引序列，并返回一个可迭代对象

```
>>> lst=[5,6,7,8,9, 10]
>>> en=enumerate(lst, start=1)
>>> print(en)
<enumerate object at 0x0000000002E3CF78>
>>> it=iter(en)
>>> next(it)
(1, 5)
>>> next(it)
(2, 6)
>>> next(it)
(3, 7)
```

```
seasons = ['Spring', 'Summer', 'Autumn', 'Winter']
for i, s in enumerate(seasons, start=1): #start默认从0开始
    print(str.format("第{0}季节: {1}".format(i, s)))
```

```
第1季节: Spring
第2季节: Summer
第3季节: Autumn
第4季节: Winter
>>>
```

zip()函数将多个可迭代的对象中对应的元素打包成一个个元组，然后返回一个可迭代的对象。如果元素个数不一致，则返回列表长度与最短的对象相同。利用运算符*还可以实现将列表解压为元组

```
evens = [0, 2, 4, 6, 8]
odds = [1, 3, 5, 7, 9]
for e, o in zip(evens, odds):
    print("{0}*{1}={2}".format(e, o, e*o))
    print(str.format("{0}*{1}={2}", e, o, e*o))
```

```
-----
0*1=0
0*1=0
2*3=6
2*3=6
4*5=20
4*5=20
6*7=42
6*7=42
8*9=72
8*9=72
>>> |
```

map(func,seq)函数将函数func作用于可迭代对象seq中的每一个元素，并将所有的调用结果作为可迭代对象返回

```
>>> m = map(len, ('spring', 'summer', 'fall', 'winter'))
>>> it = iter(m)
>>> next(it)
6
>>> next(it)
6
>>> next(it)
4
```

ch4

内置函数ord()可以把字符转换为对应的Unicode码；chr()可以把十进制数转换为对应的字符

字符串的格式化：

- 格式字符串.format(值 1, 值 2, ...)
- str.format(格式字符串, 值 1, 值 2, ...)
- format(值, 格式字符串)
- 格式字符串 % (值 1, 值 2, ...) # 兼容 Python 2 的格式, 不建议使用

```
>>> "学生人数{0}, 平均成绩{1}".format(15, 81.2)
'学生人数15, 平均成绩81.2'
>>> str.format("学生人数{0}, 平均成绩{1:2.2f}", 15, 81.2)
'学生人数15, 平均成绩81.20'
>>> format(81.2, "0.5f")          # 输出: '81.20000'
>>> "学生人数%d, 平均成绩%2.1f" % (15, 81)
'学生人数 15, 平均成绩81.0'
```

格式化字符串变量：

以f开始的字符串中可以包含嵌入在花括号{}中的变量，称之为字符串变量替换（插值）

```
>>> name = "Fred"
>>> f"He said his name is {name}."
'He said his name is Fred.'
>>> score, width, precision = 12.34567, 10, 4
>>> f"result: {score:{width}.{precision}}"
'result:      12.35'
>>> f"result: {88.58885:{8}.{4}}"
'result:      88.59' # 8表示左对齐，总共8长度。4表示一共4位
```

ch5

- python 内置的序列数据类型
 - 元组 (tuple) 、列表 (list) 、字符串 (str) 和字节数据 (bytes和bytearray)
- 元组**也称之为定值表，用于存储值固定不变的值表。
- 列表**也称之为表，用于存储其值可变的表。

可以通过 `in` 或者 `not in` 判断一个元素是否存在于序列中

```
# 系列的排序操作
sorted(iterable, key=None, reverse=False)
# iterable-可迭代对象，即要排序的对象
# key- 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。 比如，lambda x: x['age']
# reverse- 排序规则，reverse = True 降序，reverse = False 升序（默认）
```

内置函数all()和any()

```
all(iterable)    #如果序列的所有值都为True，返回True；否则，返回False
any(iterable)    #如果序列的任意值为True，返回True；否则，返回False
```

