第3章 Python的基本类型

3-2 数字：整型与浮点型

- 整数：int
- 浮点数：float
- 布尔：bool

> **注**：bool是Number类型

`type()`：可以判断类型

- `type(1 * 1)`：`<class 'int'>`
- `type(1 * 1.0)`：`<class 'float'>`
- `type(2 / 2)`：`<class 'float'>`
- `type(2 // 2)`：`<class 'int'>`

  `2 / 2`：`1.0`

  `1 // 2`：`0`

  `int(True)`：`0`

3-4 各进制的表示与转换

表示：

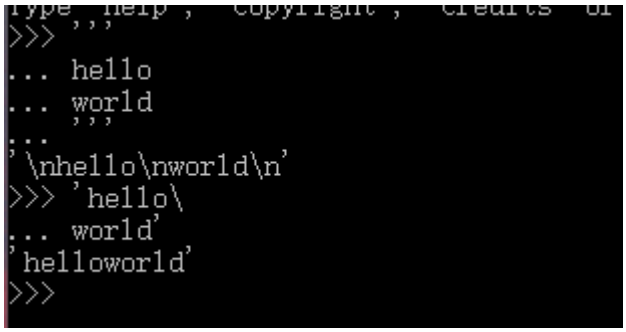
- 二进制
  - `0b10`：2
- 八进制
  - `0o11`：9
- 十六进制
  - `0x1F`：31

转换：

- 二进制
  - `bin(10)`：`'0b1010'`
  - `bin(0xE)`：`'0b1110'`
- 十进制
  - `int(0b111)`：`7`
- 八进制
  - `hex(0o7777)`：`'0xfff'`
- 十六进制
  - `oct(0b111)`：`'0o7'`

3-6 字符串

可以用单引号、双引号、三引号表示字符串



3-9 原始字符串

`print(r'c:\northwind\northwest')`：字符串前面加r，不是一个普通字符串，而是原始字符串（按字符串的样式输出，不会把`\n`当作换行）。

3-10 字符串运算 一

字符串的运算

- 字符串合并
  - `'hello' + 'world'`
  - `'hello' * 3`：`'hellohellohello'`，只能和`int`相乘
- 字符串索引
  - `'hello'[0]`
  - `'hello world'[-1]`：`'d'`，[-n]从字符串的末尾往前数n个
- 字符串截取
  - `'hello world'[0:4]`：`'hell'`，包前不包后
  - `'hello wolrd'[0:-1]`：`'hello worl'`
  - `'hello wolrd'[6:20]`：`'world'`
  - `'hello world'[6:]`：`'world'`

第4章 Python中表示“组”的概念与定义

4-1 列表的定义

`type([1, 2, 3, 4])`：`<class 'list'>`

4-2 列表的基本操作

- 增
  - `["新月打击", "苍白之瀑", "月之降临", "月神冲刺"] + ["点燃", "闪现"]`：`["新月打击", "苍白之瀑", "月之降临", "月神冲刺", "点燃", "闪现"]`
  - `["点燃", "闪现"] * 3`：`["点燃", "闪现", "点燃", "闪现", "点燃", "闪现"]`

- 查
  - `["新月打击", "苍白之瀑", "月之降临", "月神冲刺"][1]`：`"苍白之瀑"`
  - `["新月打击", "苍白之瀑", "月之降临", "月神冲刺"][-1:]`：`["月神冲刺"]`
  - 和字符串的截取基本上一样

空的列表：

- `type([])`：`<class 'list'>`
- `type(list())`：`<class 'list'>`

4-3 元组

`type((1, 2, 3, 4))`：`<class 'tuple'>`

访问方式和列表及字符串一样

- `(1, 2, 3) + (4, 5)`：`(1, 2, 3, ,4, 5)`
- `(1, 2, 3) * 2`：`(1, 2, 3, 1, 2, 3)`
- `type((1))`：`<class 'int'>`
  - 这因为`()`在python中既可以表示元组，也可以标识`(1 + 1)`这种运算，所以python硬性规定`()`中只有一个字符的话标识数序运算
- `type((1,))`：`<class 'tuple'>`

字符串、列表、元组都是序列

序列的特点：

- 序号：`[1, 2, 3][2]`
- 切片：`'hello world'[0:2:]`

序列的操作：

- 包含：`3 in [1, 2, 3, 4]`，`'hello' in 'hello world'`
- 长度：`len([1, 2, 3, 4])`，`len('hello world')`
- 最大最小：`max([1, 2, 3, 5])`，`min([2, 1, 5])`，`max('hello world') == 'w'`，`min('hello world') == ' '`

空的元组：

- `type(())`：`<class 'tuple'>`
- `type(tuple())`：`<class 'tuple'>`

4-5 set集合

集合set特点：

- 无序
- 不能重复

`type({1, 2, 3})`：`<class 'set'>`

因为无序，所以集合不支持序号和切片

set的操作：

- 长度：`len({1, 2, 3})`
- 包含：`1 not in {1, 2, 3} == False`
- 两个集合的差集：`{1, 2, 3, 4, 5, 6} - {3, 4} == {1, 2, 5, 6}`
- 两个集合的交集：`{1, 2, 3, 4, 5, 6} & {3, 4} == {3, 4}`
- 两个集合的并集：`{1, 2, 3, 4, ,5, 6} | {3, 4, 7} == {1, 2, 3, ,4, 5, 6, 7}`

空的集合：

- `type(set())`：`<class 'set'>`

4-6 dict字典

很多个key和value

`{'Q': '新月打击', 'W': '苍白之瀑', 'E': '月之降临', 'R': '月神冲刺'}`

字典中的value的范围：

- python中的任何字典

字典中的key的范围：

- 必须是不可变的类型：int，str，元组

空的字典：

- `type({})`：`<class 'dict'>`
- `type(dict())`：`<class 'dict'>`

4-7 基本数据类型总结

第5章 变量与运算符

5-1 什么是变量

```
a, b = [1, 2, 3], [5, 6]
a * 2 + b + a       # [1, 2, 3, 1, 2, 3, 5, 6, 1, 2, 3]
```

5-3 值类型与引用类型

int：值类型

list：引用类型

```
a = 1           # a指向1
b = a           # b指向1
a = 3           # a指向3，此时b还是指向1
print(b)        # 1

a = [1, 2, 3]    # a指向[1, 2, 3]
b = a           # b指向[1, 2, 3]
a[0] = '1'       # a指向的[1, 2, 3]中的第1位变成'1'，此时a的指向没有变，但是a指向的值发生了变化，所以b指向的值也会变化
print(b)        # ['1', 2, 3]
```

