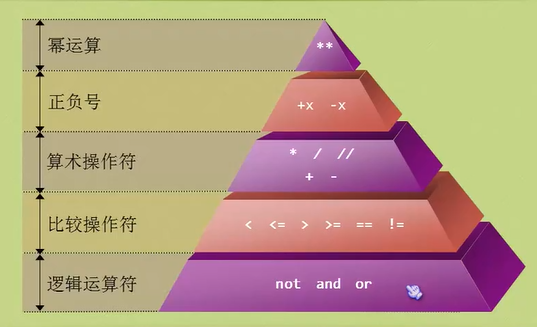
Python的语法基础部分

1. exit() 退出python交互模式
2. 严格缩进
3. #是注释
4. Python是区分大小写的
5. /除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数；//除法只取结果的整数部分
6. \*\*幂运算：3\*\*2=3\*3=9

优先级问题



* 1. 输出(一般用单引号括起来)

print('The quick brown fox', 'jumps over', 'the lazy dog')

print(100 + 200) 300

print('100 + 200 =', 100 + 200) 100 + 200 = 300

输入

1. 输入字符串 input(‘提示词’)

2.1 数据类型和变量

1. 转义字符\: \n表示换行，\t表示制表符， [\\表示的字符就是\](file:///\\表示的字符就是\)

2. r' '表示' '内部的字符串默认不转义: print(r'\\\t\\') [\\\t\\](file:///\\\t\\)

3. 允许用'''...'''的格式表示多行内容 （三重标点）

4. 类型转换 int()变成整数 str()变成字符串 float()变成浮点数

5. 类型获取：type(变量名) 或者 isinstance(变量名，数据类型)：如果变量名等于数据类型则返回true

4. 变量

\*等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量

\*在使用变量之前要先赋值

5常量

\*全部大写的变量名表示常量

6字符编码

\*ASCII编码是1个字节，而Unicode编码通常是2个字节

\* UTF-8编码是可变长编码的：<meta charset="UTF-8" />的信息，表示该网页正是用的UTF-8编码。

7字符串

\*字符串是以Unicode编码的，支持多语言

\*ord()函数获取字符的整数表示，chr()函数把编码转换为对应的字符

8 条件表达式

X if 条件 else y

9断言

Assert 条件：若关键字后面的条件为假的时候，程序自动崩溃并抛出assertionerror

10条件分支

If 条件：

条件为真执行的操作

else:

条件为假执行的在操作

While 条件：

条件为真执行的操作

For 目标 in表达式：

循环体

range([strat,] stop[,step=1])

11列表：

创建一个普通列表：

Member=[‘小甲鱼‘，’小布丁’,’黑皮’]

Member=[‘1’,’小甲鱼’,’3.14’，[1,2,3]]

Member.append(‘娃娃’)

Member.extend([‘竹林‘，’迷途‘])

Member.insert(1,’牡丹’) 第一位是插入位置

廖雪峰课程

Python基础

1.使用文本编辑器：

* 1. 直接运行py文件：mac和linux系统中，在.py文件的第一行加上一个特殊的注释

#!/usr/bin/env python3

chmod a+x代表给文件赋予执行权，

localhost:Desktop liuxxx$ chmod a+x hello.py

然后执行

localhost:Desktop liuxxx$ ./hello.py

终于成功了输出如下

hello world

2.使用4个空格的缩进、大小写敏感

3.数据类型和变量：

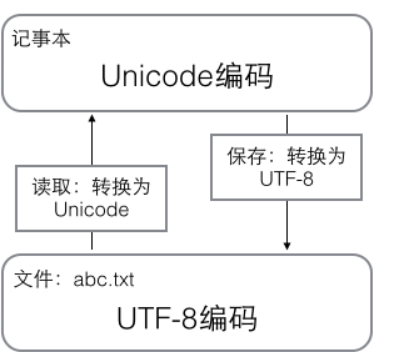
3.1 s4=r’’’hello,\n

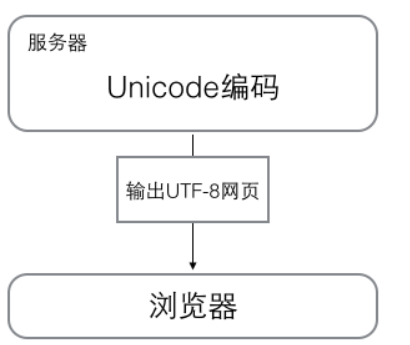
world’’’)

输出为：hello,\n

World

4.字符串编码：UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节





<meta charset="UTF-8" /> 代表使用UTF-8编码

5.‘ABC’ 字符串类型是str，在内存中以Unicode表示，所以一个字符对应若干个字节

b’ABC’ 的内容与上一行相同，但是每个字符都只占一个字节

6.以Unicode表示的str通过encode()方法可以编码为指定的bytes

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

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

>>>len(b’ABC’)

3

# -\*- coding: utf-8 -\*-代表按照UTF-8编码读取源代码

7.格式化字符串：

'Hello, %s' % 'world'

'Hi, %s, you have $%d.' % ('Michael', 1000000)

%d整数，%f浮点数，%s字符串，%x十六进制整数

8.List: list是一种有序的集合，list中可以是不同类型的元素，也可以是另一个list元素

>>> classmates = ['Michael', 'Bob', 'Tracy']

>>> len(classmates)

>>> classmates[0]

'Michael'

>>> classmates[-2]

'Bob'

>>> classmates.append('Adam') #追加元素到末尾

>>> classmates.insert(1, 'Jack') #在指定位置插入元素

要删除list末尾的元素，用pop()方法

要删除指定位置的元素，用pop(i)方法

>>> classmates.pop(1)

'Jack'

>>> classmates

['Michael', 'Bob', 'Tracy']

9 tuple元组，一旦初始化就不能修改

>>>classmates=(‘michael’,’tracy’,’lily’)

只有一个元素时定义时必须加一个逗号：>>>t=(1,)

tuple中的list可以改变

10.判断语句

If:

elif:

else:

\*\*input()返回的数据类型是str

11循环语句

For x in ……:把list或者tuple中的每个元素代入变量x,然后执行缩进块的语句。

range()函数是生成一个整数序列，range(5)生成的序列是从0开始小于5的整数；

while条件：

执行块

break 和continue的作用跟c++一致

12dict：利用key-value存储

>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}

>>> d['Michael']

95

判断key是否存在：

in 函数：>>>‘Thomas’ in d

get函数：d.get(‘Thomas’),如果不存在，返回none但是在python的交互环境下不显示这个结果，同时也可以自己指定value，d.get(‘Thomas’,-1)

删除一个key：pop(key)

dict中的key是不可变元素，所以不能是list

13set:一组key的集合，但是没有重复的key，且不存储value, set是无序的。两个set可以做数学意义上的交集、并集操作s1 & s2 s1 | s2

list为输入集合

>>> s = set([1, 1, 2, 2, 3, 3])

>>> s

{1, 2, 3}

添加元素到set中：s.add(key)

删除元素：s.remove(key)

函数

1调用函数：

help(函数名)可以用来查看函数的帮助信息

可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”

>>>a=abs

>>>a(-1)

>>>1

2定义函数

def 函数名（参数）：

return

函数日内部可以用return随时返回函数结果

定义一个空函数：

def nop():

pass

函数可以返回多个值，返回的是一个tuple

**import** math

**def** **move**(x, y, step, angle=0):

nx = x + step \* math.cos(angle)

ny = y - step \* math.sin(angle)

**return** nx, ny

>>>x,y=move(x100,100,60,math.pi/6)

