

# 快捷键

2024年2月6日 8:54

pycharm

ctrl + alt + s : 打开软件设置

ctrl + d : 复制当前行代码

shift + alt + 上\下 : 将当前行代码上移或下移

ctrl + shift + f10 : 运行当前代码文件

shift + f6 : 重命名文件

ctrl + a : 全选

ctrl + c\c\ctrl + x : 复制、粘贴、剪切

ctrl + f : 搜索

Ctrl+/ 注释或解除注释

Ctrl + p 调用方法, 传参的时候弹出提示

又或者当我们调用方法, 进行传参的时候 (快捷键ctrl + p弹出提示) :

```
import random
    a: int, b: int
random.randint()
```

全选按tab键, 退格

# 输入和输出

2024年2月6日 16:45

**python大小写敏感**

**python始终坚持4个空格的缩进**

**print语句如何输出多份内容？**

用逗号隔开，print()会依次打印每个字符串，遇到逗号 “,” 会输出一个空格

print(内容1, 内容2, ……, 内容N)

**print输出不换行**

默认print语句输出内容会自动换行，如下图：

```
print("Hello")
print("World")
```

est ×  
D:\dev\python\p  
Hello  
World

在即将完成的案例中，我们需要使用print语句，输出不换行的功能，非常简单，加上end="" 即可不换行了：

```
print("Hello", end=' ')
print("World", end=' ')
```

est ×  
D:\dev\python\python3.10  
HelloWorld

**制表符**

\t，效果等同于在键盘上按下：tab键。

让多行字符串进行对齐。

```

print("Hello World")
print("itheima best") 使用空格
                  无法对齐

print("Hello\tWorld")
print("itheima\tbest") 使用\t后，可以对齐

```



```

def main():
    print("-----主菜单-----")
    print(f" {name}，您好，欢迎来到黑马银行ATM。请选择操作：")
    print("查询余额\t[输入1]")
    print("存款\t\t[输入2]")
    print("取款\t\t[输入3]")
    print("退出\t\t[输入4]")

```

一个反斜杠对不齐就多几个

## Input

数据输出: print

数据输入: input

input()返回的数据类型是str

```

print("请告诉我你是谁？")
name = input() ←
print("Get! !!! 你是: %s" % name)

```



在前面的代码中，输出“请告诉我你是谁？”的print语句其实是多余的

input()语句其实是在要求使用者输入内容前，输出提示内容的哦，方式如下：

input(提示信息)

input()语句，默认结果是字符串，若需要数字也需要转换

```
name = input("请告诉我你是谁? ")  
print("Get! ! ! 你是: %s" % name)
```

test (1) ×

D:\dev\Python\Python3.10.4\py

请告诉我你是谁? **黑马程序员**

Get! ! ! 你是: 黑马程序员

# 数字类型

2024年2月9日 10:21

## 1. type

在print语句中，直接输出类型信息：

```
print(type("黑马程序员"))
print(type(666))
print(type(11.345))
```

```
test ×
D:\dev\Python\Python3.10
<class 'str'>
<class 'int'>
<class 'float'>
```

## 2. 用变量存储type()的结果（返回值）

```
1 string_type = type("黑马程序员")
2 int_type = type(666)
3 float_type = type(11.345)
4 print(string_type)
5 print(int_type)
6 print(float_type)
```

```
Run: test ×
D:\dev\Python\Python3.10.4\python
<class 'str'>
<class 'int'>
<class 'float'>
```

## 数字类型转换

语句(函数)	说明
int(x)	将x转换为一个整数
float(x)	将x转换为一个浮点数
str(x)	将对象 x 转换为字符串

1. 任何类型，都可以通过str()，转换成字符串
2. 字符串内必须真的是数字，才可以将字符串转换为数字

```
v_str = "我不是数字"      字符串内不是数字, 是无法完成到数字的转换的
num = int(v_str)

est (1) ×
D:\dev\Python\Python3.10.4\python.exe D:/python-learn/01_Python基础知识
Traceback (most recent call last):
  File "D:/python-learn/01_Python基础知识\test.py", line 2, in <module>
    num = int(v_str)
ValueError: invalid literal for int() with base 10: '我不是数字'
```

## 整数

对于很大的数, 例如10000000000, 很难数清楚0的个数。Python允许在数字中间以\_分隔, 因此, 写成10\_000\_000\_000和10000000000是完全一样的。十六进制数也可以写成0xa1b2\_c3d4。

## 浮点数

浮点数也就是小数, 之所以称为浮点数, 是因为按照科学记数法表示时, 一个浮点数的小数点位置是可变的, 比如,  $1.23 \times 10^9$ 和 $12.3 \times 10^8$ 是完全相等的。浮点数可以用数学写法, 如1.23, 3.14, -9.01, 等等。但是对于很大或很小的浮点数, 就必须用科学计数法表示, 把10用e替代,  $1.23 \times 10^9$ 就是 $1.23e9$ , 或者 $12.3e8$ ,  $0.000012$ 可以写成 $1.2e-5$ , 等等。

# 转义符占位符

2024年2月6日 19:17

## 字符串在Python中有多种定义形式：

单引号定义法：

双引号定义法：

三引号定义法：

三引号定义法，和多行注释的写法一样，同样支持换行操作。

使用变量接收它，它就是字符串

不使用变量接收它，就可以作为多行注释使用。

## 转义字符

q:如果我想要定义的字符串本身，是包含：单引号、双引号自身呢？如何写？

a:转义字符\可以转义很多字符，比如\n表示换行，\t表示制表符，字符\本身也要转义，所以\\表示的字符就是\，可以在Python的交互式命令行用print()打印字符串看看：

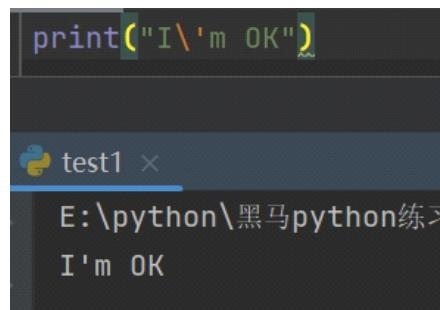
单引号定义法，可以内含双引号

双引号定义法，可以内含单引号

可以使用转义字符 (\) 来将引号解除效用，变成普通字符串

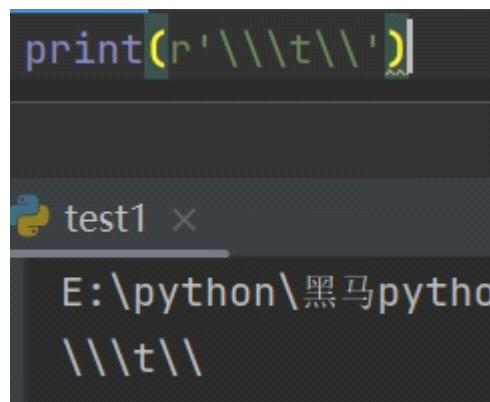
```
name = "\\"黑马程序员\""  
name = '\\'黑马程序员\''
```

\加在内部引号的前面



```
print("I\\'m OK")  
E:\python\黑马python练习  
I'm OK
```

了简化，Python还允许用r''表示'内部的字符串默认不转义，可以自己试试：



```
print(r'\\\\t\\\\')  
E:\python\黑马python  
\\\\t\\\\
```

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用'''...'''的格式表示多行内容，可以自己试试：

### 字符串加号拼接

如果我们有两个字符串（文本）字面量，可以将其拼接成一个字符串，通过+号即可完成，如：

```
print("我是: " + name + ", 我的地址是: " + address)
```

```
我是: 黑马程序员, 我的地址是: 建材城东路9号院
```

字符串无法和非字符串变量进行拼接

因为类型不一致，无法接上

### 字符串格式化

完成字符串和变量的快速拼接

```
name = "黑马程序员"  
message = "学IT就来 %s" % name  
print(message)
```



test ×

D:\dev\Python\Python3.10.4\python.