引用类型是可变的，但是值类型是不可变的

值类型：不可变

- int
- str
- tuple

引用类型：可变

- list
- set
- dict

```
a = 1
print(id(a))        # 1822974112
a = 2
print(id(a))        # 1822974144
a = [1, 2, 3]
print(hex(id(a))        # '0x241e207c608'
a[0] = '1'
print(hex(id(a))        # '0x241e207c608'

a = 'hello world'
a[0] = 'p'          # 报错，因为字符串是不可变类型
a = (1, 2, 3)
a[0] = '1'          # 报错
```

5-4 列表的可变与元组的不可变

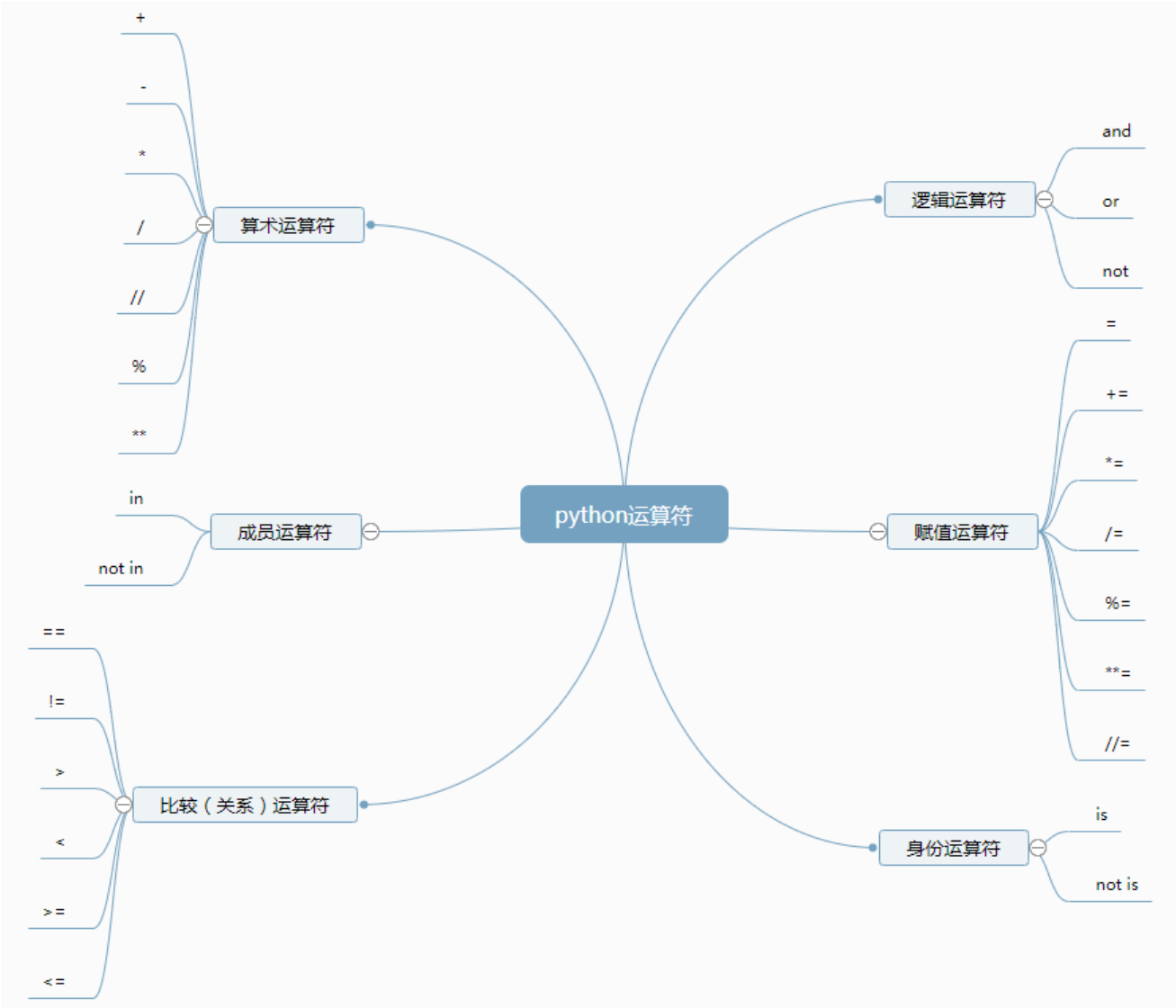
```
a = [1, 2, 3]
print(hex(id(a)))        # '0x241e2076dc8'
a.append(4)
print(hex(id(a)))        # '0x241e2076dc8'
print(a)                # [1, 2, 3, 4]

b = (1, 2, 3)
```

```
b.append(4)             # 报错

c = (1, 2, 3)
print(hex(id(c)))       # '0x241e2055d80'
c = c * 2
print(hex(id(c)))       # '0x241e1e2c228'

d = (1, 2, 3, [1, 2, 4])
d[3][2] = 5
print(d)                # (1, 2, 3, [1, 2, 5])
```

注:

- `append()`方法是修改自身，不会创建新的
- `c * 2`，`a + [4]`、`c + (123)`是 创建新的

5-5 运算符号



5-11 身份运算符

```
a, b = 1, 1.0
print(a == b)           # True
print(a is b)           # False

a = {1, 2, 3}
b = {2, 1, 3}
print(a == b)           # True
print(a is b)           # False

c = (1, 2, 3)
d = (2, 1, 3)
print(c == d)           # False
print(c is d)           # False
```

`==`比较两个值是否相等，`is`比较的是两个变量的内存地址是否相等（`id(a) == id(b)`）

5-12 如何判断变量的值、身份与类型

对象的三个特征：

- 值
- 身份（地址）：`id()`
- 类型：`type()`，`isinstance(a, (str, float))`，`isinstance(a, str)`

判断对象类型推荐`isinstance()`，不推荐`type() == str`，因为`type`不能判断子类。

6-8 常量与Pylint的规范

1. 常量

- python中的常量并不是真的常量，因为python没有任何机制可以阻止修改常量的值。但是对于值不变的变量，规范的写法是全部大写。
- 变量应该位于函数或者类中，不要直接放在模块里，模块中应该是常量

```
PI = 3.1415926
```