3函数的参数

3.1默认参数

**def** **power**(x, n=2):

s = 1

**while** n > 0:

n = n - 1

s = s \* x

**return** s

>>>power(5) #相当于调用power(5,2)

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用enroll('Bob', 'M', 7),也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用enroll('Adam', 'M', city='Tianjin')

3.2可变参数

义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个\*号。在函数内部，参数numbers接收到的是一个tuple，因此，函数代码完全不变。

**def** **calc**(\*numbers):

sum = 0

**for** n **in** numbers:

sum = sum + n \* n

**return** sum

>>> nums = [1, 2, 3]

>>> calc(\*nums)

14

3.3关键字参数：扩展函数的功能

**def** **person**(name, age, \*\*kw):

print('name:', name, 'age:', age, 'other:', kw)

>>> extra = {'city': 'Beijing', 'job': 'Engineer'}

>>> person('Jack', 24, \*\*extra)

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

3.4命名关键字参数

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

**def** **person**(name, age, \*, city, job):

print(name, age, city, job)

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符\*了

**def** **person**(name, age, \*args, city, job):

print(name, age, args, city, job)

\*args是可变参数，args接收的是一个tuple；

\*\*kw是关键字参数，kw接收的是一个dict。

4递归函数

**def** **fact**(n):

**if** n==1:

**return** 1

**return** n \* fact(n - 1)

解决递归调用栈溢出的方法是通过尾递归优化

5Format()函数：基本语法是通过 **{}** 和 **:** 来代替以前的 **%** 。

format 函数可以接受不限个参数，位置可以不按顺序。

>>>"{} {}".format("hello", "world") # 不设置指定位置，按默认顺序

'hello world'

>>> "{1} {0} {1}".format("hello", "world") # 设置指定位置

'world hello world'

6 eval函数就是实现list、dict、tuple与str之间的转化,即将输入的引号去掉。‘123’->123

高级特性

1切片（可运用在list,tuple,字符串）

取前三个数：

>>> L[0:3] #如果第一个索引是0，可以省略。这个语句代表从0开始取，直到索引3为止，但是不包括索引3

>>>L[:10:2]#代表前十个数，每两个取一个

>>>’ABCDEF’[::2]

>>>’ACE’

2迭代：python不仅可以迭代list和tuple，也可以迭代其他可迭代对象，比如dict，字符串

dict默认迭代key，

>>> **for** key **in** d:

... print(key)

如果要迭代value,可以使用 for v in d.value(); 如果要迭代key和value可以使用for k,v in d.items()

判断一个对象是否为可迭代对象：通过collections模块的Iterable类型判断：

>>> **from** collections **import** Iterable

>>> isinstance('abc', Iterable) *# str是否可迭代*

True

将list实现类似java那样的下标循环，用enumerate()函数把一个list变成索引-元素对

>>> **for** i, value **in** enumerate(['A', 'B', 'C']):

... print(i, value)

...

0 A

1 B

2 C

3列表生成式

1要生成[1x1, 2x2, 3x3, ..., 10x10]怎么做

>>> [x \* x **for** x **in** range(1, 11)]

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

for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方

>>> [x \* x **for** x **in** range(1, 11) **if** x % 2 == 0]

[4, 16, 36, 64, 100]

可以使用两层循环，可以生成全排列

>>> [m + n **for** m **in** 'ABC' **for** n **in** 'XYZ']

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

列表生成式也可以使用两个变量来生成list，dict可以同时迭代key和value

>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }

>>> [k + '=' + v **for** k, v **in** d.items()]

['y=B', 'x=A', 'z=C']

最后把一个list中所有的字符串变成小写：

>>> L = ['Hello', 'World', 'IBM', 'Apple']

>>> [s.lower() **for** s **in** L]

['hello', 'world', 'ibm', 'apple']

4生成器：这种一边循环一边计算的机制，generator保存的是算法

方法一：把一个列表生成式的[]改成()，就创建了一个generator

>>> g = (x \* x **for** x **in** range(10))

可以通过next()函数获得generator的下一个返回值

>>> **next**(g)

generator是可迭代对象，所以用for循环也可以

>>> g = (x \* x **for** x **in** range(10))

>>> **for** n **in** g:

... print(n)

生成器的算法用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

**def** **fib**(max):

n, a, b = 0, 0, 1

**while** n < max:

**yield** b

a, b = b, a + b

n = n + 1

**return** 'done'

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

>>> g = fib(6)

>>> **while** True:

... **try**:

... x = next(g)

... print('g:', x)

... **except** StopIteration **as** e:

... print('Generator return value:', e.value)

... **break**

普通函数调用直接返回结果，generator函数的“调用”实际返回一个generator对象（用next和for就可以了）：

>>> g = fib(6)

>>> g

<generator object fib at 0x1022ef948>

5迭代器：可以被next()函数调用并不断返回下一个值的对象（生成器是iterator对象）

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

>>> isinstance(iter('abc'), Iterator)

True

函数式编程

高阶函数

1变量可以指向函数，可以通过该变量来调用这个函数，函数名也是变量

>>> f = abs

>>> f(-10)

10

2传入函数

一个函数接收另一个函数作为参数，这种函数就称之为高阶函数

**def** **add**(x, y, f):

**return** f(x) + f(y)

>>>print(add(-5, 6, abs))

return 11

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

>>> **def** **f**(x):

... **return** x \* x

...

>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> list(r) #list() 方法用于将元组转换为列表

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

4reduce():reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算

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

5filter()函数：filter()也接收一个函数和一个序列。和map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。filter()函数返回的是一个Iterator，也就是一个惰性序列，所以要强迫filter()完成计算结果，需要用list()函数获得所有结果并返回list。

**def** **is\_odd**(n):

**return** n % 2 == 1

list(filter(is\_odd, [1, 2, 4, 5, 6, 9, 10, 15]))

#在一个list中，删掉偶数，只保留奇数，

反转字符串用str[::-1]

6sorted()函数：可以对list进行排序

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

>>> sorted([36, 5, -12, 9, -21], key=abs) 按绝对值大小排序，排序是从小到大

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

>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower) 按忽略大小写，按照字母序排序

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

sorted(L, key=，reverse=true) reverse代表排序是从大到小

7返回函数

1.1函数可以作为返回值

1.2每次调用函数时，即使参数相同，也会返回不同的函数

>>> f1 = lazy\_sum(1, 3, 5, 7, 9)

>>> f2 = lazy\_sum(1, 3, 5, 7, 9)

>>> f1==f2

False

8闭包

返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量

返回一个函数时，牢记该函数并未执行，返回函数中不要引用任何可能会变化的变量。

9匿名函数lamda：冒号前面的x表示函数参数，只能有一个表达式

lambda x: x\*x #等价于

def f(x):

return x\*x

10装饰器：在代码运行期间动态增加功能的方式

1.1函数对象有一个\_\_name\_\_属性，可以拿到函数的名字

>>> now.\_\_name\_\_

'now'

1.2定义日志文件

1.2.1没有带参数的decorator

**import** functools

**def** **log**(func):

@functools.wraps(func)

**def** **wrapper**(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

**return** func(\*args, \*\*kw)

**return** wrapper

1.2.2带参数的decorator

**import** functools

**def** **log**(text):

**def** **decorator**(func):

@functools.wraps(func)

**def** **wrapper**(\*args, \*\*kw):

print('%s %s():' % (text, func.\_\_name\_\_))

**return** func(\*args, \*\*kw)

**return** wrapper

**return** decorator

1.3利用@语法放在now函数定义处

@log

def now():

pass #效果相当于 now = log(now)

