>>> print('包含中文的 str')

包含中文的 str

>>> print('''line1

... line2

... line3''')

line1

line2

line3

以上为输出语法

and与 or或 not非

逻辑真 输出true 假输出false

|  |  |  |
| --- | --- | --- |
| /除 | //整除 | %取余 |
| \*乘 | \*\*幂 |  |

>>> ord('A')

65

>>> chr(66)

'B'

>>> 'ABC'.encode('ascii')

b'ABC'

>>> '中文'.encode('utf-8')

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

>>> '中文'.encode('ascii')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:

ordinal not in range(128) ---报错：汉字无asc格式

输入.encode（输出格式）

常用格式有gbk utf-8 ascii gb2312

>>> len('中文')

2

>>> L = []

>>> len(L)

0

输出长度

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

第一行注释是为了告诉 Linux/OS X 系统，这是一个 Python 可执行程序，

Windows 系统会忽略这个注释；

第二行注释是为了告诉 Python 解释器，按照 UTF-8 编码读取源代码，

否则，你在源代码中写的中文输出可能会有乱码

>>> 'Hello, %s' % 'world'

'Hello, world'

%d 整数

%f 浮点数

%s 字符串

%x 十六进制整数

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

>>> classmates

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

>>> len(classmates)

3

>>> classmates[0]

'Michael'

>>> classmates[-1]

'Tracy'

存在类似数组的格式下面称为列表

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

>>> classmates.append('Adam')

>>> classmates

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

>>> classmates.insert(1, 'Jack')

>>> classmates

['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']

>>> classmates.pop()

'Adam'

>>> classmates

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

>>> classmates.pop(1)

'Jack'

>>> classmates

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

append可在列表添加末尾项

pop内无参数删去末尾项

有参数时删除指定项

insert在指定出插入项

>>> s = ['python', 'java', ['asp', 'php'], 'scheme']

>>> len(s)

4

等价于

>>> p = ['asp', 'php']

>>> s = ['python', 'java', p, 'scheme']

>>> t = (1, 2)

>>> t

(1, 2)

>>> t = ()

>>> t

()

>>> t = (1)

>>> t

1

>>> t = (1,)

>>> t

(1,)

>>> t = ('a', 'b', ['A', 'B'])

>>> t[2][0] = 'X'

>>> t[2][1] = 'Y'

>>> t

('a', 'b', ['X', 'Y'])

以括号()同类似于列表[]但实为元祖与以上的列表有区别的是元素无法修改

age = 3

if age >= 18:

print('adult')

elif age >= 6:

print('teenager')

else:

print('kid')

if语句的基本使用方式

>>> list(range(5))

[0, 1, 2, 3, 4]

range（x）为从0到x-1的列表

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

>>> d['Michael']

95

>>> d.pop('Bob')

75

>>> d

{'Michael': 95, 'Tracy': 85}

以括号{}同类似于列表[]元组()为字典

字典的pop置参数同为减去

>>> 'Thomas' in d

False

>>> d.get('Thomas')

>>> d.get('Thomas', -1)

-1

in函数确保是否存在于字典内

get是字典提供的方法如果值不存在返还none也可以自己设置参数为其输出的值

注意：返回 None 的时候 Python 的交互式命令行不显示结果

字典有以下几个特点：

1. 查找和插入的速度极快，不会随着 值 的增加而增加；

2. 需要占用大量的内存，内存浪费多。

而 列表 相反：

1. 查找和插入的时间随着元素的增加而增加；

2. 占用空间小，浪费内存很少。

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

>>> s

{1, 2, 3}

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

>>> s

{1, 2, 3}

set和字典类似也是一组组合但是不存值并且参不能重复

创建一个set要提供一个列表为输入集合

其显示的元素不会重复并且并非有序

>>> s.add(4)

>>> s

{1, 2, 3, 4}

>>> s.add(4)

>>> s

{1, 2, 3, 4}

>>> s.remove(4)

>>> s

{1, 2, 3}

add函数是添加元素到set里

可以重复添加但是无果

remove函数是删除指定元素

>>> s1 = set([1, 2, 3])

>>> s2 = set([2, 3, 4])

>>> s1 & s2

{2, 3}

>>> s1 | s2

{1, 2, 3, 4}

set也存在交集并集等操作

>>> a = ['c', 'b', 'a']

>>> a.sort()

>>> a

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

sort可以进行排序

>>> a = 'abc'

>>> a.replace('a', 'A')

'Abc'

>>> a

'abc'

replace可以使指定字符串的指定字符替换成指定字符

但是原指定字符串不会发生改变

>>> abs(100)

100

>>> abs(-20)

20

>>> abs(12.34)

12.34

abs函数使指定数字加上绝对值

>>> max(1, 2)

2

>>> max(2, 3, 1, -5)

3

max取最大值

>>> int('123')

123

>>> int(12.34)

12

>>> float('12.34')

12.34

>>> str(1.23)

'1.23'

>>> str(100)

'100'

>>> bool(1)

True

>>> bool('')

False

int即化为整形

float 单精度

str 字符串

bool 逻辑型

def my\_abs(x):

        if x >= 0:

                return x

        else:

                return -x

>>> my\_abs('A')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 2, in my\_abs

TypeError: unorderable types: str() >= int()

>>> abs('A')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: bad operand type for abs(): 'str'

传入函数数值时可能会发生类型错误

def my\_abs(x):

        if not isinstance(x, (int, float)):

                raise TypeError('bad operand type')

        if x >= 0:

                return x

        else:

                return -x

>>> my\_abs('A')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in my\_abs

TypeError: bad operand type

数据类型检查可以用内置函数 isinstance() 实现

>>> my\_abs('A')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in my\_abs

TypeError: bad operand type

添加参数检查 传入错误的参数类型 函数就可以输出错

误

import math                ————导入math库

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(100, 100, 60, math.pi / 6)

>>> print(x, y)

151.96152422706632 70.0

>>> r = move(100, 100, 60, math.pi / 6)

>>> print(r)

(151.96152422706632, 70.0)

由此得出函数可以有多个返回值

>>> import math

>>> math.sqrt(2)

1.4142135623730951

sqrt为使指定数开根值

def power(x, n=2):

        s = 1

        while n > 0:

                n = n - 1

                s = s \* x

        return s

>>> power(5)

25

>>> power(5, 2)

25

函数也可以设定默认值

def add\_end(L=[]):

        L.append('END')

        return L

>>> add\_end([1, 2, 3])

[1, 2, 3, 'END']

>>> add\_end(['x', 'y', 'z'])

['x', 'y', 'z', 'END']

>>> add\_end()

['END']

>>> add\_end()

['END', 'END']

>>> add\_end()

['END', 'END', 'END']

函数也可以处理列表

def calc(numbers):

        sum = 0

        for n in numbers:

                sum = sum + n \* n

        return sum

>>> calc([1, 2, 3])

14

>>> calc((1, 3, 5, 7))

84

函数同时也可以处理元组

def calc(\*numbers):

        sum = 0

        for n in numbers:

                sum = sum + n \* n

        return sum

>>> calc(1, 2)