2. 模块

python每一个文件的开头需要写模块的注释

```
'''
模块说明
'''
```

第7章 包、模块、函数与变量作用域

7-5 Python工程的组织结构

- 包
  - 模块
    - 类
      - 函数、变量

7-6 Python包与模块的名字

如果想让普通的文件夹变成包的话，文件夹下必须有一个`__init__.py`文件

`__init__py`这个模块的名字就是包名（文件夹名）

7-9 `__init__.py`的用法

当导入包时，假设包名为a，`import a`会运行`__init__.py`的内容。

假设包a下有两个模块，c7.py和c8.py，在`__init__.py`中有`__all__ = ['c7']`，那么当通过*来导入这个包时：`from a import *`，只能导入c7这个模块，不能导入c8，但是可以如果`from a import c7, c8`则可以导入。

7-10 包与模块的几个常见错误

1. 包和模块时不会被重复导入的
2. 避免循环导入
3. 一旦导入一个模块，就会去执行模块里的代码

7-11 模块内置变量

`__xxx__`是python内置变量，但是从本质上来说和我们自己定义的变量没有什么区别。

```
'''
this is __doc__
'''
a = 2
c = 3

infos = dir()
print(infos)
'''
['__annotations__', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'a', 'c']
'''
# __doc__：this is __doc__(模块注释)
```

7-12 入口文件和普通模块内置变量的区别

- c12.py
- t
  - c9.py

运行c12.py

```
'''
c9 doc
'''

print("name: " + __name__)          # 命名
print("package: " + __package__)
print("doc: " + __doc__)            # 模块注释
print("file: " + __file__)
```

```
'''
c12 doc
'''
import t.c9
'''
name: t.c9
package: t      相对于入口文件的包路径
doc:
c9 doc

file: E:\workspace\vscode-workspace\imooc\Python3入门进阶\chapter7\t\c9.py
'''

print('package: ' + (__package__ or '当前模块不属于任何包'))  # 入口文件没有顶级包
print('name: ' + __name__)          # name: __main__    因为c12此时是入口文件
print('doc: ' + __doc__)            # c12 doc
print('file: ' + __file__)          # file: .\c12.py    因为c12此时是入口文件
# __file__的取值和我们运行python命令的所在目录有关，因为是在c12.py所在目录中运行的，所以此时__file__ =
.\c12.py
```

7-13 `__name__`的经典应用

```
import sys

infos = dir(sys)        # 当要查看某个模块或者某个类下面的变量的时候，就可以使用dir()，这里查看sys模块
中的变量
print(infos)
```

```
python -m seven.c15
```

这样可以让c15.py当作模块来运行（前提是c15这个模块必须有个包，且在包外执行这条命令）

7-14 相对导入和绝对导入

顶级包：入口文件的同级目录为顶级包

- main.py

- package1
  - pakage11
    - m1.py

如上，顶级包为package1

绝对路径：从顶级包开始，

相对路径：.表示当前目录，..表示上级目录，...表示上级上级目录

注：不能在入口文件中使用相对导入，只能使用绝对导入

> 相对导入之所以能找到模块是因为从`__name__`来找，但是因为入口文件的`__name__ == __main__`，所以入口文件中不能使用相对导入

第8章 Python函数

8-1 认识函数

查看python内置函数的作用：

```
# python进入解释器

help(round)
```

8-2 函数的定义及运行特点

1. 如果函数没有返回值，则默认返回None
2. 函数的定义必须放在函数调用之前

设置递归的最大层数：

```
import sys
sys.setrecursionlimit(2000)
```

8-3 如何让函数返回多个结果

```python
def damage(skill1, skill2):
    damage1 = skill1 * 3
    damage2 = skill1 * 2 + 10
    return damage1, damage2


damages = damage(3, 6)              # 不推荐这种方式
# print(damages[0], damages[6])     # 不推荐这种方式
print(type(damages))                # <class 'tuple'>

# 序列解包
skill1_damage, skill2_damage = damage(3, 6)    # 推荐这种方式
```

8-4 序列解包与链式赋值

序列解包：把一个序列拆成了多个值

```python
a, b, c = 1, 2, 3

d = 1, 2, 3
print(type(d))          # <class 'tuple'>

a, b, c = d             # 序列解包：把一个tuple拆成了多个值

a = b= c = 1
```

8-5 必须参数与关键字参数

```python
def add(x, y):      # x, y是必须参数
    # x, y 是形参
    return x + y

a = add(1, 2)       # 1, 2是实参

c = add(y = 3, x = 2)       # 这是关键字参数，调用函数时明确指定把实参赋给哪个形参
# 关键字参数的意义在于代码的可读性
```

8-6 默认参数

定义时默认参数必须放在所有非默认参数后面

调用时非关键字参数也必须放在所有关键字参数之前

8-7 可变参数

```python
def demo1(param1, param2 = 2, *param3):

    print(param1)
```

```
    # print(type(param))      # <class 'tuple'>
    print(param2)
    print(param3)


def demo2(param1, *param3, param2 = 2):
    print(param1)
    # print(type(param))      # <class 'tuple'>
    print(param2)
    print(param3)

demo1(1, 2, 3, 4, 5, 6)
a = (1, 2, 3, 4, 5, 6)
demo1(a)     # 这样传的话会变成二维元组
demo1(*a)    # *a的作用就是序列解包
demo1('a', 1, 2, 3)      # parma1 = 'a', param2 = 1, param3 = (2, 3)

demo2('a', 1, 2, 3, 'param')    # parma1 = 'a', param3 = (1, 2, 3, 'param'), param2 = 2
# 注：不建议函数参数设置地这么复杂
```

8-8 关键字可变参数

```
def squsum(*param):
    sum = 0
    for i in param:
        sum += i * i
    print(sum)

squsum(1, 2, 3, 4, 5, 6)

# 设计一个函数，可以支持任意个数的关键字参数

def city_temperature(**temps):
    print(type(temps))      # <class 'dict'>
    for key, value in temps.items():
        print(key, ":", value)

city_temperature(bj='32c', xm='23c', sh='31c')
a = {'bj': '32c', 'xm': '23c', 'sh': '31c'}
city_temperature(**a)
```