系列拆封

变量 1, 变量 2, ..., 变量 n = 系列或可迭代对象

变量个数和系列长度可以不等

```
# 可使用*元组变量，将多个值作为元组赋值给元组变量
>>> first, *middles, last = range(10)
>>> middles #输出: [1, 2, 3, 4, 5, 6, 7, 8]
>>> first, second, third, *lasts = range(10)
>>> lasts #输出: [3, 4, 5, 6, 7, 8, 9]
>>> *firsts, last3, last2, last1 = range(10)
>>> firsts #输出: [0, 1, 2, 3, 4, 5, 6]
>>> first, *middles, last = sorted([70, 85, 89, 88, 86, 95, 89]) #去掉最高分和最低分
>>> sum(middles)/len(middles) #计算去掉最高分和最低分后的平均值。输出: 87.4

# 使用临时变量
>>> _, b, _ = (1, 2, 3)
>>> b #输出: 2
```

```
>>> record = ('Zhangsan', 'szhang@abc.com', '021-622333', '13912349876')
>>> name, _, *phones = record
>>> phones      #输出: ['021-62232333', '13912349876']
```

- python中没有数组，可以用列表来代替

列表解析表达式

- `[expr for i1 in 序列1... for i2 in 序列N]` #迭代序列里所有内容，并计算生成列表
- `[expr for i1 in 序列1... for i2 in 序列N if cond expr]` #按条件迭代，并计算生成列表

```
>>> [i**2 for i in range(10)]      #平方值
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [(i,i**2) for i in range(10)]  #序号，平方值
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81)]
>>> [i for i in range(10) if i%2==0] #取偶数
[0, 2, 4, 6, 8]
>>> [(x, y, x*y) for x in range(1, 4) for y in range(1, 4) if x>=y] #二重循环
[(1, 1, 1), (2, 1, 2), (2, 2, 4), (3, 1, 3), (3, 2, 6), (3, 3, 9)]
```

字符串编码

`s.encode(encoding="utf-8", errors="strict")` #把字符串对象s 编码为字节码对象

`b1.decode(encoding="utf-8", errors="strick")` #把字节码对象b 解码为对应编码的字符串

```
>>> s1='百度ABC'
>>> b1=s1.encode(encoding="utf-8", errors="strict")
>>> b1
b'\xe7\x99\xbe\xe5\xba\xa6ABC'
>>> s2=b1.decode(encoding="utf-8", errors="strick")
>>> s3=str(object = b1,encoding="utf-8", errors="strick" )
>>> print(s3,s4)
百度ABC 百度ABC
```

字符串格式化---%运算符形式

```
>>> print('结果: %f' % 88)
结果: 88.000000

>>>print('姓名: %s, 年龄: %d, 体重: %3.2f' % ('张三', 20, 53))
姓名: 张三, 年龄: 20, 体重: 53.00

>>> print('%(lang)s has %(num)03d quote types.' % {'lang':'Python', 'num': 2})
Python has 002 quote types.

>>> print('%0*.*f' % (10, 5, 88))      #10: 占位宽度; 5: 小数点位数
'0088.00000'
```

字符串格式化---format内置函数

● `format(value)`

#等同于 `str(value)`

● `format(value, format_spec)`

```
>>> print(format(81.2, "0.5f"))
81.20000
>>> print(format(81.2, ".3%"))
8120.000%
>>> print(format(1000000.9, "_.5f"))
1_000_000.90000
>>> print(format(1024, "_b") )
100_0000_0000
>>> print(81.2.__format__("0.5f"))      #仅限浮点形式
81.20000
```