@log('execute')

def f():

pass # 效果相当于 now = log('execute')(now)

11偏函数：把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单

1将字符转换成大量的二进制字符串

**def** **int2**(x, base=2):

**return** int(x, base)#效果等价于下文用偏函数

>>> **import** functools

>>> int2 = functools.partial(int, base=2)

>>> int2('1000000')

64

模块

1一个.py文件就称之为一个模块（Module）

2按目录来组织模块的方法，称为包（Package），为了避免模块名冲突

每一个包目录下面都会有一个\_\_init\_\_.py的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包

3自己创建模块时要注意命名，不能和Python自带的模块名称冲突，最好先查看系统是否已存在该模块，检查方法是在Python交互环境执行import abc，若成功则说明系统存在此模块。

4使用模块

1.1python 模块的标准文件模板

*#!/usr/bin/env python3 第1行注释可以让这个hello.py文件直接在Unix/Linux/Mac上运行*

*# -\*- coding: utf-8 -\*-第2行注释表示.py文件本身使用标准UTF-8编码*

' a test module '第4行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释

\_\_author\_\_ = 'Michael Liao'第6行使用\_\_author\_\_变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

1.2导入模块：import xxx

导入sys模块后，我们就有了变量sys指向该模块，利用sys这个变量，就可以访问sys模块的所有功能。

sys模块有一个argv变量，用list存储了命令行的所有参数。argv至少有一个元素，因为第一个参数永远是该.py文件的名称

1.3

if \_\_name\_\_=='\_\_main\_\_':

test()

当我们在命令行运行hello模块文件时，Python解释器把一个特殊变量\_\_name\_\_置为\_\_main\_\_，而如果在其他地方导入该hello模块时，if判断将失败，因此，这种if测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

5作用域

正常的函数和变量名是公开的（public），可以被直接引用

类似\_\_xxx\_\_这样的变量是特殊变量，可以被直接引用，但是有特殊用途

类似\_xxx和\_\_xxx这样的函数或变量就是非公开的（private）

作用：非常有用的代码封装和抽象的方法

6安装第三方模块

路径，pip,anaconda

面向对象编程oop

1类与实例

1.1用class 定义类，结构为：

class Student(object):

pass

class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类

1.2创建实例是通过类名+()实现的：

>>> bart = Student()

>>> bart

1.3可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方法，在创建实例的时候，就把name，score等属性绑上去

**def** **\_\_init\_\_**(self, name, score):

self.name = name

self.score = score

特殊方法“\_\_init\_\_”前后分别有两个下划线

第一个参数永远是self，表示创建的实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。

有了\_\_init\_\_方法，在创建实例的时候，就不能传入空的参数了，必须传入与\_\_init\_\_方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去：

1.4普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且，调用时，不用传递该参数

2访问限制

在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问

**class Student(object):**

**def** **\_\_init\_\_**(self, name, score):

self.\_\_name = name

self.\_\_score = score

**def** **get\_name**(self):

**return** self.\_\_name

**def** **get\_score**(self):

**return** self.\_\_score

**def** **set\_score**(self, score):

self.\_\_score = score

在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，

你会看到以一个下划线开头的实例变量名，比如\_name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”

3继承与多态

1.1继承：最大的好处是子类获得了父类的全部功能

1.2多态：当子类和父类都存在相同的run()方法时，我们说，子类的run()覆盖了父类的run()，在代码运行的时候，总是会调用子类的run()

由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思

4获取对象信息

1.1使用type()函数：判断对象类型

如果要判断一个对象是否是函数怎么办？可以使用types模块中定义的常量

1.2isinstance()函数

1.2.1isinstance()判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上

1.2.2还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是list或者tuple

>>> isinstance([1, 2, 3], (**list**, tuple))

**True**

>>> isinstance((1, 2, 3), (**list**, tuple))

**True**

1.3dir()函数：要获得一个对象的所有属性和方法

1.4配合getattr()、setattr()以及hasattr()，我们可以直接操作一个对象的状态：

>>> **class MyObject(object):**

... **def** **\_\_init\_\_**(self):

... self.x = 9

... **def** **power**(self):

... **return** self.x \* self.x

...

>>> obj = MyObject()

>>> hasattr(obj, 'x') *# 有属性'x'吗？*

True

>>> obj.x

9

>>> hasattr(obj, 'y') *# 有属性'y'吗？*

False

>>> setattr(obj, 'y', 19) *# 设置一个属性'y'*

>>> hasattr(obj, 'y') *# 有属性'y'吗？*

True

>>> getattr(obj, 'y') *# 获取属性'y'*

19

>>> obj.y *# 获取属性'y'*

19

>>> getattr(obj, 'z', 404) *# 获取属性'z'，如果不存在，返回默认值404*

404

>>> hasattr(obj, 'power') *# 有属性'power'吗？*

True

>>> getattr(obj, 'power') *# 获取属性'power'*

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x10077a6a0>>

>>> fn = getattr(obj, 'power') *# 获取属性'power'并赋值到变量fn*

>>> fn *# fn指向obj.power*

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x10077a6a0>>

>>> fn() *# 调用fn()与调用obj.power()是一样的*

81

5实例属性和类属性

在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

**class Student(object):**

name = 'Student'

>>> s = Student() *# 创建实例s*

>>> print(s.name) *# 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性*

Student

>>> print(Student.name) *# 打印类的name属性*

Student

>>> s.name = 'Michael' *# 给实例绑定name属性*

>>> print(s.name) *# 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性*

Michael

>>> print(Student.name) *# 但是类属性并未消失，用Student.name仍然可以访问*

Student

>>> **del** s.name *# 如果删除实例的name属性*

>>> print(s.name) *# 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了*

Student

面对对象高级编程

1\_\_slots\_\_：想要限制实例的属

**class Student(object):**

\_\_slots\_\_ = ('name', 'age') *# 用tuple定义允许绑定的属性名*

仅对当前类实例起作用，对继承的子类是不起作用的

除非在子类中也定义\_\_slots\_\_，这样，子类实例允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。

2@property: 负责把一个方法变成属性调用

**class Student(object):**

@property

**def** **score**(self):

**return** self.\_score

@score.setter

**def** **score**(self, value):

**if** **not** isinstance(value, int):

**raise** ValueError('score must be an integer!')

**if** value < 0 **or** value > 100:

**raise** ValueError('score must between 0 ~ 100!')

self.\_score = value

@property的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上@property就可以了，此时，@property本身又创建了另一个装饰器@score.setter，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性

3多重继承：一个子类就可以同时获得多个父类的所有功能

Class Dog(Animal,Runable)

MixIn的目的就是给一个类增加多个功能

4定制类：

1.1\_\_str\_\_:返回一个好看的字符串

>>> **class Student(object):**

... **def** **\_\_init\_\_**(self, name):

... self.name = name

... **def** **\_\_str\_\_**(self):

... **return** 'Student object (name: %s)' % self.name

...