5

>>> calc()

0

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

>>> calc(nums[0], nums[1], nums[2])

14

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

>>> calc(\*nums)

14

形参加入\*之后即可传入

列表元组的元素可变成参数传入

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

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

>>> person('Michael', 30)

name: Michael age: 30 other: {}

>>> person('Bob', 35, city='Beijing')

name: Bob age: 35 other: {'city': 'Beijing'}

>>> person('Adam', 45, gender='M', job='Engineer')

name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}

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

>>> person('Jack', 24, city=extra['city'], job=extra['job'])

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

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

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

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

\*\*参数为关键字参数可使0个或任意个参数

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

        if 'city' in kw:

                # 有 city 参数

                pass

        if 'job' in kw:

                # 有 job 参数

                pass

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

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

        print(name, age, city, job)

>>> person('Jack', 24, city='Beijing', job='Engineer')

Jack 24 Beijing Engineer

>>> person('Jack', 24, 'Beijing', 'Engineer')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: person() takes 2 positional arguments but 4 were given

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

        print(name, age, city, job)

>>> person('Jack', 24, job='Engineer')

Jack 24 Beijing Engineer

# 缺少 \*，city 和 job 被视为位置参数

命名关键字参数

def f1(a, b, c=0, \*args, \*\*kw):

        print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

def f2(a, b, c=0, \*, d, \*\*kw):

        print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)

>>> f1(1, 2)

a = 1 b = 2 c = 0 args = () kw = {}

>>> f1(1, 2, c=3)

a = 1 b = 2 c = 3 args = () kw = {}

>>> f1(1, 2, 3, 'a', 'b')

a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}

>>> f1(1, 2, 3, 'a', 'b', x=99)

a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}

>>> f2(1, 2, d=99, ext=None)

a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}

\*存列表\*\*存字典

因而可接入任意参数

>>> args = (1, 2, 3, 4)

>>> kw = {'d': 99, 'x': '#'}

>>> f1(\*args, \*\*kw)

a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}

>>> args = (1, 2, 3)

>>> kw = {'d': 88, 'x': '#'}

>>> f2(\*args, \*\*kw)

a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}

def fact(n):

        if n==1:

                return 1

        return n \* fact(n - 1)

>>> fact(5)

120

===> fact(5)

===> 5 \* fact(4)

===> 5 \* (4 \* fact(3))

===> 5 \* (4 \* (3 \* fact(2)))

===> 5 \* (4 \* (3 \* (2 \* fact(1))))

===> 5 \* (4 \* (3 \* (2 \* 1)))

===> 5 \* (4 \* (3 \* 2))

===> 5 \* (4 \* 6)

===> 5 \* 24

===> 120

递归

>>> fact(1000)                        ————栈溢出

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 4, in fact

...

File "<stdin>", line 4, in fact

RuntimeError: maximum recursion depth exceeded in comparison

栈具有一定大小数值过大会导致溢出错误

def fact(n):

        return fact\_iter(n, 1)

def fact\_iter(num, product):

        if num == 1:

                return product

        return fact\_iter(num - 1, num \* product)

===> fact\_iter(5, 1)

===> fact\_iter(4, 5)

===> fact\_iter(3, 20)

===> fact\_iter(2, 60)

===> fact\_iter(1, 120)

===> 120

优化递归算法之后可以使函数嵌套

代码不是越多越好，而是越少越好。

代码不是越复杂越好，而是越简单越好。

>>> L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']

>>> [L[0], L[1], L[2]]

['Michael', 'Sarah', 'Tracy']

>>> r = []

>>> n = 3

>>> for i in range(n):

... r.append(L[i])

...

>>> r

['Michael', 'Sarah', 'Tracy']

>>> L[0:3]

['Michael', 'Sarah', 'Tracy']

>>> L[:3]

['Michael', 'Sarah', 'Tracy']

>>> L[1:3]

['Sarah', 'Tracy']

>>> L[-2:]

['Bob', 'Jack']

>>> L[-2:-1]

['Bob']

列表内带冒号的括弧左开右闭

>>> L = list(range(100))

>>> L

[0, 1, 2, 3, ..., 99]

>>> L[:10]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> L[-10:]

[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]

>>> L[10:20]

[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

>>> L[:10:2]

[0, 2, 4, 6, 8]

>>> L[::5]

[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85,

90, 95]

>>> L[:]

[0, 1, 2, 3, ..., 99]

后缀即切片 第三参为间距

range(x:y:z)x启到y-1间隔z

>>> (0, 1, 2, 3, 4, 5)[:3]

(0, 1, 2)

>>> 'ABCDEFG'[:3]

'ABC'

>>> 'ABCDEFG'[::2]

'ACEG'

>>> d = {'a': 1, 'b': 2, 'c': 3}

>>> for key in d:

... print(key)

...

a

c

b

for循环的值为其key

>>> for ch in 'ABC':

... print(ch)

...

A

B

C

for循环的基本用法

>>> from collections import Iterable

>>> isinstance('abc', Iterable)

# str 是否可迭代

True

>>> isinstance([1,2,3], Iterable)

# list 是否可迭代

True

>>> isinstance(123, Iterable)

# 整数是否可迭代

False

isinstance确认参数是否可迭代

迭代用重复结构，而递归用选择结构

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

... print(i, value)

...

0 A

1 B

2 C

类似于c++语言的枚举权值语法

>>> for x, y in [(1, 1), (2, 4), (3, 9)]:

... print(x, y)

...

1 1

2 4

3 9

for循环支持列表

>>> list(range(1, 11))

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

range在python3里代表列表但是不会实际举出只有使用list函数才会显示

>>> L = []

>>> for x in range(1, 11):

... L.append(x \* x)

...

>>> L

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

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

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

展示语法for 循环值 in 被循环主体

三参的for循环语法

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

[4, 16, 36, 64, 100]

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

实际的被循环主体受if语句限制

也可视为[x \* x for x in [2，4，6，8，10]]

依旧展示x\*x

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

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

for循环支持双循环语法

>>> import os # 导入 os 模块，模块的概念后面讲到

>>> [d for d in os.listdir('.')] # os.listdir 可以列出文件和目录

['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop',

'Documents', 'Downloads', 'Library', 'Movies', 'Music', 'Pictures',

'Public', 'VirtualBox VMs', 'Workspace', 'XCode']

猜测估计是os模块（库）中所有的定义函数

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

>>> for k, v in d.items():

... print(k, '=', v)

...

y = B

x = A

z = C

字典用for循环全部展示的方式函数items()

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

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

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

可以优化为以下输出方式 但输出格式有所不同

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

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

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

lower使字符串改为小写

>>> L = [x \* x for x in range(10)]

>>> L

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x \* x for x in range(10))        —未知错误

>>> g

<generator object <genexpr> at 0x1022ef630>

#generator是生成器

>>> next(g)

0

>>> next(g)

1

>>> next(g)

4

>>> next(g)

9

>>> next(g)

16

>>> next(g)

25

>>> next(g)

36

>>> next(g)

49

>>> next(g)

64

>>> next(g)

81

>>> next(g) —接下来未定义即参数溢出

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

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

>>> for n in g:

... print(n)

...