8-9 变量作用域

```
c = 50              # 全局变量

def add(x, y):
    c = x + y       # 局部变量
    print(c)

add(1, 2)           # 3
```

```
print(c)            # 50

# 变量的作用域
# 函数内的变量只会作用在函数内部
# 在函数内部可以应用函数外部的变量
# 在for循环外部可以引用for循环内部中的变量
# python中没有块级作用域
```

8-11 global关键字

```
def demo():
    global c            # 在别的模块中也可以引用c
    c = 2

demo()      # 必须要先调用一下才能访问到c，因为前面只是函数的定义
print(c)
```

第9章 高级部分：面向对象

9-5 区别模块变量与类中的变量

```
class Student():
    name = '啊咧咧'
    age = 0

    def __init__(self, name, age):     # 对于构造函数，如果要显式地写return的话，则只能return None
        name = name
        age = age
        # return None

    def do_homewrd(self):
        print(self.name + ' 做作业')


student1 = Student('石敢当', 18)        # 会自动调用构造函数（当然也可以显式调用构造函数
student.__init() )
# student2 = Student()
# student3 = Student()
# a = student1.__init__('石乐之', 18)
# print(a)                    # None
# print(type(a))              # <class 'NoneType'>
student1.do_homewrd()         # 啊咧咧 做作业，原因见下一节：类变量和实例变量

# 类变量 和 实例变量
```

9-6 类变量和实例变量

```
# 类变量是和类相关联的
# 实例变量和对象相关联的

class Student():
    # name = '啊咧咧'                        # 类变量
    # age = 0                               # 类有名字和年龄不合适，名字和年龄应该是和对象相关的
    sum = 0                                 # 学生的总数

    def __init__(self, name, age):
        self.name = name                # 实例变量，只和对象相关
        self.age = age                  # 实例变量，只和对象相关

    def do_homewrd(self):
        print(self.name + ' 做作业')

student1 = Student('石敢当', 18)
student2 = Student('喜小乐', 20)
print(student1.name)            # 石敢当
print(student2.name)            # 喜小乐
print(Student.sum)            # 学生
```

9-7 类与对象的变量查找顺序

如果访问对象的变量，会先去实例变量中去找，如果没有则去类变量中去找，如果类中没有，则去Student的父类中去找。

```
class Student():
    name = '啊咧咧'
    age = 0

    def __init__(self, name, age):     # 对于构造函数，如果要显式地写return的话，则只能return None
        name = name             # 这种赋值方式并不是赋值给实例变量
        age = age
        # return None

    def do_homewrd(self):
        print('做作业')

student1 = Student('石敢当', 18)
print(student1.name)            # 啊咧咧，为什么？


print(Student.name)             # 啊咧咧

print(student1.__dict__)        # {}
# __dict__这个字典中保存了所有相关的变量

print(Student.__dict__)
'''
{
    '__module__': '__main__',
```

```
    'name': '啊咧咧', 'age': 0,
    '__init__': <function Student.__init__ at 0x00000221572D37B8>,
    'do_homewrd': <function Student.do_homewrd at 0x00000221572D3D90>,
    '__dict__': <attribute '__dict__' of 'Student' objects>,
    '__weakref__': <attribute '__weakref__' of 'Student' objects>,
    '__doc__': None
}
'''
```

9-9 在实例方法中访问实例变量与类变量

不管是访问类变量还是实例变量都需要加前缀：

- 类里面：
  - 实例变量：`self.name`
  - 类变量：`Student.sum1`，`self.__class__.sum1`，`self.sum1`（不推荐）
- 类外面：
  - 实例变量：`student.name`
  - 类变量：`Student.sum1`，`student.sum1`（不推荐）

但是不推荐用对象去调用类变量

```
class Student():
    sum1 = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def do_homework(self):
        print('做作业')
        print(self.name)                    # 石敢当
        print(self.__class__.sum1)           # 0


student = Student('石敢当', 18)
# print(student.__dict__)            # {'name': '石敢当', 'age': 18}
student.do_homework()
print(student.name)
print(Student.sum1)
```

9-10 类方法

实例方法中的`self`和类方法中的`cls`都可以换成别的名字，但不推荐这样做。

也可以用对象调用类方法或类变量，但是不推荐这么做

```
class Student():

    sum1 = 0
```

```
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def do_homework(self):       # 实例方法是为了操作实例变量
        print('做作业')
        print(self.sum1)

    @classmethod                 # 装饰器，说明该方法是类方法
    def plus_sum(cls):           # 类方法的参数列表也有一个特定的名字：cls（约定俗成）
        cls.sum1 += 1            # 每创建一个新的学生的时候，学生总数就+1
        print(cls.sum1)


student1 = Student('石敢当', 18)
Student.plus_sum()
student2 = Student('喜小乐', 19)
Student.plus_sum()
student3 = Student('郭大路', 20)
Student.plus_sum()

# 用对象调用类方法或类变量
student1.plus_sum()
print(student1.sum1)             # 4

student1.do_homework()           # 4
```

总结：对象可以访问实例变量和实例方法，也可以访问类对象和类方法（不推荐），但是类只能访问类对象和类方法。

9-11 静态方法

静态方法和类方法/实例方法的区别：

1. 参数中没有必需的cls/self
2. 函数上加上@staticmethod装饰器
3. 静态方法和类方法都不能访问实例变量

类和实例对象都可以调用静态方法。

能用静态方法的地方都可以用类方法代替

不建议经常使用静态方法

```
class Student():
    sum1 = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def do_homework(self):
```

```
        print('做作业')

    @classmethod
    def plus_sum(cls):
        cls.sum1 += 1
        print(cls.sum1)

    @staticmethod           # 静态方法
    def add(x, y):
        print('static: ' + str(x + y))
        print(Student.sum1)


student1 = Student('石敢当', 18)
Student.plus_sum()

student1.add(1, 2)
Student.add(3, 4)
```