学IT就来 黑马程序员

% 表示：我要占位

s 表示：将变量变成字符串放入占位的地方

所以，综合起来的意思就是：我先占个位置，等一会有个变量过来，我把它变成字符串放到占位的位置

```
1 class_num = 57  
2 avg_salary = 16781  
3 message = "Python大数据学科, 北京%s期, 毕业平均工资: %s" % (class_num, avg_salary)  
4 print(message)
```

Run: test ×

D:\dev\Python\Python3.10.4\python.exe D:/python-learn/01\_你好Python/test.py

Python大数据学科, 北京57期, 毕业平均工资: 16781

其中的，%s

- % 表示：我要占位
- s 表示：将变量变成字符串放入占位的地方

多个变量占位，变量要用括号括起来，并按照占位的顺序填入

数字也能用%s占位吗？

可以的哦，这里是将数字转换成了字符串哦

也就是数字57，变成了字符串"57"被放入占位的地方

```
name = "不要放弃"  
name_2 = "会赢的"  
message = "嘟嘟: %s%s" %(name, name_2)  
print(message)
```

格式符号	转化
%s	将内容转换成字符串，放入占位位置
%d	将内容转换成整数，放入占位位置
%f	将内容转换成浮点型，放入占位位置

### 字符串数字精度控制

我们可以使用辅助符号"m.n"来控制数据的宽度和精度

m, 控制宽度，要求是数字（很少使用），设置的宽度小于数字自身，不生效

.n, 控制小数点精度，要求是数字，会进行小数的四舍五入

示例：

%5d: 表示将整数的宽度控制在5位，如数字11，被设置为5d，就会变成：[空格][空格][空格]11，用三个空格补足宽度。

%5.2f: 表示将宽度控制为5，将小数点精度设置为2

小数点和小数部分也算入宽度计算。如，对11.345设置了%7.2f后，结果是：[空格][空格]11.35。2个空格补足宽度，小数部分限制2位精度后，四舍五入为 .35

%2f: 表示不限制宽度，只设置小数点精度为2，如11.345设置%.2f后，结果是11.35

```
num1 = 11  
num2 = 11.345  
print("数字11宽度限制5，结果: %5d" % num1)  
print("数字11宽度限制1，结果: %1d" % num1)  
print("数字11.345宽度限制7，小数精度2，结果 : %7.2f" % num2)  
print("数字11.345不限制宽度，小数精度2，结果 : %.2f" % num2)
```

```
test(1) <  
D:\dev\Python\Python3.10.4\python.exe D:/python-learn/01_Python基础知识/test.py  
数字11宽度限制5，结果: 11 宽度5，补了3个空格  
数字11宽度限制1，结果: 11 宽度小于数字本身，无影响  
数字11.345宽度限制7，小数精度2，结果 : 11.35 宽度7，补了2个空格，小数精度2，四舍五入后为.35  
数字11.345不限制宽度，小数精度2，结果 : 11.35 不限制宽度，小数点后四舍五入后为.35
```

### 字符串格式化——快速写法

通过语法：f"内容{变量}"的格式来快速格式化

```
name = "传智播客"
set_up_year = 2006
stock_price = 19.99
print(f"我是{name}，我成立于: {set_up_year}，我今天的股票价格是: {stock_price}")
```

test (1) ×  
D:\dev\Python\Python3.10.4\python.exe D:/python-learn/01\_Python基础  
我是传智播客，我成立于：2006，我今天的股票价格是：19.99 不做精度控制，原样输出

这种拼接方法不理会类型，不做精度控制

# 标识符

2024年2月6日 19:03

- 内容限定

标识符命名中，只允许出现：

英文

中文

数字

下划线（\_）

这四类元素。

其余任何内容都不被允许。

- 大小写敏感

以定义变量为例：

Andy = “安迪1”

andy = “安迪2”

字母a的大写和小写，是完全能够区分的。

- 不可使用关键字

Python中有一系列单词，称之为关键字

关键字在Python中都有特定用途

我们不可以使用它们作为标识符

False	True	None	and	as	assert	break	class
continue	def	del	elif	else	except	finally	for
from	global	if	import	in	is	lambda	nonlocal
not	or	pass	raise	return	try	while	with
							yield

- 变量命名规范 - 英文字母全小写

# 运算符

2024年2月6日 19:11

```
print("1 + 1结果是: %d" % (1 + 1))
print("2 - 1结果是: %d" % (2 - 1))
print("1 * 3结果是: %d" % (1 * 3))
print("9 / 3结果是: %d" % (9 / 3))
print("9 // 2 (9整除2) 结果是: %d" % (9 // 2))
print("9 %% 2 (9余2的结果是) 结果是: %d" % (9 % 2))
print("2 的 6 次方是: 结果是: %d" % (2 ** 6))
```

test (1) ×

```
D:\dev\Python\Python3.10.4\python.exe D:/python/test(1).py
1 + 1结果是: 2
2 - 1结果是: 1
1 * 3结果是: 3
9 / 3结果是: 3
9 // 2 (9整除2) 结果是: 4
9 % 2 (9余2的结果是) 结果是: 1
2 的 6 次方是: 结果是: 64
```

整数的地板除//永远是整数，即使除不尽。要做精确的除法，使用/就可以。

## 赋值运算符

运算符	描述	实例
=	赋值运算符	把 = 号右边的结果 赋给 左边的变量，如 num = 1 + 2 * 3，结果 num 的值为 7
运算符	描述	实例
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c *= a 等效于 c = c * a
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a 等效于 c = c ** a
//=	取整除赋值运算符	c // a 等效于 c = c // a

# 对表达式格式化

2024年2月6日 20:10

```
print("1 * 1的结果是: %d" % (1 * 1))
print(f"1 * 1的结果是: {1 * 1}")
print("字符串在Python中的类型是: %s" % type('字符串'))
```

test (1) ×

```
D:\dev\Python\Python3.10.4\python.exe D:/python-
1 * 1的结果是: 1
1 * 1的结果是: 1
字符串在Python中的类型是: <class 'str'>
```

在无需使用变量进行数据存储的时候，可以直接格式化表达式，简化代码哦

# 布尔类型

2024年2月7日 8:46

布尔 (bool) 表达现实生活中的逻辑, 即真和假

True表示真

False表示假。

注意首字母大写

True本质上是一个数字记作1, False记作0

定义变量存储布尔类型数据:

变量名称 = 布尔类型字面量

布尔类型的数据, 不仅可以通过定义得到, 也可以通过比较运算符进行内容比较得到。

如下代码:

```
result = 10 > 5
print(f"10 > 5 的结果是: {result}, 类型是: {type(result)}")
```

test ×  
D:\dev\python\python3.10.4\python.exe D:/python-learn/03\_  
10 > 5 的结果是: **True**, 类型是: **<class 'bool'>**

```
result = "itcast" == "itheima"
print(f"字符串itcast是否和itheima相等, 结果是: {result}, 类型是: {type(result)}")
```

test ×  
D:\dev\python\python3.10.4\python.exe D:/python-learn/03\_Python判断语句/test.py  
字符串itcast是否和itheima相等, 结果是: **False**, 类型是: **<class 'bool'>**