0

1

4

9

16

25

36

49

64

81

等同以上

def fib(max):

n, a, b = 0, 0, 1

while n < max:

print(b)

a, b = b, a + b

n = n + 1

return 'done'

>>> fib(6)

1

1

2

3

5

8

'done'

输出斐波那契数列的前 N 个数

def fib(max):

n, a, b = 0, 0, 1

while n < max:

yield b

a, b = b, a + b

n = n + 1

return 'done'

>>> f = fib(6)

>>> f

<generator object fib at 0x104feaaa0>

def odd():

print('step 1')

yield 1

print('step 2')

yield(3)

print('step 3')

yield(5)

>>> o = odd()

>>> next(o)

step 1

1

>>> next(o)

step 2

3

>>> next(o)

step 3

5

>>> next(o)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

generator遇到 yield就中断相当于return

>>> g = fib(6)

>>> while True:

... try:

... x = next(g)

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

... except StopIteration as e:

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

... break

...

g: 1

g: 1

g: 2

g: 3

g: 5

g: 8

Generator return value: done

迭代器

一类是集合数据类型，如 list 、 tuple 、 dict 、 set 、 str 等；

一类是 generator ，包括生成器和带 yield 的 generator function。

这些可以直接作用于 for 循环的对象统称为可迭代对象： Iterable

>>> from collections import Iterable

>>> isinstance([], Iterable)

True

>>> isinstance({}, Iterable)

True

>>> isinstance('abc', Iterable)

True

>>> isinstance((x for x in range(10)), Iterable)

True

>>> isinstance(100, Iterable)

False

可以使用 isinstance() 判断一个对象是否是 Iterable 对象

生成器不但可以作用于 for 循环

还可以被 next() 函数不断调用并返回下一个值

>>> from collections import Iterable

>>> isinstance([], Iterable)

True

>>> isinstance({}, Iterable)

True

>>> isinstance('abc', Iterable)

True

>>> isinstance((x for x in range(10)), Iterable)

rue

>>> isinstance(100, Iterable)

False

生成器不但可以作用于 for 循环

还可以被 next() 函数不断调用并返回下一个值

至最后抛出 StopIteration 错误表示无法继续返回下一个值

>>> from collections import Iterator

>>> isinstance((x for x in range(10)), Iterator)

True

>>> isinstance([], Iterator)

False

>>> isinstance({}, Iterator)

False

>>> isinstance('abc', Iterator)

False

生成器都是 Iterator 对象

但 list 、 dict 、 str 虽然是 Iterable却不是Iterator

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

>>> isinstance(iter([]), Iterator)

True

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

True

凡是可作用于 for 循环的对象都是 Iterable 类型；

凡是可作用于 next() 函数的对象都是 Iterator 类型，它们表示一个惰性

计算的序列；

集合数据类型如 list 、 dict 、 str 等是 Iterable 但不是 Iterator ，不过可

以通过 iter() 函数获得一个 Iterator 对象。

Python 的 for 循环本质上就是通过不断调用 next() 函数实现的，例如：

for x in [1, 2, 3, 4, 5]:

pass

实际上完全等价于：

# 首先获得 Iterator 对象:

it = iter([1, 2, 3, 4, 5])

# 循环:

while True:

try:

# 获得下一个值:

x = next(it)

except StopIteration:

# 遇到 StopIteration 就退出循环

break

>>> abs(-10)

10

>>> abs

<built-in function abs>

可见， abs(-10) 是函数调用，而 abs 是函数本身。

>>> f = abs

>>> f

<built-in function abs>

结论：函数本身也可以赋值给变量，即：变量可以指向函数。

>>> abs = 10

>>> abs(-10)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'int' object is not callable

函数名也是变量 但内置函数不应当被重新当变量赋值

只能重启python交互环境

注：由于 abs 函数实际上是定义在 \_\_builtin\_\_ 模块中的

所以要让修改abs 变量的指向在其它模块也生效

应 \_\_builtin\_\_.abs = 10

一个函数就可以接收另一个函数作为参数 被称为高阶函数

例：

def add(x, y, f):

return f(x) + f(y)

x = -5

y = 6

f = abs

f(x) + f(y) ==> abs(-5) + abs(6) ==> 11

return 11

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

11

编写高阶函数，就是让函数的参数能够接收别的函数

>>> def f(x):

... return x \* x

...

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

>>> list(r)

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

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

并返回Iterator

同样可以用循环表达

L = []

for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:

L.append(f(n))

print(L)

但是map函数更加直观可读性强

>>> list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

['1', '2', '3', '4', '5', '6', '7', '8', '9']

将数字列表转为字符串数字列表只要一行

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

reduce ()把结果继续和序列的下一个元素做累积计算的效果

>>> from functools import reduce

>>> def add(x, y):

... return x + y

...

>>> reduce(add, [1, 3, 5, 7, 9])

25

当然能可用sun()计算

>>> from functools import reduce

>>> def fn(x, y):

... return x \* 10 + y

...

>>> reduce(fn, [1, 3, 5, 7, 9])

13579

将列表化为数字

>>> from functools import reduce

>>> def fn(x, y):

... return x \* 10 + y

...

>>> def char2num(s):

... return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6,

'7': 7, '8': 8, '9': 9}[s]

...

>>> reduce(fn, map(char2num, '13579'))

13579

将str化为int函数

from functools import reduce

def str2int(s):

def fn(x, y):

return x \* 10 + y

def char2num(s):

return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6,

'7': 7, '8': 8, '9': 9}[s]

return reduce(fn, map(char2num, s))

还可以用 lambda 函数进一步简化成：

from functools import reduce

def char2num(s):

return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7':

7, '8': 8, '9': 9}[s]

def str2int(s):

return reduce(lambda x, y: x \* 10 + y, map(char2num, s))

把字符串转化为整数的函数

def is\_odd(n):

return n % 2 == 1

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

# 结果: [1, 5, 9, 15]即删掉偶数只保留奇数

def not\_empty(s):

return s and s.strip()

list(filter(not\_empty, ['A', '', 'B', None, 'C', ' ']))

# 结果: ['A', 'B', 'C']

filter()函数旨在于正确实现一个筛选函数

def \_odd\_iter():

n = 1

while True:

n = n + 2

yield n

注意这是一个生成器 并且是一个无限序列

def \_not\_divisible(n):

return lambda x: x % n > 0

def primes():

yield 2

it = \_odd\_iter() # 初始序列

while True:

n = next(it) # 返回序列的第一个数

yield n

it = filter(\_not\_divisible(n), it) # 构造新序列

该生成器返回第一个素数2然后不断产生筛选后的新序列

# 打印 1000 以内的素数:

for n in primes():

if n < 1000:

print(n)

else:

break

primes() 也是一个无限序列

所以调用时需要设置一个退出循环的条件

>>> sorted([36, 5, -12, 9, -21])