● 格式字符串 `format(值 1, 值 2,...)` #对象方法

● `str.format(格式字符串, 值 1, 值 2,...)` ... #类方法

```
>>> print('{0} {1} {0}'.format('hello', 'world'))      #索引
hello world hello
>>> print('{a} {tom} {a}'.format(tom='hello', a='world')) #键
world hello world
>>> print("用PI={0:.2f}, 求{1}面积".format(3.1415926, "圆"))
用PI=3.14, 求圆面积
>>> print("PI={0:>20.2f}".format(3.1415926))      #宽度20右对齐
PI=
          3.14
>>> print("PI={0:>20.2%}".format(3.1415926))      # 百分比格式
PI=
          314.16%
>>> print("PI={0:<20,.3f}".format(100003.1415926))  #带千位分隔符
PI=100,003.142
print("PI={0:<20_.3f}".format(100003.1415926))  #带千位分隔符
PI=100_003.142
>>> print("结果是{:.2e}".format(1000000000))      #指数形式
结果是1.00e+09
>>> print("Tom今年{0:<d}岁, {1:<.2f}公斤".format(18, 56))  #十进制
Tom今年18岁, 56.00公斤
>>> print("Tom今年{0:<b}岁, {1:<.2f}公斤".format(18, 56))  #二进制
Tom今年10010岁, 56.00公斤
```

```
>>> print(str.format('{0} {1},{0}','hello','world',))      #索引
hello world hello
>>> print(str.format('{a} {tom} {a}', a='hello', tom='world')) #键
hello world hello
>>> print(str.format("用PI={0:.2f}, 求{1}面积", 3.1415926, "圆"))
用PI=3.14, 求圆面积
>>> print(str.format("PI={0:>20.2%}", 3.1415926))      # 百分比格式
PI=
          314.16%
>> print(str.format("结果是{:.2e}", 1000000000))      #指数形式
结果是1.00e+09
>>> print(str.format("Tom今年{0:<d}岁, {1:<.2f}公斤", 18, 56))  #十进制
```

Tom今年18岁，56.00公斤

```
>>> print(str.format("Tom今年{0:<b}岁，{1:<.2f}公斤", 18, 56)) #二进制
```

Tom今年10010岁，56.00公斤

字符串格式化---format_map()方法

格式字符串.format_map(mapping)

```
>>> People = {"name": "john", "age": 33}
>>> print("My name is {name}, i am {age} old".format_map(People))
My name is john, i am 33 old
```

bytes常量

使用字母b加单引号或双引号括起来的内容

```
>>> b'abc\x\'
b'abc\x'
>>> bytes((1,2,3))
b'\x01\x02\x03'
```

bytearray对象

bytearray是可变的bytes数据类型，可以通过bytearray创建和定义。

bytes和bytearray的方法不接受字符串参数，只接受bytes和bytearray参数

```
>>> b1=b"abc"
>>> b1.replace(b'a',b'f') #输出: b'fbc'
b'fbc'
>>> b1.replace('b','g') # TypeError: a bytes-like object is required, not 'str'
```

ch6

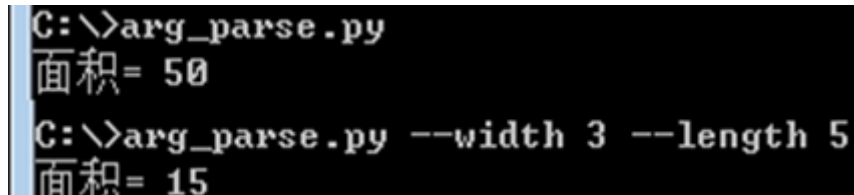
命令行参数

通过列表sys.argv访问命令行参数。argv[0]为Python脚本名，argv[1]为第1个参数，argv[2]为第2个参数

```
import sys, random
n = int(sys.argv[1])
for i in range(n):
    print(random.randrange(0,100))
```

argparse 是python自带的命令行参数解析包，可以用来方便地读取命令行参数，当你的代码需要频繁地修改参数的时候，使用这个工具可以将参数和代码分离开来，让你的代码更简洁，适用范围更广。

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--length', default=10, type=int, help='长度')
parser.add_argument('--width', default=5, type=int, help='宽度')
args = parser.parse_args()
area = args.length * args.width
print('面积=', area)
```