## 比较运算符

运算符	描述	示例
==	判断内容 <b>是否相等</b> , 满足为True, 不满足为False	如a=3, b=3, 则 (a == b) 为 True
!=	判断内容 <b>是否不相等</b> , 满足为True, 不满足为False	如a=1, b=3, 则(a != b) 为 True
>	判断运算符左侧内容 <b>是否大于</b> 右侧 满足为True, 不满足为False	如a=7, b=3, 则 (a > b) 为 True
<	判断运算符左侧内容 <b>是否小于</b> 右侧 满足为True, 不满足为False	如a=3, b=7, 则 (a < b) 为 True
>=	判断运算符左侧内容 <b>是否大于等于</b> 右侧 满足为True, 不满足为False	如a=3, b=3, 则 (a >= b) 为 True

<=	判断运算符左侧内容 <b>是否小于等于</b> 右侧 满足为True, 不满足为False	如a=3, b=3, 则 (a <= b) 为 True
----	--	---------------------------------

# 判断语句的嵌套

2024年2月7日 10:38

`if` 条件1:

    满足条件1 做的事情1

    满足条件1 做的事情2

`if` 条件2:

    满足条件2 做的事情1

    满足条件2 做的事情2

如上图，第二个if，属于第一个if内，只有第一个if满足条件，才会执行第二个if

嵌套的关键点，在于：空格缩进

通过空格缩进，来决定语句之间的：层次关系

```
print("欢迎来到黑马动物园。")
if int(input("输入你的身高: ")) 1 > 120:
    print("你的身高大于120cm, 不可以免费")
    print("不过如果你的vip等级高于3, 可以免费游玩")

    2
    if int(input("请告诉我你的vip级别: ")) > 3:
        print("恭喜你, 你的vip级别大于3, 可以免费游玩。")
    else:
        print("Sorry, 你需要补票, 10元。")
else:
    print("欢迎你小朋友, 可以免费游玩。")
```

判断有2层

当外层if满足条件（图中编号1）时，才会执行内层if判断（图中编号2）

当外层if（编号1）不满足，直接执行外层else

自由组合嵌套，需求如下：

公司要发礼物，条件是：

1. 必须是大于等于18岁小于30岁的成年人
2. 同时入职时间需满足大于两年，或者级别大于3才可领取

# match模式匹配

2024年2月9日 11:27

当我们用if ... elif ... elif ... else ... 判断时，会写很长一串代码，可读性较差。

如果要针对某个变量匹配若干种情况，可以使用match语句。

来自 <<https://www.liaoxuefeng.com/wiki/1016959663602400/1572077106626595>>

例如，某个学生的成绩只能是A、B、C，用if语句编写如下：

```
score = 'B'  
if score == 'A':  
    print('score is A.')  
elif score == 'B':  
    print('score is B.')  
elif score == 'C':  
    print('score is C.')  
else:  
    print('invalid score.')
```

## 模式匹配

如果用match语句改写，则改写如下：

```
score = 'B'  
  
match score:  
    case 'A':  
        print('score is A.')  
    case 'B':  
        print('score is B.')  
    case 'C':  
        print('score is C.')  
    case _: # _表示匹配到其他任何情况  
        print('score is ???.')  
  
使用match语句时，我们依次用case xxx匹配，并且可以在最后（且仅能在最后）加一个case _表示“任意值”，代码较if ... elif ... else ... 更易读。
```

## 复杂匹配

match语句除了可以匹配简单的单个值外，还可以匹配多个值、匹配一定范围，并且把匹配后的值绑定到变量：

```
age = 15
```

```
match age:  
    case x if x < 10:  
        print(f'< 10 years old: {x}')  
    case 10:  
        print('10 years old.')  
    case 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18:  
        print('11~18 years old.')  
    case 19:  
        print('19 years old.')  
    case _:
```

```
print('not sure.')
```

在上面这个示例中，第一个case `x` if `x < 10`表示当`age < 10`成立时匹配，且赋值给变量`x`，第二个case `10`仅匹配单个值，第三个case `11|12|...|18`能匹配多个值，用`|`分隔。

# while

2024年2月7日 14:06

**while 条件:**

    条件满足时，做的事情1

    条件满足时，做的事情2

    条件满足时，做的事情3

    ....(省略)....

只要条件满足

会无限循环执行

```
i = 0
while i < 100:
    print("小美， 我喜欢你")
    i += 1
```

while循环的注意事项

条件需提供布尔类型结果，True继续，False停止

空格缩进不能忘

请规划好循环终止条件，否则将无限循环

练习题：

设置一个范围1-100的随机整数变量，通过while循环，配合input语句，判断输入的数字是否等于随机数

无限次机会，直到猜中为止

每一次猜不中，会提示大了或小了

猜完数字后，提示猜了几次

提示：

无限次机会，终止条件不适合用数字累加来判断

可以考虑布尔类型本身 (True or False)

需要提示几次猜中，就需要提供数字累加功能

## 老师的答案

```
# 获取范围在1-100的随机数字
import random
num = random.randint(1, 100)
# 定义一个变量，记录总共猜测了多少次
count = 0

# 通过一个布尔类型的变量，做循环是否继续的标记
flag = True
while flag:
    guess_num = int(input("请输入你猜测的数字:"))
    count += 1
    if guess_num == num:
        print("猜中了")
        # 设置为False就是终止循环的条件
        flag = False
    else:
        if guess_num > num:
            print("你猜的大了")
        else:
            print("你猜的小了")

print(f"你总共猜测了{count}次")
```

我的答案：

```
import random
num = random.randint(1, 10)
print("游戏开始")
answer = int(input("请输入数字: "))
i = 1
while answer != num:
    if answer > num:
        i += 1
        print("大啦")
        answer = int(input("请输入数字: "))
    elif answer < num:
        i += 1
        print("小啦")
        answer = int(input("请输入数字: "))
    else:
        print("游戏结束，回答正确，您共猜了", i, "次")
```

# while嵌套

2024年2月7日 14:42

`while` 条件1:

条件1满足时，做的事情1

条件1满足时，做的事情2

条件1满足时，做的事情3

... (省略) ...

`while` 条件2:

条件2满足时，做的事情1

条件2满足时，做的事情2

条件2满足时，做的事情3

... (省略) ...

## 九九乘法表

```
# 定义外层循环的控制变量
i = 1
while i <= 9: i = 4

# 定义内层循环的控制变量
j = 1
while j <= i:
    # 内层循环的print语句，不要换行，通过\t制表符进行对齐
    print(f"\t{j} * {i} = {j * i}\t", end='')
    j += 1

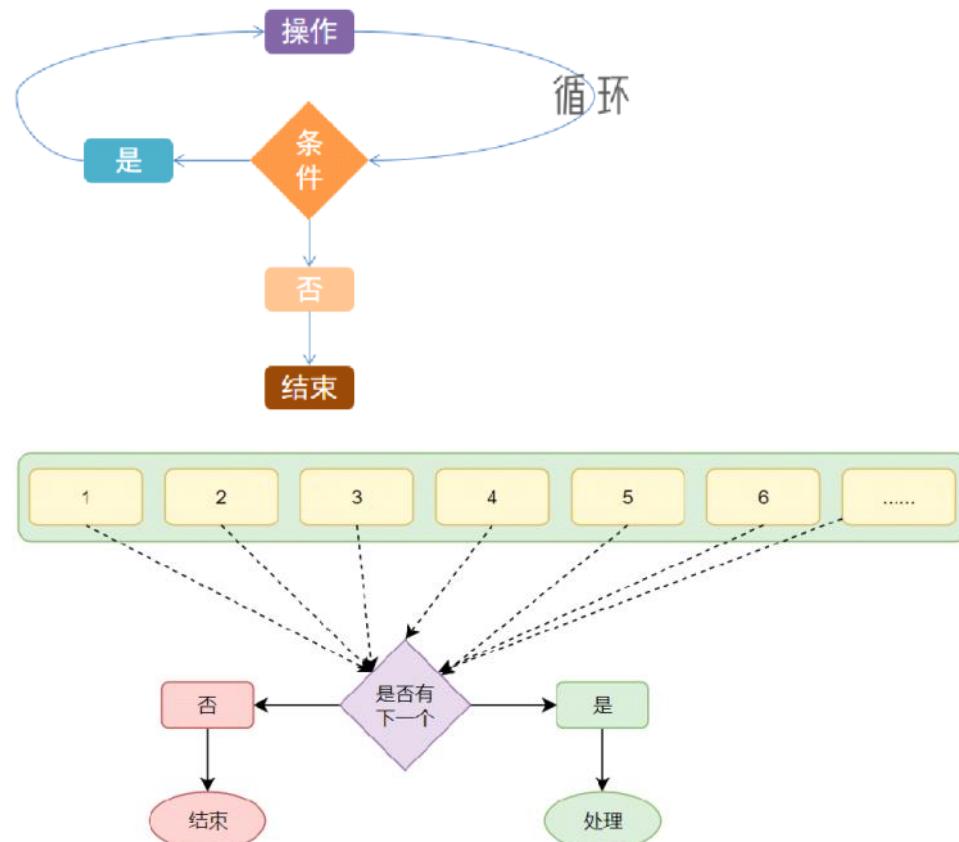
i += 1
print()      # print空内容，就是输出一个换行
```