>>> print(Student('Michael'))

Student object (name: Michael)

直接敲变量不用print，打印出来的实例还是不好看

>>> s = Student('Michael')

>>> s

<\_\_main\_\_.Student object at 0x109afb310>

这是因为直接显示变量调用的不是\_\_str\_\_()，而是\_\_repr\_\_()，两者的区别是\_\_str\_\_()返回用户看到的字符串，而\_\_repr\_\_()返回程序开发者看到的字符串，也就是说，\_\_repr\_\_()是为调试服务的。

解决办法是再定义一个\_\_repr\_\_()。但是通常\_\_str\_\_()和\_\_repr\_\_()代码都是一样的，所以，有个偷懒的写法:

**class Student(object):**

**def** **\_\_init\_\_**(self, name):

self.name = name

**def** **\_\_str\_\_**(self):

**return** 'Student object (name=%s)' % self.name

\_\_repr\_\_ = \_\_str\_\_

1.2\_\_iter\_\_

如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的\_\_next\_\_()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

**class Fib(object):**

**def** **\_\_init\_\_**(self):

self.a, self.b = 0, 1 *# 初始化两个计数器a，b*

**def** **\_\_iter\_\_**(self):

**return** self *# 实例本身就是迭代对象，故返回自己*

**def** **\_\_next\_\_**(self):

self.a, self.b = self.b, self.a + self.b *# 计算下一个值*

**if** self.a > 100000: *# 退出循环的条件*

**raise** StopIteration()

**return** self.a *# 返回下一个值*

1.3\_\_getitem\_\_:像list那样按照下标取出元素

\_\_getitem\_\_()传入的参数可能是一个int，也可能是一个切片对象slice，所以要做判断：

**class Fib(object):**

**def** **\_\_getitem\_\_**(self, n):

**if** isinstance(n, int): *# n是索引*

a, b = 1, 1

**for** x **in** range(n):

a, b = b, a + b

**return** a

**if** isinstance(n, slice): *# n是切片*

start = n.start

stop = n.stop

**if** start **is** None:

start = 0

a, b = 1, 1

L = []

**for** x **in** range(stop):

**if** x >= start:

L.append(a)

a, b = b, a + b

**return** L

1.4\_\_getattr\_\_:动态返回一个属性

**class Student(object):**

**def** **\_\_init\_\_**(self):

self.name = 'Michael'

**def** **\_\_getattr\_\_**(self, attr):

**if** attr=='score':

**return** 99

>>> s = Student()

>>> s.name

'Michael'

>>> s.score

99

1.5\_\_call\_\_:一个对象实例可以有自己的属性和方法，当我们调用实例方法，用\_\_call\_\_直接在实例本身上调用

5枚举类：定义一个clss类型，每个常量都是class的一个唯一实例

from **enum** **import** Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'

这样我们就获得了Month类型的枚举类，可以直接使用Month.Jan来引用一个常量，或者枚举它的所有成员

**for** name, member **in** Month.\_\_members\_\_.items():

print(name, '=>', member, ',', member.value)

@unique装饰器可以帮助我们检查保证没有重复值

6元类：

1.1type()函数: 可以返回一个对象的类型，又可以创建出新的类型

>>> def fn(self, name='world'): # 先定义函数

... print('Hello, %s.' % name)

...

>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello class

>>> h = Hello()

>>> h.hello()

Hello, world.

要创建一个class对象，type()函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
3. class的方法名称与函数绑定，这里我们把函数fn绑定到方法名hello上

1.2metaclass元类（难没有细看）

错误、调试和测试

1错误处理

1.1错误码：在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见. 用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错

1.2错误处理机制：

try: 当我们认为某些代码可能会出错时，就可以用try来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即except语句块，执行完except后，如果有finally语句块，则执行finally语句块，至此，执行完毕。

**try**:

print('try...')

r = 10 / 0

print('result:', r)

**except** ZeroDivisionError **as** e:

print('except:', e)

**finally**:

print('finally...')

print('END')

**try**...

**except**: division by zero

**finally**...

END

Python的错误其实也是class，所有的错误类型都继承自BaseException

1.3调用栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看err.py

*# err.py:*

**def** **foo**(s):

**return** 10 / int(s)

**def** **bar**(s):

**return** foo(s) \* 2

**def** **main**():

bar('0')

main()

执行结果如下：

$ python3 err.py

Traceback (most recent **call** **last**):

File "err.py", line 11, **in** <**module**>

main()

File "err.py", line 9, **in** main

bar('0')

File "err.py", line 6, **in** bar

return foo(s) \* 2

File "err.py", line 3, **in** foo

return 10 / **int**(s)

ZeroDivisionError: division **by** zero

1.4记录错误：Python内置的logging模块可以非常容易地记录错误信息

import logging

def foo(s):

return 10 / int(s)

def bar(s):

return foo(s) \* 2

def main():

try:

bar('0')

except Exception as e:

logging.exception(e)

main()

print('END')

$ python3 err\_logging.py

ERROR:root:division by zero

Traceback (most recent call last):

File "err\_logging.py", line 13, in main

bar('0')

File "err\_logging.py", line 9, in bar

return foo(s) \* 2

File "err\_logging.py", line 6, in foo

return 10 / int(s)

ZeroDivisionError: division by zero

END

1.5抛出错误

果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用raise语句抛出一个错误的实例：

*# err\_raise.py*

**class FooError(ValueError):**

**pass**

**def** **foo**(s):

n = int(s)

**if** n==0:

**raise** FooError('invalid value: %s' % s)

**return** 10 / n

foo('0')

$ python3 err\_raise.py

Traceback (most recent **call** **last**):

File "err\_throw.py", line 11, **in** <**module**>

foo('0')

File "err\_throw.py", line 8, **in** foo

raise FooError('invalid value: %s' % s)

\_\_main\_\_.FooError: invalid **value**: 0

在Python中，要想引发异常，最简单的形式就是输入关键字raise，后跟要引发的异常的名称。异常名称标识出具体的类： Python异常处理是那些类的对象。

2调试

1.1print()

1.2assert()

**def** **foo**(s):

n = int(s)

**assert** n != 0, 'n is zero!'

**return** 10 / n

**def** **main**():

foo('0')

assert的意思是，表达式n != 0应该是True，否则，根据程序运行的逻辑，后面的代码肯定会出错。

如果断言失败，assert语句本身就会抛出AssertionError：

$ python err.py

Traceback (most recent **call** **last**):

...

AssertionError: n **is** zero!

启动Python解释器时可以用-O参数来关闭assert, 关闭后，你可以把所有的assert语句当成pass来看。

1.3logging

**import** logging

logging.basicConfig(level=logging.INFO)

s = '0'

n = **int**(s)

logging.info('n = %d' % n)

print(10 / n)

输出

$ python err.py

INFO:root:n = 0

Traceback (most recent **call** **last**):

File "err.py", line 8, **in** <**module**>

print(10 / n)

ZeroDivisionError: division **by** zero

这就是logging的好处，它允许你指定记录信息的级别，有debug，info，warning，error等几个级别，当我们指定level=INFO时，logging.debug就不起作用了。同理，指定level=WARNING后，debug和info就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

1.4pdb: 让程序以单步方式运行，可以随时查看运行状态。

*# err.py*

s = '0'

n = int(s)

**print**(10 / n)

启动程序：

$ python -m pdb err.py

> /Users/michael/Github/learn-python3/samples/debug/err.py(2)<**module**>()

-> s = '0'

输入命令l来查看代码，输入命令n可以单步执行代码，

(Pdb) l

1 *# err.py*

2 -> s = '0'

3 n = int(s)

4 **print**(10 / n)

任何时候都可以输入命令p 变量名来查看变量

(Pdb) p s

'0'

(Pdb) p n

0

输入命令q结束调试，退出程序

(Pdb) q

1.5pdb.set\_trace()

我们只需要import pdb，然后，在可能出错的地方放一个pdb.set\_trace()，就可以设置一个断点

运行代码，程序会自动在pdb.set\_trace()暂停并进入pdb调试环境，可以用命令p查看变量，或者用命令c继续运行：

*# err.py*

**import** pdb

s = '0'

n = int(s)

pdb.set\_trace() *# 运行到这里会自动暂停*

print(10 / n)