```
C:\>arg_parse.py
面积= 50
C:\>arg_parse.py --width 3 --length 5
面积= 15
```

文件和文件对象

`f=open(file, mode='r', buffering=-1, encoding=None)`

使用open()函数时，可以指定打开文件的模式mode为：‘r’（只读）、‘w’（写入，写入前删除旧内容）、‘x’（创建新文件，如果文件存在，则导致FileExistsError）、‘a’（追加）、‘b’（二进制文件）、‘t’（文本文件，默认值）、‘+’（更新，读写）。

Buffering=-1时，使用系统默认缓冲区大小

with语句和上下文管理协议

with context [as var]

- 实现上下文管理协议的对象

操作语句

- 文件对象支持with语句，确保打开的文件自动关闭

with open(file, mode) as f:

#操作打开的文件

标准输入、输出和错误流重定向

可以使用sys.stdout、sys.stdin、和sys.stderr实现标准输出流、标准输入流和标准错误流 重定向，例如重定向标准输出流至“out.log”文件（注意重定向后的恢复）

例子：从命令行第一个参数中获取n的值，然后将 0_n 以及2的 0_n 次幂的列表打印输出到out.log文件中


```
import sys
n = int(sys.argv[1])    #从命令行第1个参数中获取n
power = 1               #2的0~n次幂赋初值
i = 0                   #计数赋初值
f = open('out.log', 'w') #指定标准输出重定向到文件
sys.stdout = f
while i <= n:
    print(str(i), ' ', str(power)) #输出i以及2的i次幂
    power = 2 * power             #计算2的i次幂
    i = i + 1                     #计数加1
sys.stdout = sys.__stdout__
print('done!')
f.close()
```

重定向和管道

- 重定向标准输出到一个文件：

程序 > 输出文件

- 重定向标准输入到一个文件：

程序 < 输入文件

例子：重新定向标准输出到一个文件

- 命令行命令：python randomseq.py 10 > scores.txt
- 命令行命令：type scores.txt

管道：将一个程序的标准输出与另一个程序的标准输入相连，这种机制称之为管道

c:\pythonpa\ch06> python randomseq.py 1000 | python average.py

****其执行结果等同于下列两行执行命令：**

c:\pythonpa\ch06> python randomseq.py 1000 > scores.txt

c:\pythonpa\ch06> python average.py < scores.txt

filter函数 实现过滤

filter函数接收一个函数和一个序列，他把传入的函数依次作用于每个元素，然后根据返回值的真假来确定保留还是丢弃

```
def is_odd(n):
    return n % 2 == 1
temlist = filter(is_odd, [1,2,3,4,5,6,7,8,9,10])
newlist = list(temlist)
print(newlist) #将不能被2整除的函数输出
```

ch7

语法错误

语法错误是指其源代码中拼写语法错误，这些错误导致Python编译器无法把Python源代码转换为字节码，故也称之为编译错误。当程序包含语法错误时，编译器将显示SyntaxError

异常处理

```
try:
    print(3/0)
except ZeroDivisionError:
    print("除数不能为零")
```

使用try...except...else...finally语句捕获处理异常

```
try:
    可能产生异常的语句
except Exception1:
    #捕获异常 Exception1
    发生异常时执行的语句
except (Exception2, Exception3):
    #捕获异常 Exception2、Exception3
    发生异常时执行的语句
except Exception4 as e:
    #捕获异常 Exception4, 其实例为 e
    发生异常时执行的语句
except:
    #捕获其它所有异常
    发生异常时执行的语句
else:
    #无异常
    无异常时执行的语句
finally:
    #不管发生异常与否, 保证执行
    不管发生异常与否, 保证执行的语句
```

捕获异常的顺序

- except块可以捕获并处理特定的异常类型（此类型称为“异常筛选器”），具有不同异常筛选器的多个except块可以串联在一起。系统自动由上向下匹配发生的异常：如果匹配（引发的异常为“异常筛选器”的类型或子类型），则执行该except块中的异常处理代码，否则继续匹配下一个except块。故用户需要将带有最具体的（即派生程度最高的）异常类的except块放到最前面。
- finally 块始终在执行完try和except块之后执行，而与是否发生异常与否无关。

自定义异常类

- 异常应该是典型的继承自Exception类，通过直接或间接的方式。自定义异常类的命名规则一般以Error或Exception为后缀。
- 创建了一个类，基类为RuntimeError，用于在异常触发时输出更多的信息。在try语句块中，用户自定义的异常后执行except块语句，变量 e 是用于创建Networkerror类的实例。