# for

2024年2月7日 15:18

while循环的循环条件是自定义的，自行控制循环条件

for循环是一种“轮询”机制，是对一批内容进行“逐个处理”



for循环就是将“待办事项”逐个完成的循环机制

**for 临时变量 in 待处理数据集:**

    循环满足条件时执行的代码

- 临时变量，在编程规范上，作用范围（作用域），只限定在for循环内部
- 如果在for循环外部访问临时变量：  
实际上是可以访问到的  
在编程规范上，是不建议这么做的

```
for i in range(5):  
    print(i)
```

```
print(i)
```

如图，第一个i是正确的，第二个i规范上是不允许的，解决方案如下图，在for循环之外先定义好变量：

```
i = 0  
for i in range(5):  
    ...
```

```
i = 0
for i in range(5):
    print(i)

print(i)
```

语法中的：待处理数据集，严格来说，称之为：可迭代类型  
可迭代类型指，其内容可以一个个依次取出的一种类型，包括：  
字符串  
列表  
元组  
等

```
# 定义字符串name
name = "itheima"

# for循环处理字符串
for x in name:
    print(x)
```

运行结果如下：

```
i
t
h
e
i
m
a
```

无法定义循环条件，只能被动取出数据处理  
要注意，循环内的语句，需要有空格缩进

# range

2024年2月7日 15:56

`range(num)`

获取一个从0开始, 到num结束的数字序列 (不含num本身)

如`range(5)`取得的数据是: [0, 1, 2, 3, 4]

`range(num1, num2)`

获得一个从num1开始, 到num2结束的数字序列 (不含num2本身)

如, `range(5, 10)`取得的数据是: [5, 6, 7, 8, 9]

`range(num1, num2, step)`

获得一个从num1开始, 到num2结束的数字序列 (不含num2本身)

数字之间的步长, 以step为准 (step默认为1)

如, `range(5, 10, 2)`取得的数据是: [5, 7, 9]

`range`通常与`for`搭配使用

例题: 送10次玫瑰花

`for x in range(10):`

`print("送玫瑰花")`

# For 嵌套

2024年2月7日 16:31

**for** 临时变量 **in** 待处理数据集(序列):

    循环满足条件应做的事情 1

    循环满足条件应做的事情 2

    循环满足条件应做的事情 N

    ...

**for** 临时变量 **in** 待处理数据集(序列):

    循环满足条件应做的事情 1

    循环满足条件应做的事情 2

    循环满足条件应做的事情 N

For 循环九九乘法表

```
for m in range (1,10):  
    for n in range (1,m+1):  
        print(f"{n}*{m}={m * n}\t", end='')  
    print()
```

# continue

2024年2月7日 20:02



在循环内，遇到continue就结束当次循环，语句2不会被执行，重新循环语句1

```
4  
5      # 演示循环中断语句 continue  
6      for i in range(1, 6):  
7          print("语句1")  
8          continue  
9          print("语句2")  
10  
for i in range(1, 6)  
Run: 09_循环中断  
语句1  
语句1  
语句1  
语句1  
语句1
```

continue关键字只可以控制：它所在的循环临时中断

```
for i in range(1, 100): 2
```

语句 1

```
for j in range(1, 100): 1
```



语句 2

```
continue
```

~~语句 3~~

语句 4

# break

2024年2月7日 20:17

break关键字用于：直接结束所在循环

```
for i in range(1, 100):
```

语句 1

break

语句 2

语句 3

## break在嵌套循环中的应用

break关键字同样只可以控制：它所在的循环结束

```
for i in range(1, 100): 2
```

语句1

```
    for j in range(1, 100): 1
```

语句2

break

语句3

语句4

```
n = 1while n <= 100:
```

```
    if n > 10: # 当n = 11时, 条件满足, 执行break语句break# break语句会结束当前循环
```

```
    print(n)
```

```
    n = n + 1print('END')
```

# 迭代和迭代器

2024年2月14日 16:55

- 可以直接作用于for循环的数据类型有以下几种：

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。

可以使用isinstance()判断一个对象是否是Iterable对象：

```
>>> from collections.abc 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
```

- 使用内建的isinstance函数可以判断一个变量是不是字符串：

```
>>> x = 'abc'
>>> y = 123
>>> isinstance(x, str)
True
>>> isinstance(y, str)
False
```

- 而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，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

可以使用isinstance()判断一个对象是否是Iterator对象：

```
>>> from collections.abc 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
```

- 为什么list、dict、str等数据类型不是Iterator？

这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以**Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。**

**Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。**

- 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：

```
it = iter([1,2,3,4,5])
while True:
    try:
        x = next(it)
        print(x)
    except StopIteration:
        break
```

# 变量

2024年2月8日 19:29

内部的局部变量如何改变全局变量

☆ 使用 `global`关键字 可以在函数内部声明变量为全局变量, 如下所示

```
1 num = 100
2
3 def testA():
4     print(num)
5
6
7 def testB():
8     # global 关键字声明a是全局变量
9     global num
10    num = 200
11    print(num)
12
13
14 testA()          # 结果: 100
15 testB()          # 结果: 200
16 print(f'全局变量num = {num}')    # 结果: 全局变量num = 200
```

# 数据容器

2024年2月9日 17:10

一种可以容纳多份数据的数据类型，容纳的每一份数据称之为1个元素  
每一个元素，可以是任意类型的数据，如字符串、数字、布尔等。

数据容器根据特点的不同，如：

- 是否支持重复元素
- 是否可以修改
- 是否有序，等分为5类，分别是：
- 列表（list）、元组（tuple）、字符串（str）、集合（set）、字典（dict）

# list列表

2024年2月9日 17:12

```
1 # 字面量
2 [元素1, 元素2, 元素3, 元素4, ...]
3
4 # 定义变量
5 变量名称 = [元素1, 元素2, 元素3, 元素4, ...]
6
7 # 定义空列表
8 变量名称 = []
9 变量名称 = list()
```

列表内的每一个数据，称之为元素

- 以 [] 作为标识
- 列表内每一个元素之间用, 逗号隔开

```
1 name_list = ['itheima', 'itcast', 'python']
2 print(name_list)
3 print(type(name_list))
['itheima', 'itcast', 'python']
<class 'list'>
```

```
1 my_list = ['itheima', 666, True]
2 print(my_list)
3 print(type(my_list))
['itheima', 666, True]
<class 'list'>
```