9-13 没有什么是不能访问

python没有任何机制可以阻止访问私有变量，之所以读不到私有变量是因为python把**私有变量的名字给换了换**。

```
class Student():
    sum1 = 0

    def __init__(self, name, age):
        self.name = name                # 公开的
        self.age = age                  # 公开的
        self.__score = 0                # 私有的，只在前面加__
        self.__class__.sum1 += 1

    def marking(self, score):
        if score < 0:
            return '不能打负分'
        self.__score = score
        print(self.name + ' 的成绩是：' + str(self.__score))

    def do_homework(self):
        self.do_english_homework()
        print('做作业')

    def do_english_homework(self):
        print('做英语作业')


student = Student('石敢当', 18)
# print(student.__score)            # 'Student' object has no attribute '__score'
student.__score = -1                 # 这里设置的不是self.__score中的__score，而是python动态语言的特点，给student新添加了一个成员属性
student.marking(100)
print(student.__dict__)     # {'name': '石敢当', 'age': 18, '_Student__score': 100, '__score':
```

```
-1}
print(student._Student__score)          # 100, self.__score中的__score被转换成了_Student__score
```

9-14 继承

python支持多继承

```
# 继承

class Human():
    sum = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def do_homework(self):
        print('do english homework')

    def get_name(self):
        return self.name

class Student(Human):
    def __init__(self, school, name, age):
        self.school = school
        # Human.__init__(self, name, age)           # 不建议这么做，用类调用实例方法
        super(Student, self).__init__(name, age)    # 建议这种方式

    def do_homework(self):
        print('do homwork')
        super(Student, self).do_homework()

student = Student('人民路小学', '石敢当', 18)
print(student.age)
print(student.get_name())
student.do_homework()
```

第十章 正则表达式JSON

不管是python还是java中，如果要使用正则表达式匹配`\`，因为在正则表达式中`\\`表示一个`\`（`\`就不能用来匹配`\`了，因为一个`\`连同后面的字母会有别的含义，例如`\n`表示匹配回车，所以如果需要匹配`\n`这个字符串，则正则表达式中需要使用`\\n`），但是在python或java中的字符串也会进行转义，在字符串中同样是`\\`表示`\`，所以综合起来如果要匹配`\d`这个字符串，就需要使用`\\\\d`。

简单来说：java和python中`\\`表示要插入一个正则表达式的反斜线，`\\\\`表示要插入一个普通的反斜线。

```
import re

a = 'python1111java678javascript\\w'        # 这个字符串中\\表示一个\

r = re.findall('\\d', a)     # re.findall('[0-9]', a), \\d就是概括字符集
print(r)

r = re.findall('\\w', a)     # re.findall('[0-9a-fA-Z_]', a)，还可以匹配下划线
# 匹配时字符串中\\w ==> \w，在正则表达式中表示[0-9a-fA-Z_]
print(r)
r = re.findall('\\\\w', a)
print(r)            # ['\\w']，两个\\表示一个\
```

如上，如果使用`\`的方式的话，只推荐两种写法

- `\\d`：表示`[0-9]`
- `\\\\d`：表示`'\d'`这个字符串

或者使用`r'\d'`的这种方式（python）中

```
import re

a = 'python1111java678javascript\\w'

r = re.findall('\\d', a)     # re.findall('[0-9]', a), \\d就是概括字符集
print(r)

r = re.findall('\\w', a)     # re.findall('[0-9a-fA-Z_]', a)，还可以匹配下划线
print(r)
r = re.findall('\\\\w', a)
print(r)            # ['\\w']，两个\\表示一个\

r = re.findall(r'\d', a)    # 和'\\d'一样
print(r)
```

10-6 贪婪和非贪婪

```
import re

a = 'python 1111java678javascript'

# r = re.findall('[a-z]', a)
# print(r)        # ['p', 'y', 't', 'h', 'o', 'n', 'j', 'a', 'v', 'a', 'j', 'a', 'v', 'a', 's',
'c', 'r', 'i', 'p', 't']

r = re.findall('[a-z]{3,}', a)      # 注意：{3,}逗号后面不能有空格
print(r)        # ['python', 'java', 'javascript']

# 贪婪和非贪婪

"""
```

```
默认情况下python是贪婪的模式：如果把正则表达式的长度限定在一定的区间中，会倾向于尽可能多地取最大的一个长度值
比如上面正则表达式的最小长度是3，但匹配到长度为3时，并不认为匹配成功，会继续往后寻找，一直到一个字符不满足这个匹配条件才会停止
当匹配到pyt时，已经满足匹配条件，但是此时匹配不会停止，会一直向后寻找，直到找到空格时不满足匹配条件，此时停止。
"""


r = re.findall('[a-z]{3,6}?', a)      # 注非贪婪模式
print(r)        # ['pyt', 'hon', 'jav', 'jav', 'asc', 'rip']
```

10-7 匹配0次1次或无限多次

```
import re

a = 'pytho01111python1pythonn2'

r = re.findall('python*', a)        # *：对前面的一个字符匹配0次或者无限多次
print(r)            # ['pytho', 'python', 'pythonn']
r = re.findall('python+', a)        # +：对前面的一个字符匹配1次或者无限多次
print(r)            # ['python', 'pythonn']
r = re.findall('python?', a)        # +：对前面的一个字符匹配0次或者1次
print(r)            # ['pytho', 'python', 'python']

# ?可以用来进行去重
```

`?`的作用：

- 如果前面是一个范围（例如`{3,6}`），这个`?`表示转化为非贪婪
- 其他情况，`?`表示前面的一个字符重复0次或1次

10-8 边界匹配符

```
"""
边界匹配
"""

import re

qq = '100000112'
# 匹配qq号：4-8
r = re.findall('\\d{4,8}', qq)
print(r)            # 如果qq的长度大于8，也会有匹配结果，所以这个匹配逻辑不对
r = re.findall('^\\d{4,8}$', qq)        # ^：表示从字符串开始位置匹配；&：表示从字符串末尾位置开始匹配
print(r)            # []
```

10-9 组

```python
import re

s = 'life is short, i use python'

# 把life和python之间的匹配处理
r = re.search('life.*python', s)
print(r.group())        # life is short, i use python
r = re.search('life(.*)python', s)
# group()的参数指定要得到的组号
# group(0)记录的永远是正则表达式匹配的完整的结果
print(r.group(0))       # life is short, i use python
# 如果想得到完整匹配结果内部的某个分组，则参数从1开始
print(r.group(1))       #  is short, i use

r = re.findall('life(.*)python', s)
print(r)                # [' is short, i use ']

s = 'life is short, i use python, i love python'
r = re.search('life(.*)python(.*)python', s)
print(r.group(1))       #  is short, i use
print(r.group(2))       # , i love
print(r.groups())       # (' is short, i use ', ', i love ')
```