1.6IDE

目前比较好的Python IDE有PyCharm：<http://www.jetbrains.com/pycharm/>

3单元测试：单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

1.1

**import** unittest

**from** mydict **import** Dict

**class TestDict(unittest.TestCase):**

**def** **test\_init**(self):

d = Dict(a=1, b='test')

self.assertEqual(d.a, 1)

self.assertEqual(d.b, 'test')

self.assertTrue(isinstance(d, dict))

**def** **test\_key**(self):

d = Dict()

d['key'] = 'value'

self.assertEqual(d.key, 'value')

**def** **test\_attr**(self):

d = Dict()

d.key = 'value'

self.assertTrue('key' **in** d)

self.assertEqual(d['key'], 'value')

**def** **test\_keyerror**(self):

d = Dict()

**with** self.assertRaises(KeyError):

value = d['empty']

**def** **test\_attrerror**(self):

d = Dict()

**with** self.assertRaises(AttributeError):

value = d.empty

**if** \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

为了编写单元测试，我们需要引入Python自带的unittest模块，编写单元测试时，我们需要编写一个测试类，从unittest.TestCase继承

以test开头的方法就是测试方法，不以test开头的方法不被认为是测试方法，测试的时候不会被执行

对每一类测试都需要编写一个test\_xxx()方法。由于unittest.TestCase提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是assertEqual()

另一种重要的断言就是期待抛出指定类型的Error，比如通过d['empty']访问不存在的key时，断言会抛出KeyError

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在mydict\_test.py的最后加上两行代码

**if** \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

这样就可以把mydict\_test.py当做正常的python脚本运行：$ python mydict\_test.py

另一种方法是在命令行通过参数-m unittest直接运行单元测试:$ python -m unittest mydict\_test

1.2setUp与tearDown

可以在单元测试中编写两个特殊的setUp()和tearDown()方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

4文档测试

doctest非常有用，不但可以用来测试，还可以直接作为示例代码。通过某些文档生成工具，就可以自动把包含doctest的注释提取出来。用户看文档的时候，同时也看到了doctest

if \_\_name\_\_ == '\_\_main\_\_':

import doctest

doctest.testmod()

super()函数：用于调用父类(超类)的一个方法

Python 3 可以使用直接使用**super().xxx**

class A:

pass

class B(A):

def add(self, x):

super().add(x)

I0编程

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。

要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。

1文件读写

在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）

1.1读文件

使用Python内置的open()函数，传入文件名和标示符

>>> f = open('/Users/michael/test.txt', 'r') #标识符’r’表示读

如果文件不存在，open()函数就会抛出一个IOError的错误，并且给出错误码和详细的信息告诉你文件不存在

如果文件打开成功，接下来，调用read()方法可以一次读取文件的全部内容，Python把内容读到内存，用一个str对象表示

>>> f.read()

'Hello, world!'

最后一步是调用close()方法关闭文件。文件使用完毕后必须关闭

>>> f.close()

由于文件读写时都有可能产生IOError，一旦出错，后面的f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally来实现

**try**:

f = open('/path/to/file', 'r')

print(f.read())

**finally**:

**if** f:

f.close()

但是每次这样写很繁琐，Python引入了with语句来自动帮我们调用close()方法：

**with** open('/path/to/file', 'r') **as** f:

print(f.read())

效果与上面是一样的

反复调用read(size)方法，每次最多读取size个字节的内容；调用readline()可以每次读取一行内容，调用readlines()一次读取所有内容并按行返回list。

1.1.1如果读取二进制文件（因为之前说的都是读取文本文件，并且是UTF-8编码的）

要读取二进制文件，比如图片、视频等等，用'rb'模式打开文件

>>> f = open('/Users/michael/test.jpg', 'rb')

>>> f.read()

b'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' *# 十六进制表示的字节*

1.1.2字符编码

要读取非UTF-8编码的文本文件，需要给open()函数传入encoding参数，例如，读取GBK编码的文件：

>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk')

>>> f.read()

'测试'

open()函数还接收一个errors参数，表示如果遇到编码错误后如何处理。最简单的方式是直接忽略：

>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk', errors='ignore')

1.2写文件

写文件和读文件是一样的，唯一区别是调用open()函数时，传入标识符'w'或者'wb'表示写文本文件或写二进制文件

>>> f = open('/Users/michael/test.txt', 'w')

>>> f.write('Hello, world!')

>>> f.close()

可以反复调用write()来写入文件，但是务必要调用f.close()来关闭文件。忘记调用close()的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用with语句来得保险

**with** open('/Users/michael/test.txt', 'w') **as** f:

f.write('Hello, world!')

文件读写是通过open()函数打开的文件对象完成的。使用with语句操作文件IO是个好习惯

2 StringIO 和 BytesIO

1.1StringIO: 在内存中读写str,要把str写入StringIO，我们需要先创建一个StringIO，然后，像文件一样写入即可

>>> **from** io **import** StringIO

>>> f = StringIO()

>>> f.write('hello')

5

>>> f.write(' ')

1

>>> f.write('world!')

6

>>> print(f.getvalue())

hello world!

getvalue()方法用于获得写入后的str

读取StringIO：可以用一个str初始化StringIO，然后，像读文件一样读取

>>> **from** io **import** StringIO

>>> f = StringIO('Hello!\nHi!\nGoodbye!')

>>> **while** True:

... s = f.readline()

... **if** s == '':

... **break**

... print(s.strip())

...

Hello!

Hi!

Goodbye!

1.2BytesIO: 操作二进制数据, 实现了在内存中读写bytes

>>> **from** io **import** BytesIO

>>> f = BytesIO()

>>> f.write('中文'.encode('utf-8'))

6

>>> print(f.getvalue())

b'\xe4\xb8\xad\xe6\x96\x87'

和StringIO类似，可以用一个bytes初始化BytesIO，然后，像读文件一样读取

>>> **from** io **import** BytesIO

>>> f = BytesIO(b'\xe4\xb8\xad\xe6\x96\x87')

>>> f.read()

b'\xe4\xb8\xad\xe6\x96\x87'

3操作文件和目录

操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python内置的os模块也可以直接调用操作系统提供的接口函数

>>> **import** os

>>> os.name *# 操作系统类型*

'posix'

如果是posix，说明系统是Linux、Unix或Mac OS X，如果是nt，就是Windows系统。

要获取详细的系统信息，可以调用uname()函数

>>> os.uname()

posix.uname\_result(sysname='Darwin', nodename='MichaelMacPro.local', release='14.3.0', version='Darwin Kernel Version 14.3.0: Mon Mar 23 11:59:05 PDT 2015; root:xnu-2782.20.48~5/RELEASE\_X86\_64', machine='x86\_64')

1.1环境变量

在操作系统中定义的环境变量，全部保存在os.environ这个变量中

>>> os.environ

environ({'VERSIONER\_PYTHON\_PREFER\_32\_BIT': 'no', 'TERM\_PROGRAM\_VERSION': '326', 'LOGNAME': 'michael', 'USER': 'michael', 'PATH': '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/mysql/bin', ...})

要获取某个环境变量的值，可以调用os.environ.get('key')

>>> os.environ.get('PATH')

'/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/mysql/bin'

1.2操作文件和目录

操作文件和目录的函数一部分放在os模块中，一部分放在os.path模块中

查看、创建和删除目录可以这么调用

*# 查看当前目录的绝对路径:*

>>> os.path.abspath('.')