列表可以一次存储多个数据，且可以为不同的数据类型，支持嵌套

```
1 my_list = [ [1, 2, 3], [4, 5, 6] ]
2 print(my_list)
3 print(type(my_list))
[[1, 2, 3], [4, 5, 6]]
<class 'list'>
```

## 列表的下标索引



列表中的每一个元素，都有其位置下标索引，从前向后的方向，从0开始，依次递增。我们只需要按照下标索引，即可取得对应位置的元素。

```
1 # 语法: 列表[下标索引]
2
3 name_list = ['Tom', 'Lily', 'Rose']
4 print(name_list[0]) # 结果: Tom
5 print(name_list[1]) # 结果: Lily
6 print(name_list[2]) # 结果: Rose
```

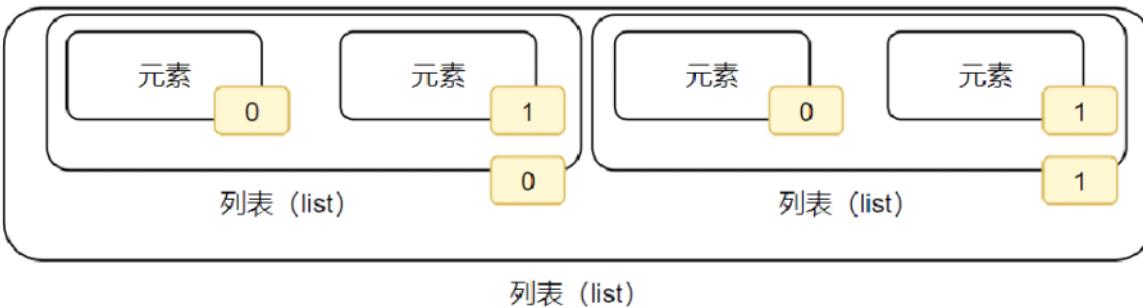
## 反向索引



如图，从后向前，下标索引为：-1、-2、-3，依次递减。

```
1 # 语法: 列表[标号]
2
3 name_list = ['Tom', 'Lily', 'Rose']
4 print(name_list[-1]) # 结果: Rose
5 print(name_list[-2]) # 结果: Lily
6 print(name_list[-3]) # 结果: Tom
```

## 嵌套列表的下标索引



```
1 # 2层嵌套list
2 my_list = [ [1, 2, 3], [4, 5, 6] ]
3
4 # 获取内层第一个list
5 print(my_list[0])                      # 结果: [1, 2, 3]
6
7 # 获取内层第一个list的第一个元素
8 print(my_list[0][0])                    # 结果: 1
```

# list方法

2024年2月9日 17:28

编号	使用方式	作用
1	列表. append(元素)	向列表中追加一个元素
2	列表. extend(容器)	将数据容器的内容依次取出，追加到列表尾部
3	列表. insert(下标, 元素)	在指定下标处，插入指定的元素
4	del 列表[下标]	删除列表指定下标元素
5	列表. pop(下标)	删除列表指定下标元素
6	列表. remove(元素)	从前向后，删除此元素第一个匹配项
7	列表. clear()	清空列表
8	列表. count(元素)	统计此元素在列表中出现的次数
9	列表. index(元素)	查找指定元素在列表的下标 找不到报错ValueError
10	len(列表)	统计容器内有多少元素

在Python中，如果将函数定义为class (类) 的成员，那么函数会称之为：方法

```
def add(x, y):  
    return x + y
```

函数

```
class Student:  
  
    def add(self, x, y):  
        return x + y
```

方法

方法和函数功能一样，有传入参数，有返回值，只是方法的使用格式不同：

函数的使用：

```
num = add(1, 2)
```

方法的使用：

```
student = Student()  
num = student.add(1, 2)
```

## 1. 查找某元素的下标

功能：查找指定元素在列表的下标，如果找不到，报错ValueError

语法：列表.index(元素)

index就是列表对象（变量）内置的方法（函数）

```
1 my_list = ["itheima", "itcast", "python"]
2 print(my_list.index("itcast"))      # 结果： 1
```

## 2. 修改某索引位置的元素值

语法：列表[下标] = 值

可以使用如上语法，直接对指定下标（正向、反向下标均可）的值进行：重新赋值（修改）

```
1 # 正向下标
2 my_list = [1, 2, 3]
3 my_list[0] = 5
4 print(my_list)  # 结果： [5, 2, 3]
5
6 # 反向下标
7 my_list = [1, 2, 3]
8 my_list[-3] = 5
9 print(my_list)  # 结果： [5, 2, 3]
```

## 3. 插入元素

语法：列表.insert(下标, 元素)，在指定的下标位置，插入指定的元素

```
1 my_list = [1, 2, 3]
2 my_list.insert(1, "itheima")
3 print(my_list)  # 结果： [1, "itheima", 3, 4]
```

## 4. 追加元素

语法：列表.append(元素)，将指定元素，追加到列表的尾部

```
1 my_list = [1, 2, 3]
2 my_list.append(4)
3 print(my_list)  # 结果： [1, 2, 3, 4]
4
5 my_list = [1, 2, 3]
6 my_list.append([4, 5, 6])
7 print(my_list)  # 结果： [1, 2, 3, [4, 5, 6]]
```

追加多个，用extend []

```
1 mylist = ["dudu", "nameqin", "alice xueer"]
2 print(mylist)
3 mylist.extend([2, "momo"])
4 print(mylist)
```

Run:  test1 ×

▶ E:\python\黑马python练习\pythonProject1\.venv\Scripts\python.exe  
['dudu', 'nameqin', 'alice xueer']  
['dudu', 'nameqin', 'alice xueer', 2, 'momo']  
Process finished with exit code 0

## 5.删除元素

语法1: del 列表[下标]

语法2: 列表.pop(下标)

```
1 my_list = [1, 2, 3]
2
3 # 方式1
4 del my_list[0]
5 print(my_list) # 结果: [2, 3]
6 # 方式2
7 my_list.pop(0)
8 print(my_list) # 结果: [2, 3]
```

```
mylist = ["itcast", "itheima", "python"]
element = mylist.pop(2)
print(f"通过pop方法取出元素后列表内容: {mylist}, 取出的元素是: {element}")
```

## 6.删除元素在列表中的第一个匹配项

语法: 列表.remove(元素)

```
1 my_list = [1, 2, 3, 2, 3]
2 my_list.remove(2)
3 print(my_list) # 结果: [1, 3, 2, 3]
```

从左到右, 找到第一个2, 删掉

## 7.清空列表

语法: 列表.clear()

```
1 my_list = [1, 2, 3]
2 my_list.clear()
3 print(my_list) # 结果: []
```

## 8.统计某元素在列表内的数量

语法: 列表.count(元素)

```
1 | my_list = [1, 1, 1, 2, 3]
2 | print(my_list.count(1))  # 结果: 3
```

## 9.统计列表内，有多少元素

语法: len(列表)

可以得到一个int数字，表示列表内的元素数量

```
1 | my_list = [1, 2, 3, 4, 5]
2 | print(len(my_list))          # 结果5
```

# list (列表) 的遍历

2024年2月10日 20:25

将容器内的元素依次取出进行处理的行为，称之为：遍历、迭代。

如何遍历列表的元素呢？

- 可以使用前面学过的while循环，如何在循环中取出列表的元素呢？
- 使用列表[下标]的方式取出，循环条件如何控制？
- 定义一个变量表示下标，从0开始
- 循环条件为 下标值 < 列表的元素数量

while循环

```
index = 0

while index < len(列表):
    元素 = 列表[index]
    对元素进行处理
    index += 1

def list_while_func():
    mylist = ["dudu", "damegin", "choucoupi"]
    index = 0
    while index < len(mylist):
        element = mylist[index]
        print(f"列表的元素: {element}")
        index += 1

list_while_func()
```

for循环

对比while，for循环更加适合对列表等数据容器进行遍历。

```
for 临时变量 in 数据容器:
    对临时变量进行处理
```

表示，从容器内，依次取出元素并赋值到临时变量上。

在每一次的循环中，我们可以对临时变量（元素）进行处理。

```
def list_for_func():
    mylist = ["dudu", "damegin", "chougoupi"]
    for element in mylist:
        print(f"列表的元素: {element}")

list_for_func()
```