10-10 匹配模式

`re.findall()`还有第三个参数，表示匹配模式

```python
import re

language = 'PythonC#\nJavaPHP'

r = re.findall('c#.{1}', language, re.I)            # re.I:忽略大小写的一个模式
print(r)        # [] , .表示匹配除了换行符之外的任意一个字符
r = re.findall('c#.{1}', language, re.I | re.S)            # re.S：可以让.匹配任意一个字符，包括换行符
print(r)        # ['C#\n']
```

10-11 re.sub正则替换

```python
import re

language = 'PythonC#JavaC#PHP'
r = re.sub('C#', 'GO', language, 0)     # 0表示匹配将无限地替换下去（把所有的C#全替换掉）
print(r)            # PythonGOJavaGOPHP
r = re.sub('C#', 'GO', language, 1)
print(r)            # PythonGOJavaC#PHP
r = language.replace('C#', 'GO')
print(r)            # PythonGOJavaGOPHP
```

```
def convert(value):
    print(value)            # <_sre.SRE_Match object; span=(6, 8), match='C#'>
    # span：匹配的字符串出现的位置：index：6，7
    return '@' + value.group() + '@'

r = re.sub('C#', convert, language, 1)  # 第二个参数可以传一个函数
# 当正则表达式匹配到一个结果时，就会把它传到函数中，convert函数的返回结果将会替换'C#'
print(r)        # Python@C#@JavaC#PHP
```

10-13 search与match函数

- match从字符串首字母开始匹配，如果没有找着，则返回None
- search搜索整个字符串，直到找到第一个符合的，就返回
- match和search返回的结果比findall()返回的结果复杂
- match和search只要搜索到一个就会直接返回，findall会把所有的匹配结果返回

```
import re

s = 'A8C432Dds3sd23'

r = re.match('\\d', s)
print(r)        # None，match从字符串首字母开始匹配，如果没有找着，则返回None
r = re.search('\\d', s)
print(r)        # <_sre.SRE_Match object; span=(1, 2), match='8'>
# search搜索整个字符串，直到找到第一个符合的，就返回
# match和search返回的结果比findall()返回的结果复杂
# match和search只要搜索到一个就会直接返回，findall会把所有的匹配结果返回
print(r.span())
```

10-14 group分组

```
import re

s = 'life is short, i use python'

# 把life和python之间的匹配处理
r = re.search('life.*python', s)
print(r.group())        # life is short, i use python
r = re.search('life(.*)python', s)
# group()的参数指定要得到的组号
# group(0)记录的永远是正则表达式匹配的完整的结果
print(r.group(0))       # life is short, i use python
# 如果想得到完整匹配结果内部的某个分组，则参数从1开始
print(r.group(1))       #  is short, i use

r = re.findall('life(.*)python', s)
print(r)                # [' is short, i use ']


s = 'life is short, i use python, i love python'
```

```
r = re.search('life(.*)python(.*)python', s)
print(r.group(1))       #  is short, i use
print(r.group(2))       # , i love
print(r.groups())       # (' is short, i use ', ', i love ')
```

10-16 理解JSON

> JSON：JavaScript Object Notation

本质：一种轻量级的数据交换格式。

JSON是一种***数据格式***。

字符串是JSON的表现形式。

符合JSON格式的字符串叫***JSON字符串***。

json字符串的规范：

- json字符串中的**每个key都需要加引号**
- json字符串**必须用双引号**，不能用单引号
- json字符串中布尔值不需要加双引号（true/false）

注：JSON字符串和JavaScript中的对象的形式一样。

```
json_str1 = '[{"name":"leihou", "age":18, "male":true}, {"name":"alielie", "age":20}]'
```

反序列化：

```
import json
json_str = '{"name":"leihou", "age":18}'
student = json.loads(json_str)
```

序列化：

```
import json
student = {'name':'leihou', 'age':18}
json_str = json.dumps(student)
```

10-19 小谈JSON，JSON对象与JSON字符串

JSON也可以理解为ECMAScript的一个实现。

- **JSON**：数据交换的一种数据格式
- **JSON字符串**：符合JSON格式的字符串
- **JSON对象**：只在JavaScript中成立

第11章 Python的高级语法与用法

11-1 枚举其实是一个类

```
# python3中新增了枚举

from enum import Enum
class VIP(Enum):
    YELLOW = 1
    GREEN = 2
    BLACK = 3
    RED = 4

print(VIP.BLACK)            # VIP.BLACK：这才是枚举的意义所在，如果是普通类的话，就是打印 3
print(VIP.BLACK.value)      # 3
print(VIP.BLACK.name)       # BLACK
print(VIP['BLACK'])         # VIP.BLACK，通过枚举名称可以获得枚举类型

# VIP.BLACK：枚举类型
# VIP.BLACK.value：枚举的值
# VIP.BLACK.name：枚举的名称

for v in VIP:
    print(v)                # 枚举可以遍历
```

11-4 枚举的比较运算

```
from enum import Enum
class VIP(Enum):
    YELLOW = 1
    GREEN = 2
    BLACK = 3
    RED = 4

print(VIP.GREEN == 2)               # False
print(VIP.BLACK == VIP.GREEN)       # False
# print(VIP.GREEN > VIP.BLACK)      # 枚举不能进行大小比较
print(VIP.GREEN == VIP.GREEN)       # True
print(VIP.BLACK is VIP.BLACK)       # True
```

11-5 枚举注意事项

1. 枚举的名称不能相同

```
class VIP(Enum):
    YELLOW = 1
    YELLOW = 2      # 会报错
    BLACK = 3
    RED = 4
```

2. 枚举的值如果相等的话

```
from enum import Enum
class VIP(Enum):
    YELLOW = 1
    YELLOW_ALIAS = 1
    BLACK = 3
    RED = 4

for v in VIP:
    print(v)    # VIP.YELLOW, VIP.BLACK, VIP.RED

print(VIP.YELLOW_ALIAS)     # VIP.YELLOW , # VIP.YELLOW , YELLOW_ALIAS就相当于YELLOW的别名

for v in VIP.__members__.items():
    print(v)
'''
('YELLOW', <VIP.YELLOW: 1>)
('YELLOW_ALIAS', <VIP.YELLOW: 1>)
('BLACK', <VIP.BLACK: 3>)
('RED', <VIP.RED: 4>)
'''

for v in VIP.__members__:
    print(v)
'''
YELLOW
YELLOW_ALIAS
BLACK
RED
'''
```