```
class Networkerror(RuntimeError):
    def __init__(self, arg):
        self.args = arg
try:
    raise Networkerror("Bad hostname")
except Networkerror as e:
    print (''.join(e.args))
```

断言处理

断言一般用于以下情况：

- (1) 前置条件断言：代码执行之前必须具备的特性
- (2) 后置条件断言：代码执行之后必须具备的特性
- (3) 前后不变断言：代码执行前后不能变化的特性

```
a = int(input("请输入整数a: "))
b = int(input("请输入整数b: "))
assert b != 0, '除数不能为0'
c = a / b
print(a, '/', b, '=', c)
```

ch8

函数的功能

- 实现结构化程序设计
- 减少程序的复杂度
- 实现代码复用
- 提高代码的质量
- 协作开发
- 实现特殊功能

python函数分类

- 内置函数
- 标准库函数
- 第三方库函数
- 用户自定义函数

形式参数和实际参数

- 声明函数时所声明的参数，为形参
- 调用函数时，提供函数所需要的参数的值，为实参

python参数传递方法时传递对象引用，而不是传递对象的值

可选参数

- 在声明函数时，如果希望函数的一些参数是可选的，可以在声明函数时为这些参数指定默认值（和C++相同）。必选参数在前,默认参数在后。
- 调用该函数时，如果没有传入对应的实参值，则函数使用声明时指定的默认参数值。

位置参数

函数调用时，实参默认按位置顺序传递形参。按位置传递的参数称之为位置参数（比如C++函数）

命名参数

按名称指定传入的参数称为命名参数，也称之为关键字参数。使用关键字参数具有三个优点：参数按名称意义明确；传递的参数与顺序无关；如果有多个可选参数，则可以选择指定某个参数值

可变参数

- 在声明函数时，通过带星的参数，如*param1，允许向函数传递可变数量的实参。调用函数时，从那一点后所有的参数被收集为一个元组。
- 在声明函数时，也可以通过带双星的参数，如**param2，允许向函数传递可变数量的实参。调用函数时，从那一点后所有的参数被收集为一个字典。
- 带星或双星的参数一般位于形参列表的最后位置。（在带星号的参数后面声明的参数强制为命名参数）

强制命名参数

在带星号的参数后面申明参数会导致强制命名参数（Keyword-only），调用时必须显式使用命名参数传递值。

非局部语句nonlocal

在函数体中，可以定义嵌套函数，在嵌套函数中，如果要为定义在上级函数体的局部变量赋值，可以使用nonlocal语句，表明变量不是所在块的局部变量，而是在上级函数体中定义的局部变量。nonlocal语句可以指定多个非局部变量。例如nonlocal x, y, z

eval函数（动态表达式求值）

```
eval(expression, global=None, local=None)
```

expression是动态表达式的字符串；globals和locals是求值时使用的上下文环境的全局变量和局部变量，如果不指定，则使用当前运行上下文

exec函数（动态语句的执行）

```
exec(str[, global[, locals]])
```

str是动态语句的字符串；globals和locals是使用的上下文环境的全局变量和局部变量，如果不指定，则使用当前运行上下文

- 通常，eval用于动态表达式求值，返回一个值；exec用于动态语句的执行，不返回值

compile函数（动态语句的执行）

```
compile(source, filename, mode) # 返回代码对象
```

compile() 函数将一个字符串编译为字节代码；如果是多行语句，则每一行的结尾必须有换行符\n。filename为包含代码的文件；mode为编译模式，可以为：'exec'（用于语句系列执行）、'eval'（用于表达式求值）和'single'（用于单个交互语句）。

```
>>> co = compile("for i in range(10): print(i, end=' ')", '', 'exec')
>>> exec(co)          #输出: 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