## while循环和for循环的对比

while循环和for循环，都是循环语句，但细节不同：

- **在循环控制上：**

while循环可以自定循环条件，并自行控制

for循环不可以自定循环条件，只可以一个个从容器内取出数据

- **在无限循环上：**

while循环可以通过条件控制做到无限循环

for循环理论上不可以，因为被遍历的容器容量不是无限的

- **在使用场景上：**

while循环适用于任何想要循环的场景

for循环适用于，遍历数据容器的场景或简单的固定次数循环场景

```
my_list = [1, 2, 3, 4, 5]
for i in my_list:
    print("小美，我喜欢你")
```

D:\dev\python\python3.10.4\

小美，我喜欢你  
小美，我喜欢你  
小美，我喜欢你  
小美，我喜欢你  
小美，我喜欢你

定义一个列表，内容是：[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

遍历列表，取出列表内的偶数，并存入一个新的列表对象中

使用while循环和for循环各操作一次

```
D:\dev\python\python3.10.4\python.exe D:/python-learn/05_数据容器/test.py
```

通过while循环，从列表：[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]中取出偶数，组成新列表：[2, 4, 6, 8, 10]

通过for循环，从列表：[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]中取出偶数，组成新列表：[2, 4, 6, 8, 10]

提示：

通过if判断来确认偶数

通过列表的append方法，来增加元素

```
usage
def list_even():
    list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    even_list = []
    for element in list:
        if element % 2 == 0:
            even_list.append(element)
    return even_list
print(list_even())

```

test1 ×

E:\python\黑马python练习\pythonProject1\.venv\  
[2, 4, 6, 8, 10]

Process finished with exit code 0

# 列表生成式List Comprehensions

2024年2月14日 16:58

来自 <<https://www.liaoxuefeng.com/wiki/1016959663602400/1017317609699776>>

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

```
>>> print([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]
```

- 还可以使用两层循环，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']
```

三层和三层以上的循环就很少用到了。

- for循环其实可以同时使用两个甚至多个变量，比如dict的items()可以同时迭代key和value：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C'}
>>> for k, v in d.items():
...     print(k, '=', v)
...
y = B
x = A
z = C
```

因此，列表生成式也可以使用两个变量来生成list：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C'}
>>> [k + '=' + v for k, v in d.items()]
['y=B', 'x=A', 'z=C']
```

- 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']
```

- 列表生成式中if...else 的用法

1. 例如，以下代码正常输出偶数：

```
>>> [x for x in range(1, 11) if x % 2 == 0]
[2, 4, 6, 8, 10]
```

但是，我们不能在最后的if加上else：

```
>>> [x for x in range(1, 11) if x % 2 == 0 else 0]
File "<stdin>", line 1[x for x in range(1, 11) if x % 2 == 0 else 0]
```

```
SyntaxError: invalid syntax
```

这是因为跟在for后面的if是一个筛选条件，不能带else，否则如何筛选？

2. 另一些童鞋发现把if写在for前面必须加else，否则报错：

---

```
>>> [x if x % 2 == 0 for x in range(1, 11)]
  File "<stdin>", line 1
    [x if x % 2 == 0 for x in range(1, 11)]
               ^
```

```
SyntaxError: invalid syntax
```

这是因为for前面的部分是一个表达式，它必须根据x计算出一个结果。因此，考察表达式： $x \text{ if } x \% 2 == 0$ ，它无法根据x计算出结果，因为缺少else，必须加上else：

```
>>> [x if x % 2 == 0 else -x for x in range(1, 11)]
[-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
```

上述for前面的表达式 $x \text{ if } x \% 2 == 0 \text{ else } -x$ 才能根据x计算出确定的结果。

可见，在一个列表生成式中，for前面的if ... else是表达式，而for后面的if是过滤条件，不能带else。

# 生成器generator

2024年2月14日 17:39

- 通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。
- 所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。**在Python中，这种一边循环一边计算的机制，称为生成器：generator。生成器都是Iterator迭代器。**

来自 <<https://www.liaoxuefeng.com/wiki/1016959663602400/1017318207388128>>

- 把一个列表生成式的[]改成()，就创建了一个generator：

```
>>> 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))
>>> g
<generator object <genexpr> at 0x1022ef630>
```

创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator。

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过next()函数获得generator的下一个返回值：

```
>>> 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)
Traceback(most recent call last):File "<stdin>", line 1, in <module>
```

## StopIteration

我们讲过，generator保存的是算法，每次调用next(g)，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用next(g)实在是太变态了，正确的方法是使用for循环，因为generator也是可迭代对象：

## 2.yield法

斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'
```

从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator函数，只需要把print(b)改为yield b就可以了：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```

如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator函数，调用一个generator函数将返回一个generator：

generator函数和普通函数的执行流程不一样。普通函数是顺序执行，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。

举个简单的例子，定义一个generator函数，依次返回数字1, 2, 3：

```
def simple_generator():
    yield 1
    yield 2
    yield 3
```

调用generator函数会创建一个generator对象，多次调用generator函数会创建多个相互独立的generator。

正确的写法是创建一个generator对象，然后不断对这一个generator对象调用next()：

```
gen = simple_generator()
print(next(gen)) # 输出 1
print(next(gen)) # 输出 2
print(next(gen)) # 输出 3
```

回到fib的例子，我们在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator函数后，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代：

```
for value in simple_generator():
    print(value) # 依次输出 1, 2, 3
```

## 小结

generator是非常强大的工具，在Python中，可以简单地把列表生成式改成generator，也可以通过函数实现复杂逻辑的generator。

要理解generator的工作原理，它是在for循环的过程中不断计算出下一个元素，并在适当的条件结束for循环。对于函数改成的generator来说，遇到return语句或者执行到函数体最后一行语句，就是结束generator的指令，for循环随之结束。

- 请注意区分普通函数和generator函数，普通函数调用直接返回结果：

```
>>> r = abs(6)
>>> r
6
```

- generator函数的调用实际返回一个generator对象：

```
>>> g = fib(6)
>>> g
<generator object fib at 0x1022ef948>
```

## 练习杨辉三角形

```
def triangles():
    r=[1]
    while True:
        yield(r)
        r = [1]+[r[i]+r[i+1] for i in range(len(r)-1)]+[1]
# for i in range(len(r)-1): 这部分是一个循环，i 从 0 开始，一直到
# len(r)-2。len(r)-1 是因为我们要访问 r[i] 和 r[i+1]，而最后一个元素的索引是
# len(r)-1，所以循环应该在前一个位置结束。r[i]+r[i+1]: 对于循环中的每个 i，这部
# 分代码取当前行 r 的第 i 个元素和第 i+1 个元素，并将它们相加。
```

```
N = 10
for row in triangles():
    print(row)
    if len(row) > N:
        break
```



# 元组tuple

2024年2月11日 12:22

元组同列表一样，都是可以封装多个、不同类型的元素在内。

但最大的不同点在于：

元组一旦定义完成，就不可修改

所以，当我们需要在程序内封装数据，又不希望封装的数据被篡改，那么元组就非常合适了

## 定义元组

元组定义：定义元组使用小括号，且使用逗号隔开各个数据，数据可以是不同的数据类型。