11-6 枚举转换

```
from enum import Enum

class VIP(Enum):
    YELLOW = 1
    GREEN = 2
    BLACK = 3
    RED = 4

a = 1
print(VIP(a))
```

11-9 一切皆对象

python中一切皆对象，函数也是对象

11-10 什么是闭包

闭包：函数以及函数在**定义时候**的外部环境变量（同时还不能是全局的环境变量）所构成的一个整体

```
def curve_pre():
    a = 25
    def curve(x):
        return a * x * x
    return curve

a = 10
f = curve_pre()
print(f(2))         # 100，
print(f.__closure__[0].cell_contents)           # 25
```

```
def curve_pre():
    # a = 25
    def curve(x):
        return a * x * x
    return curve

a = 10
f = curve_pre()
print(f(2))         # 40，这里没有形成闭包，因为a没有在函数定义的环境变量中
```

闭包的意义在于：它保存的是一个环境。

```
def f1():
    a = 10
    def f2():
        a = 20          # a此时将被python认为是一个局部变量
        print(a)
    print(a)            # 10
    f2()                # 20
    print(a)            # 10

f1()
# 不是闭包
```

下面以java代码为例为什么闭包可以保存环境

```
interface Counter { int next(); }

public class LocalInnerClass {
    private int count = 0;

    Counter getCounter1(String name) {
        class LocalCounter1 implements Counter {

            @Override
```

```
            public int next() {
                System.out.print(name + ": ");
                return count++;
            }
        }
        return new LocalCounter1();
    }

    Counter getCounter2(String name) {
        return new Counter() {
            @Override
            public int next() {
                System.out.print(name + ": ");
                return count++;
            }
        };
    }

    public static void main(String[] args) {
        LocalInnerClass lic = new LocalInnerClass();
        Counter
                c1 = lic.getCounter1("Local inner"),
                c2 = lic.getCounter2("anonymous inner");
        for (int i = 0; i < 5; i++)
            System.out.println(c1.next());
        for (int j = 0; j < 5; j++)
            System.out.println(c2.next());
    }
}
/**
 * Local inner: 0
 * Local inner: 1
 * Local inner: 2
 * Local inner: 3
 * Local inner: 4
 * anonymous inner: 5
 * anonymous inner: 6
 * anonymous inner: 7
 * anonymous inner: 8
 * anonymous inner: 9
 */
```

因为count是lic的成员变量，所以只要调用getCounterx，就是用lic来调用的，因此闭包可以保存环境。

11-14 非闭包解决

```
origin = 0

def go(step):
    new_pos = origin + step
    origin = new_pos
    return new_pos
```

```
print(go(2))
print(go(3))
print(go(4))

# 这段代码是错的，因为在go()定义时，line5 origin出现在等号坐标，所以翻译器就会认为origin是局部变量，当line8执行go(2)时，
# 因为line4 用到了origin，但是origin还没有赋值，所以就会报错。

# 解决方案：在new_pos = origin + step之前 加上global origin
```

11-15 用闭包解决

```
origin = 0

def factory(pos):
    def go(step):
        nonlocal pos
        new_pos = pos + step
        pos = new_pos
        return new_pos
    return go

tourist = factory(origin)
print(tourist(2))           # 2
print(tourist(3))           # 5
print(tourist(4))           # 9

# 因为闭包可以保存环境变量，所以就可以记忆住上次调用的状态(pos的值)



# 或者
def factory():
    pos = 0
    def go(step):
        nonlocal pos
        new_pos = pos + step
        pos = new_pos
        return new_pos
    return go

tourist = factory()
print(tourist(2))           # 2
print(tourist(3))           # 5
print(tourist(4))           # 9
```

11-16 小谈函数式编程

闭包的两个用处：

1. 从模块级别调用某个局部变量
2. 保存环境变量，记忆上次调用的状态
    - 缺点：环境变量长驻内存，容易造成内存泄漏

第12章 函数式编程

12-1 lambda表达式

```
f2 = lambda x, y: a = x + y      # 会报错，因为：后面只能是表达式，不能是代码块
f2(1, 2)

f = lambda x, y: x + y
print(add(1, 2))
```

12-2 三元表达式

```
f = lambda x, y: x if x > y else y
```

12-3 map

```
list_x = [1, 2, 3, 4, 5, 6, 7, 8]
r = map(lambda x: x**2, list_x)     # class map(func, *iterables)
print(r)        # <map object at 0x0000028797C68E10>，map和list一样都是class
print(list(r))  # [1, 4, 9, 16, 25, 36, 49, 64]
```

```
list_x = [1, 2, 3, 4, 5, 6, 7, 8]
list_y = [8, 7, 6, 5, 4, 3]
r = map(lambda x, y: x*y, list_x, list_y)     # 后面列表的传入个数必须和lambda表达式的参数个数一致
print(list(r))  # [8, 14, 18, 20, 20, 18]
# 通过map计算得到的结果列表元素个数取决于长度较小的列表的长度
```

12-5 reduce

```
from functools import reduce        # 需要导入模块

list_x = ['1', '2', '3', '4', '5', '6', '7', '8']
# reduce在做连续计算
r = reduce(lambda x, y: x + y, list_x, '0')          # reduce中的函数参数必须有两个参数
print(r)            # 012345678
```

12-6 filter

```
list_x = [1, 0, 0, 1, 2, 0, 3, 2, 0]
r = filter(lambda x: x, list_x)
print(r)                # <filter object at 0x0000020AFC3C8E10>
print(list(r))          # [1, 1, 2, 3, 2]
```

12-8 装饰器

不用装饰器

```
import time

def print_current_time(func):
    print(time.time())
    func()

def f1():
    print('this is a f1')

def f2():
    print('this is a f2')

print_current_time(f1)
print_current_time(f2)
```

使用装饰器

```
import time

def decorator(func):            # 装饰器
    def wrapper():              # 把c8.py中的print_current_time又封装了一层
        print(time.time())
        func()
    return wrapper

def f1():
    print('this is a f1')

f = decorator(f1)
f()
```

完整版

```python
import time

def decorator(func):
    def wrapper():
        print(time.time())
        func()
    return wrapper

@decorator          # @ + 装饰器的名字
def f1():
    print('this is a f1')

f1()        # 装饰器的最大意义：保证原来的调用方式不变
```