'/Users/michael'

# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来:

>>> os.path.join('/Users/michael', 'testdir')

'/Users/michael/testdir'

# 然后创建一个目录:

>>> os.mkdir('/Users/michael/testdir')

# 删掉一个目录:

>>> os.rmdir('/Users/michael/testdir')

把两个路径合成一个时，不要直接拼字符串，而要通过os.path.join()函数，这样可以正确处理不同操作系统的路径分隔符。在Linux/Unix/Mac下，os.path.join()返回这样的字符串

part-1/part-2

Windows下会返回这样的字符串：

part-1\part-2

样的道理，要拆分路径时，也不要直接去拆字符串，而要通过os.path.split()函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

>>> os.path.split('/Users/michael/testdir/file.txt')

('/Users/michael/testdir', 'file.txt')

s.path.splitext()可以直接让你得到文件扩展名，很多时候非常方便

>>> os.path.splitext('/path/to/file.txt')

('/path/to/file', '.txt')

文件操作使用下面的函数。假定当前目录下有一个test.txt文件

*# 对文件重命名:*

>>> os.rename('test.txt', 'test.py')

*# 删掉文件:*

>>> os.remove('test.py')

利用Python的特性来过滤文件。比如我们要列出当前目录下的所有目录，只需要一行代码

>>> [x **for** x **in** os.listdir('.') **if** os.path.isdir(x)]

['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash', '.vim', 'Applications', 'Desktop', ...]

要列出所有的.py文件，也只需一行代码

>>> [x **for** x **in** os.listdir('.') **if** os.path.isfile(x) **and** os.path.splitext(x)[1]=='.py']

['apis.py', 'config.py', 'models.py', 'pymonitor.py', 'test\_db.py', 'urls.py', 'wsgiapp.py']

4序列化

变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python提供了pickle模块来实现序列化

我们尝试把一个对象序列化并写入文件：

>>> **import** pickle

>>> d = dict(name='Bob', age=20, score=88)

>>> pickle.dumps(d)

b'\x80\x03}q\x00(X\x03\x00\x00\x00ageq\x01K\x14X\x05\x00\x00\x00scoreq\x02KXX\x04\x00\x00\x00nameq\x03X\x03\x00\x00\x00Bobq\x04u.'

pickle.dumps()方法把任意对象序列化成一个bytes，然后，就可以把这个bytes写入文件。或者用另一个方法pickle.dump()直接把对象序列化后写入一个file-like Object

>>> f = open('dump.txt', 'wb')

>>> pickle.dump(d, f)

>>> f.close()

当我们要把对象从磁盘读到内存时，可以先把内容读到一个bytes，然后用pickle.loads()方法反序列化出对象，也可以直接用pickle.load()方法从一个file-like Object中直接反序列化出对象。我们打开另一个Python命令行来反序列化刚才保存的对象：

>>> f = open('dump.txt', 'rb')

>>> d = pickle.load(f)

>>> f.close()

>>> d

{'age': 20, 'score': 88, 'name': 'Bob'}

1.1JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式, 比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

Python内置的json模块提供了非常完善的Python对象到JSON格式的转换。我们先看看如何把Python对象变成一个JSON

>>> **import** json

>>> d = dict(name='Bob', age=20, score=88)

>>> json.dumps(d)

'{"age": 20, "score": 88, "name": "Bob"}'

dumps()方法返回一个str，内容就是标准的JSON。类似的，dump()方法可以直接把JSON写入一个file-like Object

要把JSON反序列化为Python对象，用loads()或者对应的load()方法，前者把JSON的字符串反序列化，后者从file-like Object中读取字符串并反序列化

>>> json\_str = '{"age": 20, "score": 88, "name": "Bob"}'

>>> json.loads(json\_str)

{'age': 20, 'score': 88, 'name': 'Bob'}

由于JSON标准规定JSON编码是UTF-8，所以我们总是能正确地在Python的str与JSON的字符串之间转换

1.2JSON进阶

Python的dict对象可以直接序列化为JSON的{}，不过，很多时候，我们更喜欢用class表示对象，比如定义Student类，然后序列化

**import** json

**class Student(object):**

**def** **\_\_init\_\_**(self, name, age, score):

self.name = name

self.age = age

self.score = score

s = Student('Bob', 20, 88)

print(json.dumps(s))

运行代码，毫不留情地得到一个TypeError

Traceback (most recent **call** **last**):

...

TypeError: <\_\_main\_\_.Student object **at** 0x10603cc50> **is** **not** JSON serializable

错误的原因是Student对象不是一个可序列化为JSON的对象。

如果连class的实例对象都无法序列化为JSON，这肯定不合理！

别急，我们仔细看看dumps()方法的参数列表，可以发现，除了第一个必须的obj参数外，dumps()方法还提供了一大堆的可选参数。

这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把Student类实例序列化为JSON，是因为默认情况下，dumps()方法不知道如何将Student实例变为一个JSON的{}对象。

可选参数default就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为Student专门写一个转换函数，再把函数传进去即可

**def** **student2dict**(std):

**return** {

'name': std.name,

'age': std.age,

'score': std.score

}

这样，Student实例首先被student2dict()函数转换成dict，然后再被顺利序列化为JSON

>>> **print**(json.dumps(s, **default**=student2dict))

{"age": 20, "name": "Bob", "score": 88}

不过，下次如果遇到一个Teacher类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意class的实例变为dict：

**print**(json.dumps(s, **default**=lambda obj: obj.\_\_dict\_\_))

因为通常class的实例都有一个\_\_dict\_\_属性，它就是一个dict，用来存储实例变量。也有少数例外，比如定义了\_\_slots\_\_的class。

同样的道理，如果我们要把JSON反序列化为一个Student对象实例，loads()方法首先转换出一个dict对象，然后，我们传入的object\_hook函数负责把dict转换为Student实例：

**def** **dict2student**(d):

**return** Student(d['name'], d['age'], d['score'])

运行结果如下

>>> json\_str = '{"age": 20, "score": 88, "name": "Bob"}'

>>> print(json.loads(json\_str, object\_hook=dict2student))

<\_\_main\_\_.Student object at 0x10cd3c190>

进程与线程

对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个Word就启动了一个Word进程。

有些进程还不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

我们要同时执行多个任务怎么办？

有两种解决方案：

一种是启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。

还有一种方法是启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。

当然还有第三种方法，就是启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有3种方式：

* 多进程模式；
* 多线程模式；
* 多进程+多线程模式。

1多进程

nix/Linux操作系统提供了一个fork()系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是fork()调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回0，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用getppid()就可以拿到父进程的ID。

Python的os模块封装了常见的系统调用，其中就包括fork，可以在Python程序中轻松创建子进程

**import** os

print('Process (%s) start...' % os.getpid())

*# Only works on Unix/Linux/Mac:*

pid = os.fork()

**if** pid == 0:

print('I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid()))

**else**:

print('I (%s) just created a child process (%s).' % (os.getpid(), pid))

运行结果如下：

Process (876) **start**...

I (876) just created a child process (877).

I am child process (877) **and** my parent **is** 876

有了fork调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的Apache服务器就是由父进程监听端口，每当有新的http请求时，就fork出子进程来处理新的http请求。

1.1multiprocessing

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。multiprocessing模块就是跨平台版本的多进程模块

multiprocessing模块提供了一个Process类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结束

**from** multiprocessing **import** Process

**import** os

*# 子进程要执行的代码*

**def** **run\_proc**(name):

print('Run child process %s (%s)...' % (name, os.getpid()))

**if** \_\_name\_\_=='\_\_main\_\_':

print('Parent process %s.' % os.getpid())

p = Process(target=run\_proc, args=('test',))

print('Child process will start.')

p.start()

p.join()

print('Child process end.')