内置map()函数

map()函数基本形式：map(f, iterable, ...)。将函数f应用于可迭代对象，返回结果为可迭代对象

```
>>> def is_odd(x):
        return x % 2 == 1
>>> list(map(is_odd, range(5)))
[False, True, False, True, False]
```

内置filter()函数

filter()函数实现为内置的filter(f, iterable)可迭代对象，将函数f应用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素，返回结果为可迭代对象

```
>>> def is_odd(x):
        return x % 2 == 1
>>> list(filter(is_odd, range(10)))    #输出: [1, 3, 5, 7, 9]
```

Lambda表达式和匿名函数

lambda是一种简便的、在同一行中定义函数的方法；lambda实际上生成一个函数对象，即匿名函数。lambda实际上生成一个函数对象，即**匿名函数**

匿名函数示例：

```
>>> f = lambda x, y: x + y
>>> type(f)    #输出: <class 'function'><class 'function'>
>>> f(12, 34)    #计算两数之和。输出: 46
```

operator模块和操作符函数

operator模块是Python内置操作符的函数接口，它定义了对应算术和比较等操作的函数。可以用于map()、filter()等需要传递函数对象作为参数的场合，可以直接使用而不需要使用函数定义或者lambda表达式，使得代码更加简洁

```
>>> import operator
>>> a = 'hello'
>>> operator.concat(a, ' world')
'hello world'
```

```
>>> import operator
>>> list(map(operator.gt, [1,5,7,3,9], [2,8,4,6,0]))
[False, False, True, False, True]
```

ch9

声明类的过程也就是抽象和封装的过程

- 继承：在已有类的基础上创建新类，这其中的一种做法就是让一个类从另一个类那里将属性和方法直接继承下来，从而减少重复代码的编写。
- 多态：子类在继承了父类的属方法后，通过方法重写我们可以让父类的同一个行为在子类中拥有不同的实现版本，进而由继承产生的不同的对象对同一消息会作出不同的响应，分别执行不同的操作。

属性和方法

- 属性：类中的变量。类中的变量或者说类中的属性又包括类属性和实例属性（对象属性），私有属性和公有属，特殊属性和自定义属性等。
- 方法：在类内定义的函数。第一个参数是self的方法称为对象实例方法否则称为类方法。

@property装饰器默认提供一个只读属性，如果需要可以使用对应的getter、setter和deleter装饰器实现其他访问属性函数

```
# 调用格式：
property(fget=None, fset=None, fdel=None, doc=None)

class Person13:
    def __init__(self, name):
        self.__name = name
    def getname(self):
        return self.__name
    def setname(self, value):
        self.__name = value
    def delname(self):
        del self.__name
    name = property(getname, setname, delname, "I'm the 'name' property.")

#测试代码
p = Person13('爱丽丝');
print(p.name)
p.name = '罗伯特';
print(p.name)
```

对象方法

方法的声明格式如下：

def 方法名(self,[形参列表]):
函数体

方法的调用格式如下：

对象.方法名([实参列表])

方法重载

所谓重载，是指可以定义多个重名的方法，只要保证方法签名是唯一的；方法签名包括三个部分：方法名、参数数量和参数类型。

Python不支持重载， 但可以用默认参数和可变参数实现多种实际参数传递。

继承

派生类：python支持多重继承，即一个派生类可以继承于多个基类

```
class Person:
    #基类
    def __init__(self, name, age): #构造函数
        self.name = name
        self.age = age
    def say_hi(self):
        #定义基类方法say_hi
        print('我叫{0}, {1}岁'.format(self.name, self.age))
```

```
class Student(Person):          #派生类
    def __init__(self, name, age, stu_id): #构造函数
        Person.__init__(self, name, age) #调用基类构造函数
        self.stu_id = stu_id          #学号
    def say_hi(self):              #定义派生类方法say_hi
        Person.say_hi(self)         #调用基类方法say_hi
        print('我是学生, 我的学号为: ', self.stu_id)

p1 = Person('张王一', 33)          #创建对象
p1.say_hi()

s1 = Student('李姚二', 20, '2018101001') #创建对象
s1.say_hi()
```