能接受定义时候的复杂，但是不能接受调用时候的复杂

```python
# 带参数的装饰器
import time

def decorator(func):
    def wrapper(*args):
        print(time.time())
        print(args)         # ('1', '2')，args是个列表
        print(*args)        # 1 2，*args是把列表解包
        func(*args)
    return wrapper

@decorator          # @ + 装饰器的名字
def f1(func_name):
    print('this is a ', func_name)

@decorator
def f2(func_name1, func_name2):
    print(func_name1, func_name2)

@decorator
def f3(func_name1, func_name2, func_name3):
    pass

# f1('f1')
f2('1', '2')
```

```python
import time

def decorator(func):
    def wrapper(*args, **kw):
        print(time.time())
        func(*args, **kw)
    return wrapper

@decorator
```

```
def f1(func_name):
    print('this is a ', func_name)

@decorator
def f2(func_name1, func_name2):
    print(func_name1, func_name2)

@decorator
def f3(func_name1, func_name2, **kw):
    print("--" + func_name1 + "--" + func_name2)
    print(kw)

f1('f1')
f2('1', '2')
f3('f1', 'f2', a=1, b=2)
```

第14章 Pythonic

14-3 列表推导式

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

b = [i ** 2 for i in a]
print(b)    # [1, 4, 9, 16, 25, 36, 49, 64]
c = [i**2 for i in a if i >= 5]
print(c)    # 条件筛选，[25, 36, 49, 64]
c = {i**2 for i in a if i >= 5}
print(c)    # {64, 25, 36, 49}

# set也可以被推导
# 元组也可以

a = {1, 2, 3, 4, 5, 6, 7, 8}
c = [i**2 for i in a if i >= 5]
print(c)    # [25, 36, 49, 64]
c = {i**2 for i in a if i >= 5}
print(c)    # {64, 25, 36, 49}

# dict也可以被推导
students = {
    '喜小乐': 18,
    '石敢当': 20,
    '横小五': 15
}

b = [key for key, value in students.items()]
print(b)            # ['喜小乐', '石敢当', '横小五']
c = {value:key for key, value in students.items()}
print(c)            # {18: '喜小乐', 20: '石敢当', 15: '横小五'}
```

```
d = (key for key, value in students.items())
print(d)            # <generator object <genexpr> at 0x000001FF11CF61A8>
for x in d:
    print(x)
    '''
    喜小乐
    石敢当
    横小五
    '''
```

14-5 迭代器和可迭代对象

迭代器和可迭代对象

- 可迭代对象：凡是可以被for in循环遍历的对象都是可迭代对象
- 迭代器：1.迭代器是一个对象；2.迭代器一定是可迭代的
- **可迭代对象不一定是迭代器**

```
# 只要实现了__iter__和__next__就是迭代器
class BookCollection:
    def __init__(self):
        self.data = ['Harry Potter', 'Konan', '犬夜叉']
        self.cur = 0


    def __iter__(self):
        return self

    def __next__(self):
        if self.cur >= len(self.data):
            raise StopIteration()
        r = self.data[self.cur]
        self.cur += 1
        return r


books = BookCollection()
# 对于迭代器，可以使用next()，但是不能next(list)，所以list不是迭代器，只是可迭代对象
print(next(books))      # Harry Potter
print(next(books))      # Konan
print(next(books))      # 犬夜叉

# 迭代器是一次性的，所以只能迭代一次，如果想第二次迭代，只能重新复制
books = BookCollection()
for book in books:
    print(book)


import copy


books1 = copy.copy(books)           # 浅拷贝
```

```
books1 = copy.deepcopy(books)       # 深拷贝
```

生成器

```
def gen(max):                   # 生成器
    n = 0
    while n <= max:
        n += 1
        yield n

g = gen(10000)
for i in g:
    print(i)

# 也可以使用next()
# print(next(g))
# print(next(g))
# print(next(g))

n = (i for i in range(0, 10000))        # 生成器
```

14-6 None

None也是对象

```
print(type(None))       # <class 'NoneType'>
```

14-7 对象存在并不一定是True

- 如果类下面没有定义 `__bool__` 和 `__len__` 方法的话，那么bool结果就是True
- 如果类下面没有 `__bool__` 方法但是有 `__len__` 方法，如果 `__len__` 返回0，则bool结果为False，否则为True
- 如果类下面有 `__bool__` 方法，则bool结果取决于 `__bool__` 方法的返回结果

注：

- `len()` 方法也是调用 `__len__` 方法

```
class Test():
    pass

test = Test()

print(bool(test))           # True

class Test1():
    def __len__(self):
        return 0
```

```
test1 = Test1()
print(bool(test1))          # False

class Test2():
    def __bool__(self):
        return True
    def __len__(self):
        return 0

test2 = Test2()
print(bool(test2))          # True

class Test3():
    def __bool__(self):
        return False        # 只能返回bool类型
    def __len__(self):
        return 8            # 只能返回int

test3 = Test3()
print(bool(test3))          # False
print(len(test3))           # 8，如果Test3中没有__len__()函数，则len(test3)会报错
```

14-9 装饰器的副作用

使用装饰器后，会导致函数的一些信息丢失，如注解、函数名等等

```
import time

def decorator(func):
    def wrapper():
        print(time.time())
        func()
    return wrapper


def f1():
    '''
        This is f1
    '''
    print(f1.__name__)          # f1()
print(help(f1))
'''
Help on function f1 in module __main__:

f1()
    This is f1

None
'''
f1()
```

```
@decorator
def f2():
    '''
        This is f2
    '''
    print(f2.__name__)          # wrapper
print(help(f2))
'''
Help on function wrapper in module __main__:

wrapper()

None
'''
f2()
```

解决方法：`@wraps(func)`

```
import time
from functools import wraps

def decorator(func):
    @wraps(func)            # 把func这个函数的一些信息复制到wrapper这个闭包函数上
    def wrapper():
        print(time.time())
        func()
    return wrapper


def f1():
    '''
        This is f1
    '''
    print(f1.__name__)          # f1()
print(help(f1))
'''
Help on function f1 in module __main__:

f1()
    This is f1

None
'''
f1()



@decorator
def f2():
    '''

        This is f2
```

```
    '''
    print(f2.__name__)          # f2
print(help(f2))
'''
Help on function f2 in module __main__:

f2()
    This is f2

None
'''
f2()
```