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

sorted() 函数就可以对 list 进行排序

>>> sorted([36, 5, -12, 9, -21], key=abs)

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

可自定义排序 例如绝对值大小

key 指定的函数将作用于 list 的每一个元素上

并根据 key 函数返回的结果进行排序

>>> sorted(['bob', 'about', 'Zoo', 'Credit'])

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

一般情况是按照ascii进行比较的

大写字母 Z 会排在小写字母 a 的前面

>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)

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

lower函数使其化为小写

最后以sorted函数进行ascii比较

>>> sorted(['bob', 'about','Zoo','Credit'],key=str.lower,reverse=True)

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

reverse=True目的是进行反向排序默认为false

def calc\_sum(\*args):

ax = 0

for n in args:

ax = ax + n

return ax

可变参数的求和

def lazy\_sum(\*args):

def sum():

ax = 0

for n in args:

ax = ax + n

return ax

return sum

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

>>> f

<function lazy\_sum.<locals>.sum at 0x101c6ed90>

>>> f()

25

调用函数 f 时才真正计算求和的结果

这种闭包的程序结构中相关参数和变量都保存在返回的函数

def count():

fs = []

for i in range(1, 4):

def f():

return i\*i

fs.append(f)

return fs

f1, f2, f3 = count()

>>> f1()

9

>>> f2()

9

>>> f3()

9

原因就在于返回的函数引用了变量 i 但它并非立刻执行

等到 3 个函数都返回时 它们所引用的变量 i 已经变成了 3

因而：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

def count():

def f(j):

def g():

return j\*j

return g

fs = []

for i in range(1, 4):

fs.append(f(i)) # f(i)立刻被执行，因此 i 的当前值被传入 f()

return fs

>>> f1, f2, f3 = count()

>>> f1()

1

>>> f2()

4

>>> f3()

9

只要记住函数返回值不要伴随循环或者会变的变量即可

另外该代码可以用lambda函数缩短代码

匿名函数 lambda x: x \* x 实际上就是

def f(x):

return x \* x

匿名函数没有函数名字因而不必担心函数名冲突

>>> f = lambda x: x \* x

>>> f

<function <lambda> at 0x101c6ef28>

>>> f(5)

25

另外匿名函数可以赋值给一个变量再利用变量来调用该函数

def build(x, y):

return lambda: x \* x + y \* y

匿名函数可以作为函数的返回值返回

>>> def now():

... print('2015-3-25')

...

>>> f = now

>>> f()

2015-3-25

如同上一例 可以赋值给变量运作函数

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

'now'

>>> f.\_\_name\_\_

'now'

函数对象有一个name属性可以拿到函数的名字

def log(func):

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

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

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

return wrapper

@log

def now():

print('2015-3-25')

>>> now()

call now():#打印的日志

2015-3-25

在代码运行期间动态增加功能的方式成为装饰器

调用函数会在函数前打印一行日志

@log 放到 now() 函数的定义处

相当于执行了语句now = log(now)

def log(text):

def decorator(func):

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

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

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

return wrapper

return decorator

@log('execute')

def now():

print('2015-3-25')

>>> now()

execute now():

2015-3-25

也就是相当于>>> now = log('execute')(now)

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

'wrapper'

这是要注意的地方

import functools

def log(func):

@functools.wraps(func)#解读略有困难

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

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

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

return wrapper

或是带有参数的

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

以上是两种完整的装饰器的写法

functools是一个模块 后细解

至此只需记住在定义 wrapper() 的前面加@functools.wraps(func)

>>> int('12345')

12345

偏函数基本用法

>>> int('12345', base=8)

5349#12345八进制转5349十进制

>>> int('12345', 16)

74565#12345十六进制转74565十进制

第二参数默认值是10即进制转换

def int2(x, base=2):

return int(x, base)

>>> int2('1000000')

64

>>> int2('1010101')

85

以此方法二进制转十进制

>>> import functools

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

>>> int2('1000000')

64

>>> int2('1010101')

85

functools.partial 是把一个函数的某些参数给固定住返回一个新的函数

模块最大程度上提高了代码的可维护性

模块编写完毕就可以被其他地方引用

使用模块可以避免函数名和变量名冲突

注意：尽量不要和内置函数名冲突

python内置函数链接：https://docs.python.org/3/library/functions.html

import 模块名字

即可引入模块

文件夹目录存放某模块名字

模块会变成：文件夹名称.模块名称

每一个包目录下面都会有一个 \_\_init\_\_.py 的文件

用来提示这是一个包

否则python会把这个目录当成普通目录

常用引入模块的方式 为 模块名字.模块内的函数（）

作用域等同于

类似 \_\_xxx\_\_ 这样的变量是特殊变量可以被直接引用但有特殊用途

>>> import sys

>>> sys.path.append('/Users/michael/my\_py\_scripts')

加入path运行结束之后失效

class Student(object):

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

self.name = name

self.score = score

def print\_score(self):

print('%s: %s' % (self.name, self.score))

以上是类的封装特性

类内的函数第一个参数必须是self

self类似于对象名引用对象内的特性如同引用模块内的函数

class Student(object):

...

def get\_grade(self):

if self.score >= 90:

return 'A'

elif self.score >= 60:

return 'B'

else:

return 'C'

得以发现封装特性是将数据和逻辑封装起来

class Student(object):

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

self.\_\_name = name

self.\_\_score = score

def print\_score(self):

print('%s: %s' % (self.\_\_name, self.\_\_score))

>>> bart = Student('Bart Simpson', 98)

>>> bart.\_\_name

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Student' object has no attribute '\_\_name'

使用私有变量时对外部代码来说没变动但已经无法从外部访问和修改变量了

class Student(object):

...

def get\_name(self):

return self.\_\_name

def get\_score(self):

return self.\_\_score

get\_name 和 get\_score这样的方法可以外部代码获取内部变量name和score

class Student(object):

...

def set\_score(self, score):

self.\_\_score = score

set\_score 方法可以外部代码修改score

但实际上直观来看与公共变量实际没有任何区别

class Student(object):

...

def set\_score(self, score):

if 0 <= score <= 100:

self.\_\_score = score

else:

raise ValueError('bad score')

因而实际上对次还是有主要原因是可以外部修改内部数据的时候防止传入无效参数

注：在python变量名类似\_\_XXX\_\_一般情况下是特殊变量。特殊变量是可以直接访问的

因而不能命名类似的变量名

有时见类似\_开头的变量名外部是可以访问的约定俗成于视为私有变量不要随意访问

遗憾的是python本身是没有任何机制避免破坏内部的既定变量因而需要注意

class Animal(object):

def run(self):

print('Animal is running...')

class Dog(Animal):

pass

dog = Dog()

dog.run()

Animal is running...