```
1 # 定义元组字面量
2 (元素, 元素, ..., 元素)
3 # 定义元组变量
4 变量名称 = (元素, 元素, ..., 元素)
5 # 定义空元组
6 变量名称 = ()          # 方式1
7 变量名称 = tuple()     # 方式2
```

元组也支持嵌套：

```
1 # 定义一个嵌套元组
2 t1 = ( (1, 2, 3), (4, 5, 6)
3 print(t1[0][0]) # 结果: 1
```

元组只有一个数据，这个数据后面要添加逗号

```
1 # 定义3个元素的元组
2 t1 = (1, 'Hello', True)
3
4 # 定义1个元素的元组
5 t2 = ('Hello',)      # 注意，必须带有逗号，否则不是元组类型
```

元组由于不可修改的特性，所以其操作方法非常少。

编号	方法	作用
1	index()	查找某个数据，如果数据存在返回对应的下标，否则报错
2	count()	统计某个数据在当前元组出现的次数
3	len(元组)	统计元组内的元素个数

```
1 # 根据下标（索引）取出数据
2 t1 = (1, 2, 'hello')
3 print(t1[2]) # 结果: 'hello'
4
5 # 根据index(), 查找特定元素的第一个匹配项
6 t1 = (1, 2, 'hello', 3, 4, 'hello')
7 print(t1.index('hello')) # 结果: 2
8
9 # 统计某个数据在元组内出现的次数
10 t1 = (1, 2, 'hello', 3, 4, 'hello')
11 print(t1.count('hello')) # 结果: 2
12
13 # 统计元组内的元素个数
14 t1 = (1, 2, 3)
15 print(len(t1)) # 结果 3
```

元组操作的注意事项：

- 不可以修改元组的内容，否则会直接报错

```
1 # 尝试修改元组内容
2 t1 = (1, 2, 3)
3 t1[0] = 5
Traceback (most recent call last):
  File "D:\python-learn\05_数据容器\test.py", line 3, in <module>
    t1[0] = 5
TypeError: 'tuple' object does not support item assignment
```

- 可以修改元组内的list的内容（修改元素、增加、删除、反转等）

```
1 # 尝试修改元组内容
2 t1 = (1, 2, ['itheima', 'itcast'])
3 t1[2][1] = 'best'
4 print(t1) # 结果: (1, 2, ['itheima', 'best'])
```

- 不可以替换list为其它list或其它类型

```
1 # 尝试修改元组内容
2 t1 = (1, 2, ['itheima', 'itcast'])
3 t1[2] = [1, 2, 3]
4 print(t1)
```

```
Traceback (most recent call last):
  File "D:\python-learn\05_数据容器\test.py", line 3, in <module>
    t1[2] = [1, 2, 3]
TypeError: 'tuple' object does not support item assignment
```

元组的遍历：

```
my_tuple = (1, 2, 3, 4, 5)
index = 0
while index < len(my_tuple):
    print(my_tuple[index])
    index += 1
my_tuple = (1, 2, 3, 4, 5)
for i in my_tuple:
    print(i)
```

```
D:\dev\python\python3.10.4\py
1
2
3
4
5
```

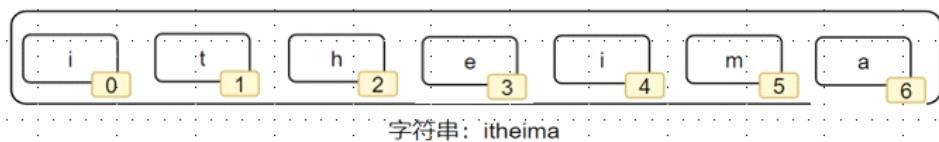
# str字符串容器

2024年2月11日 13:19

编号	操作	说明
1	字符串[下标]	根据下标索引取出特定位置字符
2	字符串.index(字符串)	查找给定字符的第一个匹配项的下标
3	字符串.replace(字符串1, 字符串2)	将字符串内的全部字符串1, 替换为字符串2 不会修改原字符串, 而是得到一个新的
4	字符串.split(字符串)	按照给定字符串, 对字符串进行分隔 不会修改原字符串, 而是得到一个新的列表
5	字符串.strip() 字符串.strip(字符串)	移除首尾的空格和换行符或指定字符串
6	字符串.count(字符串)	统计字符串内某字符串的出现次数, 计数
7	len(字符串)	统计字符串的字符个数

字符串是字符的容器, 一个字符串可以存放任意数量的字符。

如, 字符串: "itheima"



和其它容器如: 列表、元组一样, 字符串也可以通过下标进行访问

从前向后, 下标从0开始

从后向前, 下标从-1开始

```
1 # 通过下标获取特定位置字符
2 name = "itheima"
3 print(name[0]) # 结果i
4 print(name[-1]) # 结果a
```

同元组一样, 字符串是一个: 无法修改的数据容器。

- 修改指定下标的字符 (如: 字符串[0] = "a")
  - 移除特定下标的字符 (如: del 字符串[0]、字符串.remove()、字符串.pop()等)
  - 追加字符等 (如: 字符串.append())
- 均无法完成

- **查找**特定字符串的下标索引值

语法: 字符串.index(字符串)

```
my_str = "itcast and itheima"
print(my_str.index("and")) # 结果7
```

- 字符串的**替换**

语法：字符串.replace(字符串1, 字符串2)

功能：将字符串内的全部：字符串1，替换为字符串2

注意：不是修改字符串本身，而是得到了一个新字符串哦

```
1 name = "itheima itcast"
2 new_name = name.replace("it", "传智")
3
4 print(new_name)      # 结果: 传智heima 传智cast
5 print(name)          # 结果: itheima itcast
```

- 字符串的分割

语法：字符串.split(分隔符字符串)

功能：按照指定的分隔符字符串，将字符串划分为多个字符串，并存入列表对象中

注意：字符串本身不变，而是得到了一个列表对象

```
1 name = "传智播客 传智教育 黑马程序员 博学谷"
2 name_list = name.split(" ")
3
4 print(name_list)      # 结果: ['传智播客', '传智教育', '黑马程序员', '博学谷']
5 print(type(name_list)) # 结果: <class 'list'>
```

可以看到，字符串按照给定的<空格>进行了分割，变成多个子字符串，并存入一个列表对象中。

- 字符串去除前后空格

语法：字符串.strip()

```
my_str = "  itheima and itcast  "
print(my_str.strip())      # 结果: "itheima and itcast"
```

- 字符串的规整操作（去前后指定字符串）

语法：字符串.strip(字符串)

```
my_str = "12itheima and itcast21"
print(my_str.strip("12"))      # 结果: "itheima and itcast"
```

注意，传入的是“12” 其实就是：“1”和“2”都会移除，是按照单个字符。

如果想去除中间用replace

```
print(my_str.replace("and", "")) # 输出 "itcast itheima"
```

- 计数——统计字符串中某字符串的出现次数

语法：字符串.count(字符串)

```
my_str = "itheima and itcast"
print(my_str.count("it"))      # 结果: 2
```

- 统计字符串的长度

语法：len(字符串)