执行结果如下：

Parent process 928.

Process will **start**.

Run child process test (929)...

Process **end**.

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动，这样创建进程比fork()还要简单。

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步

1.2Pool:如果要启动大量的子进程，可以用进程池的方式批量创建子进程

**from** multiprocessing **import** Pool

**import** os, time, random

**def** **long\_time\_task**(name):

print('Run task %s (%s)...' % (name, os.getpid()))

start = time.time()

time.sleep(random.random() \* 3)

end = time.time()

print('Task %s runs %0.2f seconds.' % (name, (end - start)))

**if** \_\_name\_\_=='\_\_main\_\_':

print('Parent process %s.' % os.getpid())

p = Pool(4)

**for** i **in** range(5):

p.apply\_async(long\_time\_task, args=(i,))

print('Waiting for all subprocesses done...')

p.close()

p.join()

print('All subprocesses done.')

执行结果如下：

**Parent** process 669.

Waiting **for** all subprocesses done...

Run task 0 (671)...

Run task 1 (672)...

Run task 2 (673)...

Run task 3 (674)...

Task 2 runs 0.14 seconds.

Run task 4 (673)...

Task 1 runs 0.27 seconds.

Task 3 runs 0.86 seconds.

Task 0 runs 1.41 seconds.

Task 4 runs 1.91 seconds.

All subprocesses done.

代码解读：

对Pool对象调用join()方法会等待所有子进程执行完毕，调用join()之前必须先调用close()，调用close()之后就不能继续添加新的Process了。

请注意输出的结果，task 0，1，2，3是立刻执行的，而task 4要等待前面某个task完成后才执行，这是因为Pool的默认大小在我的电脑上是4，因此，最多同时执行4个进程。这是Pool有意设计的限制，并不是操作系统的限制。如果改成：

p = Pool(5)

就可以同时跑5个进程,Pool的默认大小是CPU的核数

1.3子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。

subprocess模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。

下面的例子演示了如何在Python代码中运行命令nslookup www.python.org，这和命令行直接运行的效果是一样的

**import** subprocess

print('$ nslookup www.python.org')

r = subprocess.call(['nslookup', 'www.python.org'])

print('Exit code:', r)

如果子进程还需要输入，则可以通过communicate()方法输入

**import** subprocess

print('$ nslookup')

p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

output, err = p.communicate(b'set q=mx\npython.org\nexit\n')

print(output.decode('utf-8'))

print('Exit code:', p.returncode)

上面的代码相当于在命令行执行命令nslookup，然后手动输入

**set** q=mx

python.org

exit

1.4进程间通信

Process之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python的multiprocessing模块包装了底层的机制，提供了Queue、Pipes等多种方式来交换数据。

我们以Queue为例，在父进程中创建两个子进程，一个往Queue里写数据，一个从Queue里读数据

**from** multiprocessing **import** Process, Queue

**import** os, time, random

# 写数据进程执行的代码:

**def** **write**(q):

print('Process to write: %s' % os.getpid())

**for** value **in** ['A', 'B', 'C']:

print('Put %s to queue...' % value)

q.put(value)

time.sleep(random.random())

# 读数据进程执行的代码:

**def** **read**(q):

print('Process to read: %s' % os.getpid())

**while** True:

value = q.get(True)

print('Get %s from queue.' % value)

**if** \_\_name\_\_=='\_\_main\_\_':

# 父进程创建Queue，并传给各个子进程：

q = Queue()

pw = Process(target=write, args=(q,))

pr = Process(target=read, args=(q,))

# 启动子进程pw，写入:

pw.start()

# 启动子进程pr，读取:

pr.start()

# 等待pw结束:

pw.join()

# pr进程里是死循环，无法等待其结束，只能强行终止:

pr.terminate()

运行结果如下：

Process to write: 50563

Put A to queue...

Process to read: 50564

Get A **from** queue.

Put B to queue...

Get B **from** queue.

Put C to queue...

Get C **from** queue.

2多线程

Python的标准库提供了两个模块：\_thread和threading，\_thread是低级模块，threading是高级模块，对\_thread进行了封装。绝大多数情况下，我们只需要使用threading这个高级模块。

启动一个线程就是把一个函数传入并创建Thread实例，然后调用start()开始执行：

**import** time, threading

# 新线程执行的代码:

**def** **loop**():

print('thread %s is running...' % threading.current\_thread().name)

n = 0

**while** n < 5:

n = n + 1

print('thread %s >>> %s' % (threading.current\_thread().name, n))

time.sleep(1)

print('thread %s ended.' % threading.current\_thread().name)

print('thread %s is running...' % threading.current\_thread().name)

t = threading.Thread(target=loop, name='LoopThread')

t.start()

t.join()

print('thread %s ended.' % threading.current\_thread().name)

执行结果如下：

thread MainThread **is** running...

thread LoopThread **is** running...

thread LoopThread >>> 1

thread LoopThread >>> 2

thread LoopThread >>> 3

thread LoopThread >>> 4

thread LoopThread >>> 5

thread LoopThread ended.

thread MainThread ended.

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python的threading模块有个current\_thread()函数，它永远返回当前线程的实例。主线程实例的名字叫MainThread，子线程的名字在创建时指定，我们用LoopThread命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，如果不起名字Python就自动给线程命名为Thread-1，Thread-2

1.1Lock

多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了

如果我们要确保balance计算正确，就要给change\_it()上一把锁，当某个线程开始执行change\_it()时，我们说，该线程因为获得了锁，因此其他线程不能同时执行change\_it()，只能等待，直到锁被释放后，获得该锁以后才能改。由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过threading.Lock()来实现

balance = 0

lock = threading.Lock()

**def** **run\_thread**(n):

**for** i **in** range(100000):

*# 先要获取锁:*

lock.acquire()

**try**:

*# 放心地改吧:*

change\_it(n)

**finally**:

*# 改完了一定要释放锁:*

lock.release()

3ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而全局变量的修改必须加锁。

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

ThreadLocal应运而生，不用查找dict，ThreadLocal帮你自动做这件事

**import** threading

*# 创建全局ThreadLocal对象:*

local\_school = threading.local()

**def** **process\_student**():

*# 获取当前线程关联的student:*

std = local\_school.student

print('Hello, %s (in %s)' % (std, threading.current\_thread().name))

**def** **process\_thread**(name):

*# 绑定ThreadLocal的student:*

local\_school.student = name

process\_student()

t1 = threading.Thread(target= process\_thread, args=('Alice',), name='Thread-A')

t2 = threading.Thread(target= process\_thread, args=('Bob',), name='Thread-B')

t1.start()

t2.start()

t1.join()

t2.join()

执行结果：

Hello, Alice (**in** Thread-A)

Hello, Bob (**in** Thread-B)

全局变量local\_school就是一个ThreadLocal对象，每个Thread对它都可以读写student属性，但互不影响。你可以把local\_school看成全局变量，但每个属性如local\_school.student都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题，ThreadLocal内部会处理。

可以理解为全局变量local\_school是一个dict，不但可以用local\_school.student，还可以绑定其他变量，如local\_school.teacher等等。

ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

4进程vs线程

5分布式进程

Python的multiprocessing模块不但支持多进程，其中managers子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于managers模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

举个例子：如果我们已经有一个通过Queue通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务的进程分布到两台机器上。怎么用分布式进程实现？

原有的Queue可以继续使用，但是，通过managers模块把Queue通过网络暴露出去，就可以让其他机器的进程访问Queue了。

我们先看服务进程，服务进程负责启动Queue，把Queue注册到网络上，然后往Queue里面写入任务：

# task\_master.py

**import** random, time, queue

**from** multiprocessing.managers **import** BaseManager

# 发送任务的队列:

task\_queue = queue.Queue()

# 接收结果的队列:

result\_queue = queue.Queue()

# 从BaseManager继承的QueueManager:

**class QueueManager(BaseManager):**

**pass**

# 把两个Queue都注册到网络上, callable参数关联了Queue对象:

QueueManager.register('get\_task\_queue', callable=**lambda**: task\_queue)

QueueManager.register('get\_result\_queue', callable=**lambda**: result\_queue)

# 绑定端口5000, 设置验证码'abc':

manager = QueueManager(address=('', 5000), authkey=b'abc')

# 启动Queue:

manager.start()

# 获得通过网络访问的Queue对象:

task = manager.get\_task\_queue()

result = manager.get\_result\_queue()

# 放几个任务进去:

**for** i **in** range(10):

n = random.randint(0, 10000)

print('Put task %d...' % n)

task.put(n)

# 从result队列读取结果:

print('Try get results...')