一般对于Dog来说Animal是它的父类也就是class类的继承性

启程特性最大的好处是子类获得了父类的全部功能

同时在运行代码的时候总会调用子类的功能就获得了继承中的一个特性多态性

a = list() # a 是 list 类型

b = Animal() # b 是 Animal 类型

c = Dog() # c 是 Dog 类型

>>> isinstance(a, list)

True

>>> isinstance(c, Animal)

True

>>> isinstance(b, Dog)

False

isinstance()用于判断变量a的类型是否为list(class）

发现由于继承性c不仅是Dog类型也是Animal类型

但是b并不会是Dog类型所以继承性只有单向性

def run\_twice(animal):#注:python的大小写是区分的此animal与Animal(class)无关

animal.run()

animal.run()

>>> run\_twice(Animal())

Animal is running...

Animal is running...

>>> run\_twice(Dog())

Dog is running...

Dog is running…

class Tortoise(Animal):

def run(self):

print('Tortoise is running slowly...')

>>> run\_twice(Tortoise())

Tortoise is running slowly...

Tortoise is running slowly...

会发现新增的一个子类不用对函数做修改也可正常运行因于多态

Dog和Tortoise都是Animal类型因而只要对Animal类型进行操作其本身或子类有人都可用

著名的“开闭”原则：

对扩展开放：允许新增 Animal 子类；

对修改封闭：不需要修改依赖 Animal 类型的 run\_twice() 等函数。

>>> type(123)

<class 'int'>

>>> type('str')

<class 'str'>

>>> type(None)

<type(None) 'NoneType'>

>>> type(abs)

<class 'builtin\_function\_or\_method'>

>>> type(a)

<class '\_\_main\_\_.Animal'>

type()可以判断对象类型

一个变量指向函数或者类也可以此判断

>>> type(123)==type(456)

True

>>> type(123)==int

True

>>> type('abc')==type('123')

True

>>> type('abc')==str

True

>>> type('abc')==type(123)

False

可以判断两种数据的类型是否相同

>>> import types

>>> def fn():

... pass

...

>>> type(fn)==types.FunctionType

True

>>> type(abs)==types.BuiltinFunctionType

True

>>> type(lambda x: x)==types.LambdaType

True

>>> type((x for x in range(10)))==types.GeneratorType

True

type()可以判断一个对象是否是函数

例继承关系是：object -> Animal -> Dog -> Husky

>>> a = Animal()

>>> d = Dog()

>>> h = Husky()

>>> isinstance(h, Husky) and isinstance(h, Dog) and isinstance(h, Animal)

True

>>> isinstance(d, Dog) and isinstance(d, Animal)

True

>>> isinstance(d, Husky)

False

>>> isinstance('a', str)

True

>>> isinstance(123, int)

True

>>> isinstance(b'a', bytes)

True

isinstance()函数判断变量的类型

由此也可以体现继承性的特性是包含关系

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

True

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

True

可以判断是否是 list 或者 tuple

>>> dir('ABC')

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_',

'\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_',

'\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_',

'\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_',

'\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_',

'\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_',

'\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count',

'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map',

'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',

'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',

'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',

'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',

'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',

'upper', 'zfill']

dir()函数返回的是一个列表为该对象的所有属性和方法

>>> len('ABC')

3

>>> 'ABC'.\_\_len\_\_()

3

这类方法的代码是等价的

>>> class MyDog(object):

... def \_\_len\_\_(self):

... return 100

...

>>> dog = MyDog()

>>> len(dog)

100

自己写的函数

>>> 'ABC'.lower()

'abc'

普通的方法

>>> 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

hasattr()函数测试是否有该属性

setattr()函数设置属性

getattr()函数获取属性

>>> getattr(obj, 'z') # 获取属性'z'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'MyObject' object has no attribute 'z'

获取的属性如果不存在会报错

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

404

也就可以用这种方式避免报错

>>> 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

也可以获得对象的方法

class Student(object):

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

self.name = name

s = Student('Bob')

s.score = 90

>>> 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

以此要注意命名实例属性和类属性不要相同

class Student(object):

pass

>>> s = Student()

>>> s.name = 'Michael' # 动态给实例绑定一个属性

>>> print(s.name)

Michael

————————或者是————————

>>> def set\_age(self, age): # 定义一个函数作为实例方法

... self.age = age

...

>>> from types import MethodType

>>> s.set\_age = MethodType(set\_age, s) # 给实例绑定一个方法

>>> s.set\_age(25) # 调用实例方法

>>> s.age # 测试结果

25

能够给实例绑定一个属性

>>> s2 = Student() # 创建新的实例

>>> s2.set\_age(25) # 尝试调用方法

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Student' object has no attribute 'set\_age'

但是另一个相同的类的不同变量没有定义之前是不能使用的

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

... self.score = score

...

>>> Student.set\_score = MethodType(set\_score, Student)#难点!!!

注：函数第一个参数传入的是函数的地址

也就是说一个类添加一个引入的函数的属性

类名.添加类内函数名=MethodType(要引入的函数(地址),类名)

给 class 绑定方法后，所有实例均可调用：

>>> s.set\_score(100)

>>> s.score

100

这是python动态性语言的特殊功能 静态语言中很难实现

class Student(object):

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

>>> s = Student() # 创建新的实例

>>> s.name = 'Michael' # 绑定属性'name'

>>> s.age = 25 # 绑定属性'age'

>>> s.score = 99 # 绑定属性'score'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Student' object has no attribute 'score'

报错的原因是由于\_\_slots\_\_的作用是限制class实例能添加的属性

由于score属性没有在允许绑定的属性名称内所以会报错

>>> class GraduateStudent(Student):

... pass

...

>>> g = GraduateStudent()

>>> g.score = 9999

会发现被限制类的子类是不受影响的

倘若要使得一个类其子类页同时受到限制 定义一个实力允许定义的属性限制

在类上使用 \_\_slots\_\_即可

class Student(object):

def get\_score(self):

return self.\_score

def set\_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

>>> s = Student()

>>> s.set\_score(60) # ok!

>>> s.get\_score()

60

>>> s.set\_score(9999)

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

会发现由定义类之后的数据无法对类的任意实例进行操作

因而可以保护数据不被不合逻辑的数据修改

class Student(object):

@property #难点!!

def score(self):

return self.\_score

@score.setter #获取值

def score(self, value):

if not isinstance(value, int):#定义获取的值非整形 报错

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

if value < 0 or value > 100:#定义获取的值越界 报错

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

self.\_score = value

>>> s = Student()

>>> s.score = 60 # OK，实际转化为 s.set\_score(60)

>>> s.score # OK，实际转化为 s.get\_score()

60

>>> s.score = 9999

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

看到@符号可以会想到之前装饰器（@log）也同是在函数前声明的

@property 装饰器就是负责把一个方法变成属性调用的

于是就由了对属性控制的操作了

class Student(object):

@property

def birth(self):

return self.\_birth

@birth.setter

def birth(self, value):

self.\_birth = value#指定输入的值使其可写

@property

def age(self):

return 2015 - self.\_birth

这是不定义赋值方法只可读的属性

brith是可读可写属性 age只可读

class Animal(object):

pass

class Mammal(Animal):

pass

class Runnable(object):

def run(self):

print('Running...')

class Dog(Mammal, Runnable):

pass

通过多重继承可以使得Dog获得多个父类所有的功能

class Dog(Mammal, RunnableMixIn, CarnivorousMixIn):

pass

这种设计成为MixIn让一个类继承多个父类的功能

这样设计组合的时候可以考虑用多种功能的方式继承分类而不是复杂的设计层次继承

>>> class Student(object):

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

... self.name = name

...

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

<\_\_main\_\_.Student object at 0x109afb190>

>>> 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)

\_\_str\_\_这样的定义便于显示内部只要==重要的数据

>>> s = Student('Michael')

>>> s

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

不用print时发现还是不够整洁

class Student(object):

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

self.name = name

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

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

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

\_\_repr\_\_ 是显示变量调用于str类似

\_\_str\_\_() 是返回用户看到的字符串

\_\_repr\_\_() 是返回程序开发者看到的字符串

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

这是以斐波那契数列为例的类用作for循环

>>> for n in Fib():

... print(n)

...

1

1

2

3

5

...

46368

75025

也就是说 \_\_iter\_\_是对自身的迭代而\_\_next\_\_是对自身的递推

raise StopIteration();也就是使自身错误从而退出循环

>>> Fib()[5]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'Fib' object does not support indexing

Fib类虽然能用作for循环但是不能把其当作一个列表

class Fib(object):

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

a, b = 1, 1

for x in range(n):

a, b = b, a + b

return a

>>> f = Fib()

>>> f[0]

1

>>> f[1]

1

>>> f[2]

2

>>> f[3]

3

>>> f[10]

89

>>> f[100]

573147844013817084101

\_\_getitem\_\_可是类的迭代变为一个可取制定项的列表

>>> list(range(100))[5:10]

[5, 6, 7, 8, 9]

这是列表的切片方法但在Fib类的情况下是无法执行的

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

>>> f = Fib()

>>> f[0:5]

[1, 1, 2, 3, 5]

>>> f[:10]

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

>>> f[:10:2]

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

在\_\_getitem\_\_的情况下对类形成的迭代列表的切片也是可行的

如果是字典 \_\_getitem\_\_() 的参数也可能是一个可以作key值

\_\_setitem\_\_() 方法对列表或字典进行集合赋值。

\_\_delitem\_\_() 方法是用于删除某个元素

class Student(object):

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

self.name = 'Michael'

>>> s = Student()

>>> print(s.name)

Michael

>>> print(s.score)

Traceback (most recent call last):

...

AttributeError: 'Student' object has no attribute 'score'

报错提示并没有找到score这一属性

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

当调用的属性不存在的时候会调用\_\_getattr\_\_来尝试获得属性从而返回score值

lass Student(object):

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

if attr=='age':

return lambda: 25

>>> s.age()

25

当然只有在没有找到属性的情况下才会调用\_getattr\_\_来尝试获得属性

倘若调用其他值会默认返回None（函数的return）

class Student(object):

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

if attr=='age':

return lambda: 25

raise AttributeError('\'Student\' object has no attribute\'%s\'' % attr)

为了使类只响应几个属性 这样设置可以同样报错

class Chain(object):

def \_\_init\_\_(self, path=''):

self.\_path = path

def \_\_getattr\_\_(self, path):

return Chain('%s/%s' % (self.\_path, path))

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

return self.\_path

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

>>> Chain().status.user.timeline.list

'/status/user/timeline/list'

这是一种链式调用可以应用于一些平台的url

class Student(object):

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

self.name = name

def \_\_call\_\_(self):

print('My name is %s.' % self.name)

>>> s = Student('Michael')

>>> s() # self 参数不要传入

My name is Michael.

\_\_call\_\_() 方法定义后就可以直接对实例进行调用

>>> callable(Student())

True

>>> callable(max)

True

>>> callable([1, 2, 3])

False

>>> callable(None)

False

>>> callable('str')

False

表 示 
描 
正 则 表 达 式 示 例 
述 
符 号 
匹 配 文 本 字 符 串 的 字 面 值 literal 
literal 
foo 
匹 配 正 则 表 达 式 “ 1 或 者 2 
foobar 
rel 2 
匹 配 任 何 字 符 （ 除 了 \ n 之 外 ） 
b.b 
匹 配 字 符 串 起 始 部 分 
ADear 
匹 配 字 符 串 终 止 部 分 
/bin/*sh$ 
匹 配 0 次 或 者 多 次 前 面 出 现 的 正 则 表 达 式 
[A-Za-z0-9] * 
匹 配 1 次 或 者 多 次 前 面 出 现 的 正 则 表 达 式 
[a-Z] 十\.com 
匹 配 0 次 或 者 I 次 前 面 出 现 的 正 则 表 达 式 
goo? 
匹 配 丿 V 次 前 面 出 现 的 正 则 表 达 式 
[ 0 ． 9 ] { 3 } 
匹 配 M 、 次 前 面 出 现 的 正 则 表 达 式 
[ 0 ． 9 ] { 5 ， 9 } 
匹 配 来 自 字 符 集 的 任 意 单 一 字 符 
[aeiou] 
匹 配 ． 丫 。 、 y 范 围 中 的 任 意 单 一 字 符 
[ 0 ． 9 ] ， [A-Za-z] 
不 匹 配 此 字 符 集 中 出 现 的 任 何 一 个 字 符 ， 包 括 某 一 范 围 的 字 符 （ 如 果 在 此 字 符 集 中 出 现 ） 
[Aaeiou] ， [AA-Za-z0-9] 
用 于 匹 配 上 面 频 繁 出 现 / 重 复 出 现 符 号 的 非 贪 婪 版 本 （ * 、 + 、 ？ 、 { } ） 
0 十 囝 { }) ？ 
匹 配 封 闭 的 正 则 表 达 式 ， 然 后 另 存 为 子 组 
([0-9]{3})?,f(ooIu)bar 