**for** i **in** range(10):

r = result.get(timeout=10)

print('Result: %s' % r)

# 关闭:

manager.shutdown()

print('master exit.')

请注意，当我们在一台机器上写多进程程序时，创建的Queue可以直接拿来用，但是，在分布式多进程环境下，添加任务到Queue不可以直接对原始的task\_queue进行操作，那样就绕过了QueueManager的封装，必须通过manager.get\_task\_queue()获得的Queue接口添加。

然后，在另一台机器上启动任务进程（本机上启动也可以）：

# task\_worker.py

**import** time, sys, queue

**from** multiprocessing.managers **import** BaseManager

# 创建类似的QueueManager:

**class QueueManager(BaseManager):**

**pass**

# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:

QueueManager.register('get\_task\_queue')

QueueManager.register('get\_result\_queue')

# 连接到服务器，也就是运行task\_master.py的机器:

server\_addr = '127.0.0.1'

print('Connect to server %s...' % server\_addr)

# 端口和验证码注意保持与task\_master.py设置的完全一致:

m = QueueManager(address=(server\_addr, 5000), authkey=b'abc')

# 从网络连接:

m.connect()

# 获取Queue的对象:

task = m.get\_task\_queue()

result = m.get\_result\_queue()

# 从task队列取任务,并把结果写入result队列:

**for** i **in** range(10):

**try**:

n = task.get(timeout=1)

print('run task %d \* %d...' % (n, n))

r = '%d \* %d = %d' % (n, n, n\*n)

time.sleep(1)

result.put(r)

**except** Queue.Empty:

print('task queue is empty.')

# 处理结束:

print('worker exit.')

任务进程要通过网络连接到服务进程，所以要指定服务进程的IP。

现在，可以试试分布式进程的工作效果了。先启动task\_master.py服务进程：

$ python3 task\_master.py

Put task 3411...

Put task 1605...

Put task 1398...

Put task 4729...

Put task 5300...

Put task 7471...

Put task 68...

Put task 4219...

Put task 339...

Put task 7866...

Try get results...

task\_master.py进程发送完任务后，开始等待result队列的结果。现在启动task\_worker.py进程：

$ python3 task\_worker.py

Connect to server 127.0.0.1...

run task 3411 \* 3411...

run task 1605 \* 1605...

run task 1398 \* 1398...

run task 4729 \* 4729...

run task 5300 \* 5300...

run task 7471 \* 7471...

run task 68 \* 68...

run task 4219 \* 4219...

run task 339 \* 339...

run task 7866 \* 7866...

worker exit.

task\_worker.py进程结束，在task\_master.py进程中会继续打印出结果：

Result: 3411 \* 3411 = 11634921

Result: 1605 \* 1605 = 2576025

Result: 1398 \* 1398 = 1954404

Result: 4729 \* 4729 = 22363441

Result: 5300 \* 5300 = 28090000

Result: 7471 \* 7471 = 55815841

Result: 68 \* 68 = 4624

Result: 4219 \* 4219 = 17799961

Result: 339 \* 339 = 114921

Result: 7866 \* 7866 = 61873956

这个简单的Master/Worker模型有什么用？其实这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算n\*n的代码换成发送邮件，就实现了邮件队列的异步发送。

正则式

在正则表达式中，如果直接给出字符，就是精确匹配。用\d可以匹配一个数字，\w可以匹配一个字母或数字

.可以匹配任意字符，要匹配变长的字符，在正则表达式中，用\*表示任意个字符（包括0个），用+表示至少一个字符，用?表示0个或1个字符，用{n}表示n个字符，用{n,m}表示n-m个字符

\d{3}\s+\d{3,8}。

我们来从左到右解读一下：

1. \d{3}表示匹配3个数字，例如'010'；
2. \s可以匹配一个空格（也包括Tab等空白符），所以\s+表示至少有一个空格，例如匹配' '，' '等；
3. \d{3,8}表示3-8个数字，例如'1234567'。

1.1进阶

做更精确地匹配，可以用[]表示范围，比如：

* [0-9a-zA-Z\\_]可以匹配一个数字、字母或者下划线；
* [0-9a-zA-Z\\_]+可以匹配至少由一个数字、字母或者下划线组成的字符串，比如'a100'，'0\_Z'，'Py3000'等等；
* [a-zA-Z\\_][0-9a-zA-Z\\_]\*可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；
* [a-zA-Z\\_][0-9a-zA-Z\\_]{0, 19}更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

A|B可以匹配A或B，所以(P|p)ython可以匹配'Python'或者'python'。

^表示行的开头，^\d表示必须以数字开头。

$表示行的结束，\d$表示必须以数字结束。

你可能注意到了，py也可以匹配'python'，但是加上^py$就变成了整行匹配，就只能匹配'py'了。

1.2Re模块：Python提供re模块，包含所有正则表达式的功能

使用Python的r前缀，就不用考虑转义的问题了

s = r'ABC\-001' *# Python的字符串*

*# 对应的正则表达式字符串不变：*

*# 'ABC\-001'*

先看看如何判断正则表达式是否匹配：

>>> **import** re

>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')

<\_sre.SRE\_Match object; span=(0, 9), match='010-12345'>

>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')

>>>

match()方法判断是否匹配，如果匹配成功，返回一个Match对象，否则返回None。常见的判断方法就是：

test = '用户输入的字符串'

**if** re.match(r'正则表达式', test):

print('ok')

**else**:

print('failed')

1.3切片字符串

>>> re.split(r'[\s\,\;]+', 'a,b;; c d')

['a', 'b', 'c', 'd']

1.4分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用()表示的就是要提取的分组（Group）

>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')

>>> m

<\_sre.SRE\_Match object; span=(0, 9), match='010-12345'>

>>> m.group(0)

'010-12345'

>>> m.group(1)

'010'

>>> m.group(2)

'12345'

1.5贪婪匹配

正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的0：

>>> re.match(r'^(\d+)(0\*)$', '102300').groups()

('102300', '')

由于\d+采用贪婪匹配，直接把后面的0全部匹配了，结果0\*只能匹配空字符串了。

必须让\d+采用非贪婪匹配（也就是尽可能少匹配），才能把后面的0匹配出来，加个?就可以让\d+采用非贪婪匹配：

>>> re.match(r'^(\d+?)(0\*)$', '102300').groups()

('1023', '00')

1.6编译

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

>>> **import** re

# 编译:

>>> re\_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')

# 使用：

>>> re\_telephone.match('010-12345').groups()

('010', '12345')

>>> re\_telephone.match('010-8086').groups()

('010', '8086')