特 殊 字 符 
\d 
匹 配 任 何 十 进 制 数 字 ， 与 [ 0 ． 9 ] 一 致 (D 与 \ d 相 反 ， 不 匹 配 任 何 非 数 值 型 的 数 字 ） 
data\d 十 .txt 
匹 配 任 何 字 母 数 字 字 符 ， 与 [ A ． za ． zo ． 9 一 ] 相 同 ()W 与 之 相 反 ） 
匹 配 任 何 空 格 字 符 ， 与 № \ t \ \ v \ 月 相 同 ()S 与 之 相 反 ） 
oflsthe 
匹 配 任 何 单 词 边 界 ()B 与 之 相 反 ） 
\bThe\b 
匹 配 己 保 存 的 子 组 气 参 见 上 面 的 （ 一 ） ） 
price: \16 
逐 字 匹 配 任 何 特 殊 字 符 c （ 即 ， 仅 按 照 字 面 意 义 匹 配 ， 不 匹 配 特 殊 含 义 ） 
匹 配 字 符 串 的 起 始 （ 结 束 ） （ 另 见 上 面 介 绍 的 ^ 和 $ ） 
扩 展 表 示 法 
(?iLmsux) 
在 正 则 表 达 式 中 嵌 入 一 个 或 者 多 个 特 殊 “ 标 记 ” 参 数 （ 或 者 通 过 函 数 / 方 法 ） 
表 示 一 个 匹 配 不 用 保 存 的 分 组 
像 一 个 仅 由 name 标 识 而 不 是 数 字 ID 标 识 的 正 则 分 组 匹 配 
(?P<data>) 
(?P=name) 
在 同 一 字 符 串 中 匹 配 由 （ ？ p < name ） 分 组 的 之 前 文 本 
(?P=data) 
表 示 注 释 ， 所 有 内 容 都 被 忽 略 
(?#comment) 
匹 配 条 件 是 如 果 ． 一 出 现 在 之 后 的 位 置 ， 而 不 使 用 输 入 字 符 串 ； 称 作 正 向 前 视 断 言 
匹 配 条 件 是 如 果 一 不 出 现 在 之 后 的 位 置 ， 而 不 使 用 输 入 字 符 串 ； 称 作 负 向 前 视 断 舀 
匹 配 条 件 是 如 果 一 出 现 在 之 前 的 位 置 ， 而 不 使 用 输 入 字 符 串 ； 称 作 正 向 后 视 断 言 
（ ？ < ： 800 刁 
匹 配 条 件 是 如 果 一 不 出 现 在 之 前 的 位 置 ， 而 不 使 用 输 入 字 符 串 ； 称 作 负 向 后 视 断 舀 
（ ？ < ！ 192 \ ． 168 \ ． ） 
如 果 分 组 所 提 供 的 湿 或 者 “ me （ 名 称 ） 存 在 ， 就 返 回 正 则 表 达 式 的 条 件 匹 配 Y, 如 
（ ？ （ 四 川 N ） 
（ ？ 0 ） y 区 ） 
果 不 存 在 ， 就 返 回 N; 是 可 选 项 

at I home 
r2d2 1 c3po 
bat bet I bit 
at, home 
r2dX c3po 
baG bit 

正 则 表 达 式 模 式 
.end 
匹 配 的 字 符 串 
匹 配 在 字 母 "f" 和 ％ " 之 间 的 任 意 一 个 字 符 ； 例 如 fao 、 四 o 、 f#o 等 
任 意 两 个 字 符 
匹 配 在 字 符 串 end 之 前 的 任 意 一 个 字 符 

正 则 表 达 式 模 式 
/bin/tcsh$ 
ASubject: hi$ 
匹 配 的 字 符 串 
任 何 以 From 作 为 起 始 的 字 符 串 
任 何 以 /bin/tcsh 作 为 结 尾 的 字 符 串 
任 何 由 单 独 的 字 符 串 subject. hi 构 成 的 字 符 串 

正 则 表 达 式 模 式 
the 
\bthe 
\bthe\b 
\Bthe 
匹 配 的 字 符 串 
任 何 包 含 the 的 字 符 串 
任 何 以 the 开 始 的 字 符 串 
仅 仅 匹 配 单 词 the 
任 何 包 含 但 并 不 以 the 作 为 起 始 的 字 符 串 

正 则 表 达 式 模 式 
b[aeiu]t 
[cr][23][dp][02] 
匹 配 的 字 符 串 
bat 、 bet 、 bit 、 but 
一 个 包 含 四 个 字 符 的 字 符 串 ， 第 一 个 字 符 是 "c" 或 "r ” ， 然 后 是 “ 2 ” 或 “ 3 " ， 后 面 
是 "d ” 或 “ p ” ， 最 后 要 么 是 ％ ” 要 么 是 “ 2 ” 。 例 如 ， c2do 、 r3p2 、 r2d2 、 c3po 等 

正 则 表 达 式 模 式 
z. [ 0 一 9 ] 
[r-u][env-y][us] 
aeiOLl] 
匹 配 的 字 符 串 
字 母 "z" 后 面 跟 着 任 何 一 个 字 符 ， 然 后 跟 着 一 个 数 字 
字 母 "r" 、 s' 、 "t" 或 者 "u" 后 面 跟 着 "e" 、 n 、 v 、 w 、 "x" 或 者 "y ' ， 然 后 跟 着 
一 个 非 元 音 字 符 （ 练 习 ： 为 什 么 我 们 说 “ 非 元 音 ' 而 不 是 “ 辅 音 " ？ ） 
不 匹 配 制 表 符 或 者 \ n 
在 一 个 ASCII 系 统 中 ， 所 有 字 符 都 位 于 “ ” 和 "a ” 之 间 ， 即 34 一 97 之 间 
u ” 或 者 “ 
S 

正 则 表 达 式 模 式 
[dn] ot? 
0 ？ 卩 ． 9 ] 
0 四 ] { 巧 ， 16 } 
[KQRBNP][a-h][1-8]-[a-h][1-8] 
匹 配 的 字 符 串 
字 母 "d ” 或 者 “ n ” ， 后 面 跟 着 一 个 ％ ” ， 然 后 是 最 多 一 个 “ t ” ， 例 如 ， do 、 no 、 dot 、 not 
任 何 数 值 数 字 ， 它 可 能 前 置 一 个 “ 0 ” ， 例 如 ， 匹 配 一 系 列 数 （ 表 示 从 1 、 9 月 的 数 值 ） ， 不 
管 是 一 个 还 是 两 个 数 字 
匹 配 巧 或 者 16 个 数 字 （ 例 如 信 用 卡 号 码 ） 
匹 配 全 部 有 效 的 （ 和 无 效 的 ） HTML 标 签 
在 “ 长 代 数 " 标 记 法 中 ， 表 示 国 际 象 棋 合 法 的 棋 盘 移 动 （ 仅 移 动 ， 不 包 括 吃 子 和 将 军 ） 。 
即 “ K ” 、 “ Q ” 、 “ R ” 、 “ B ” 、 "N ” 或 "P ” 等 字 母 后 面 加 上 "al 
“ h8 ” 之 间 的 棋 盘 坐 标 。 
前 面 的 坐 标 表 示 从 哪 里 开 始 走 棋 ， 后 面 的 坐 标 代 表 走 到 哪 个 位 置 （ 棋 格 ） 上 

正 则 表 达 式 模 式 
\w 十 -\d 十 
匹 配 的 字 符 串 
一 个 由 字 母 数 字 组 成 的 字 符 串 和 一 串 由 一 个 连 字 符 分 隔 的 数 字 
第 一 个 字 符 是 字 母 ； 其 余 字 符 （ 如 果 存 在 ） 可 以 是 字 母 或 者 数 字 （ 几 乎 等 价 于 python 中 的 有 
效 标 识 符 [ 参 见 练 习 ] ） 
美 国 电 话 号 码 的 格 式 ， 前 面 是 区 号 前 缀 ， 例 如 800 巧 55 ． 1212 
以 XXX@YYY.com 格 式 表 示 的 简 单 电 子 邮 件 地 址 

正 则 表 达 式 模 式 
匹 配 的 字 符 串 
表 示 简 单 浮 点 数 的 字 符 串 ， 也 就 是 说 ， 任 何 十 进 制 数 字 ， 后 面 可 以 接 一 个 小 数 点 和 零 个 或 
者 多 个 十 进 制 数 字 ， 例 如 “ 0 ． 004 " 、 
“ 75 ． " 等 
名 字 和 姓 氏 ， 以 及 对 名 字 的 限 制 （ 如 果 有 ， 首 字 母 必 须 大 写 ， 后 续 字 母 小 写 ） ， 全 名 前 可 以 
有 可 选 的 "Mr. ” 、 "Mrs. ” 、 "Ms. ” 或 者 "M. ” 作 为 称 谓 ， 以 及 灵 活 可 选 的 姓 氏 ， 可 以 有 多 
个 单 词 、 横 线 以 及 大 写 字 母 

正 则 表 达 式 模 式 
(?#comment) 
(?!.net) 
（ ？ < ： 80 叼 
（ ？ < ！ 192 \ 彐 68 \ .) 
（ ？ 0 ） y 区 ） 
匹 配 的 字 符 串 
以 句 点 作 为 结 尾 的 字 符 串 ， 例 如 “ 
twitter. ” 、 "facebook. ” ， 但 是 这 些 匹 配 不 会 保 存 下 来 
google. ” 、 
供 后 续 的 使 用 和 数 据 检 索 
此 处 并 不 做 匹 配 ， 只 是 作 为 注 释 
如 果 一 个 字 符 串 后 面 跟 着 ".com ” 才 做 匹 配 操 作 ， 并 不 使 用 任 何 目 标 字 符 串 
如 果 一 个 字 符 串 后 面 不 是 跟 着 “ net ” 才 做 匹 配 操 作 
如 果 字 符 串 之 前 为 “ 800 一 ” 才 做 匹 配 ， 假 定 为 电 话 号 码 ， 同 样 ， 并 不 使 用 任 何 输 入 字 符 串 
如 果 一 个 字 符 串 之 前 不 是 “ 192 ． 168 ． ” 才 做 匹 配 操 作 ， 假 定 用 于 过 滤 掉 一 组 C 类 IP 地 址 
如 果 一 个 匹 配 组 1 （ \ 1 ） 存 在 ， 就 与 y 匹 配 ； 否 则 ， 就 与 x 匹 配 

函 数 / 方 法 
仅 仅 是 re 模 块 函 数 
compile(pattern ， flags= 0 ） 
re 模 块 函 数 和 正 则 表 达 式 对 象 的 方 法 
match(pattern ， s 伊 g ， 刀 åg 、 ： 0 ） 
search(pattern ， string ， 刀 口 gs ： 0 ） 
findall(patter'/' 5 啱 [ ， 丿 g 到 ） 
n 山 ter 口 rn ， 翩 ． g [ ， 刀 口 g 到 ） 
split 仂 閉 ， 、 伊 g ， max=()) 
re 模 块 函 数 和 正 则 表 达 式 对 象 方 法 
sub(pattern ， / ， string ， count=()) 
purge() 
使 用 任 何 可 选 的 标 记 来 编 译 正 则 表 达 式 的 模 式 ， 然 后 返 回 一 个 正 则 表 达 式 对 象 
尝 试 使 用 带 有 可 选 的 标 记 的 正 则 表 达 式 的 模 式 来 匹 配 字 符 串 。 如 果 匹 配 成 功 ， 就 返 回 
匹 配 对 象 ； 如 果 失 败 ， 就 返 回 None 
使 用 可 选 标 记 搜 索 字 符 串 中 第 一 次 出 现 的 正 则 表 达 式 模 式 。 如 果 匹 配 成 功 ， 则 返 回 匹 
配 对 象 ； 如 果 失 败 ， 则 返 回 None 
查 找 字 符 串 中 所 有 （ 非 重 复 ） 出 现 的 正 则 表 达 式 模 式 ， 并 返 回 一 个 匹 配 列 表 
与 行 ndall() 函 数 相 同 ， 但 返 回 的 不 是 一 个 列 表 ， 而 是 一 个 迭 代 器 。 对 于 每 一 次 匹 配 ， 迭 
代 器 都 返 回 一 个 匹 配 对 象 
根 据 正 则 表 达 式 的 模 式 分 隔 符 ， split 函 数 将 字 符 串 分 割 为 列 表 ， 然 后 返 回 成 功 匹 配 的 
列 表 ， 分 隔 最 多 操 作 max 次 （ 默 认 分 割 所 有 匹 配 成 功 的 位 置 ） 
使 用 ， 印 / 替 换 所 有 正 则 表 达 式 的 模 式 在 字 符 串 中 出 现 的 位 置 ， 除 非 定 义 count, 否 则 就 
将 替 换 所 有 出 现 的 位 置 （ 另 见 subn() 函 数 ， 该 函 数 返 回 替 换 操 作 的 数 目 ） 
清 除 隐 式 编 译 的 正 则 表 达 式 模 式 
常 用 的 匹 配 对 象 方 法 （ 查 看 文 档 以 获 取 更 多 信 息 ） 
group(num=O) 
groups(defau/t=None) 
groupdict(defau/t=None) 
返 回 整 个 匹 配 对 象 ， 或 者 编 号 为 num 的 特 定 子 组 
返 回 一 个 包 含 所 有 匹 配 子 组 的 元 组 （ 如 果 没 有 成 功 匹 配 ， 则 返 回 一 个 空 元 组 ） 
返 回 一 个 包 含 所 有 匹 配 的 命 名 子 组 的 字 典 ， 所 有 的 子 组 名 称 作 为 字 典 的 键 （ 如 果 没 有 
成 功 匹 配 ， 则 返 回 一 个 空 字 典 ） 
常 用 的 模 块 属 性 （ 用 于 大 多 数 正 则 表 达 式 函 数 的 标 记 ） 
re.l 、 re.IGNORECASE 
re 上 、 re LOCALE 
re.M 、 re.MULTILINE 
re.S 、 re 匚 DOTALL 
re.X 、 re.VERBOSE 
不 区 分 大 小 写 的 匹 配 
根 据 所 使 用 的 本 地 语 言 环 境 通 过 \ w 、 NW 、 \b 、 、 、 实 现 匹 配 
^ 和 $ 分 别 匹 配 目 标 字 符 串 中 行 的 起 始 和 结 尾 ， 而 不 是 严 格 匹 配 整 个 字 符 串 本 身 的 起 始 
和 结 尾 
” （ 点 号 ） 通 常 匹 配 除 了 \ n （ 换 行 符 ） 之 外 的 所 有 单 个 字 符 ； 该 标 记 表 示 “ ， ” （ 点 号 ） 
能 够 匹 配 全 部 字 符 
通 过 反 斜 线 转 义 ， 否 则 所 有 空 格 加 上 # （ 以 及 在 该 行 中 所 有 后 续 文 字 ） 都 被 忽 略 ， 除 非 
在 一 个 字 符 类 中 或 者 允 许 注 释 并 且 提 高 可 读 性 