```
1 my_str = "1234 abcd !@#$ 黑马程序员"
2 print(len(my_str))      结果: 20
```

- 字符串也支持while循环和for循环进行遍历

# 序列

2024年2月11日 16:45

**序列**是指：内容连续、有序，可使用下标索引的一类数据容器  
列表、元组、字符串，均可以视为序列。

元素1	元素2	元素3	元素4	元素…	元素n	
0	1	2	3	…	n-1	← 索引（下标）

元素1	元素2	元素3	元素…	元素n-1	元素n	
-n	-(n-1)	-(n-2)	…	-2	-1	← 索引（下标）

如图，序列的典型特征就是：有序并可用下标索引，字符串、元组、列表均满足这个要求

## 切片slicing

序列支持切片，即：列表、元组、字符串，均支持进行切片操作

切片：从一个序列中，取出一个子序列

### 语法：序列[起始下标:结束下标:步长]

表示从序列中，从指定位置开始，依次取出元素，到指定位置结束，得到一个新序列：

起始下标表示从何处开始，**可以留空，留空视作从头开始**

结束下标（不含）表示何处结束，**可以留空，留空视作截取到结尾**

步长表示，依次取元素的间隔

- 步长1表示，一个个取元素（**步长默认是1，可以省略不写**）
- 步长2表示，每次跳过1个元素取
- 步长N表示，每次跳过N-1个元素取
- 步长为负数表示，反向取（注意，起始下标和结束下标也要反向标记）

注意，此操作不会影响序列本身，而是会得到一个新的序列（列表、元组、字符串）

`[:2]`表示取前两个元素

```
# 对str进行切片, 从头开始, 到最后结束, 步长-1
my_str = "01234567"
result4 = my_str[::-1]           # 等同于将序列反转了
print(f"结果4: {result4}")
```

```
# 对列表进行切片, 从3开始, 到1结束, 步长-1
my_list = [0, 1, 2, 3, 4, 5, 6]
result5 = my_list[3:1:-1]
print(f"结果5: {result5}")
```

```
# 对元组进行切片, 从头开始, 到尾结束, 步长-2
my_tuple = (0, 1, 2, 3, 4, 5, 6)
result6 = my_tuple[::-2]
print(f"结果6: {result6}")
```

# 集合set

2024年2月11日 17:30

编号	操作	说明
1	集合.add(元素)	集合内添加一个元素
2	集合.remove(元素)	移除集合内指定的元素
3	集合.pop()	从集合中随机取出一个元素
4	集合.clear()	将集合清空
5	集合1.difference(集合2)	得到一个新集合, 内含2个集合的差集 原有的2个集合内容不变
6	集合1.difference_update(集合2)	在集合1中, 删除集合2中存在的元素 集合1被修改, 集合2不变
7	集合1.union(集合2)	得到1个新集合, 内含2个集合的全部元素 原有的2个集合内容不变
8	len(集合)	得到一个整数, 记录了集合的元素数量

- 列表可修改、支持**重复**元素且有序
- 元组、字符串不可修改、支持**重复**元素且有序

局限就在于：它们都支持重复元素。

如果场景需要对内容做去重处理，列表、元组、字符串就不方便了。

而**集合**，最主要的特点就是：不支持元素的重复（自带去重功能）、并且内容无序

集合的定义：

```
1 # 定义集合字面量
2 {元素, 元素, ..., 元素}
3 # 定义集合变量
4 变量名称 = {元素, 元素, ..., 元素}
5 # 定义空集合
6 变量名称 = set()
```

和列表、元组、字符串等定义基本相同：

列表使用：[]

元组使用：()

字符串使用：""

集合使用：{}  
{}  
{ }  
{ , , , }

```
1 names = {"黑马程序员", "传智播客", "itcast", "itheima", "黑马程序员", "传智播客"}
2 print(names)
D:\dev\python\python3.10.4\python.exe D:/python-learn/05_数据容器/test.py
{'itheima', '黑马程序员', '传智播客', 'itcast'}
```

因为要对元素做去重处理

所以无法保证顺序和创建的时候一致

首先，因为集合是无序的，所以**集合不支持：下标索引访问**

但是集合和列表一样，是允许修改的，所以我们来看看集合的修改方法。

- 添加新元素

语法：集合.add(元素)。将指定元素，添加到集合内

结果：集合本身被修改，添加了新元素

```
1 my_set = {"Hello", "World"}  
2 my_set.add("itheima")  
3 print(my_set) # 结果 {'Hello', 'itheima', 'World'}
```

- 移除元素

语法：集合.remove(元素)，将指定元素，从集合内移除

结果：集合本身被修改，移除了元素

```
1 my_set = {"Hello", "World", "itheima"}  
2 my_set.remove("Hello")  
3 print(my_set) # 结果 {'World', 'itheima'}
```

- 从集合中随机取出元素

语法：集合.pop()，功能，从集合中随机取出一个元素

结果：会得到一个元素的结果。同时集合本身被修改，元素被移除

```
1 my_set = {"Hello", "World", "itheima"}  
2 element = my_set.pop()  
3 print(my_set) # 结果 {'World', 'itheima'}  
4 print(element) # 结果 'Hello'
```

- 清空集合

语法：集合.clear()，功能，清空集合

结果：集合本身被清空

```
1 my_set = {"Hello", "World", "itheima"}  
2 my_set.clear()  
3 print(my_set) # 结果: set() 空集合
```

- 取出2个集合的差集

语法：集合1.difference(集合2)，功能：取出集合1和集合2的差集（集合1有而集合2没有的）

结果：得到一个新集合，集合1和集合2不变

```
1 set1 = {1, 2, 3}
2 set2 = {1, 5, 6}
3 set3 = set1.difference(set2)
4 print(set3)      # 结果: {2, 3}      得到的新集合
5 print(set1)      # 结果: {1, 2, 3} 不变
6 print(set2)      # 结果: {1, 5, 6} 不变
```

- 消除2个集合的差集

语法：集合1.difference\_update(集合2)

功能：对比集合1和集合2，在集合1内，删除和集合2相同的元素。

结果：集合1被修改，集合2不变

```
1 set1 = {1, 2, 3}
2 set2 = {1, 5, 6}
3 set1.difference_update(set2)
4 print(set1)      # 结果: {2, 3}
5 print(set2)      # 结果: {1, 5, 6}
```

- 2个集合合并

语法：集合1.union(集合2)

功能：将集合1和集合2组合成新集合

结果：得到新集合，集合1和集合2不变

```
1 set1 = {1, 2, 3}
2 set2 = {1, 5, 6}
3 set3 = set1.union(set2)
4 print(set3)      # 结果: {1, 2, 3, 5, 6}, 新集合
5 print(set1)      # 结果: {1, 2, 3}, set1不变
6 print(set2)      # 结果: {1, 5, 6}, set2不变
```

- 查看集合的元素数量

语法：len(集合)

功能：统计集合内有多少元素

结果：得到一个整数结果

```
1 set1 = {1, 2, 3}
2 print(len(set1))    # 结果3
```

- 集合for循环遍历

要注意：集合不支持下标索引，所以也就不支持使用while循环。

```
1 set1 = {1, 2, 3}
2 for i in set1:
3     print(i)
4
5 # 结果
6 1
7 2
8 3
```

# 字典dict

2024年2月11日 19:14

```
1 # 定义字典字面量
2 {key: value, key: value, ...., key: value}
3 # 定义字典变量
4 my_dict = {key: value, key: value, ...., key: value}
5 # 定义空字典
6 my_dict = {}          # 空字典定义方式1
7 my_dict = dict()      # 空字典定义方式2
```

记录学生成绩的需求, 可以如下记录:

```
1 stu_score = {"王力鸿": 99, "周杰轮": 88, "林俊节": 77}
```

## • 字典的注意事项

字典同集合一样, 不可以使用下标索引  
键值对的Key和Value可以是任意类型 (Key不可为字典)  
字典内Key不允许重复, 重复添加等同于覆盖原有数据

字典可以通过Key值来取得对应的Value

```
1 # 语法, 字典 [Key] 可以取到对应的value
2 stu_score = {"王力鸿": 99, "周杰轮": 88, "林俊节": 77}
3 print(stu_score["王力鸿"])      # 结果99
4 print(stu_score["周杰轮"])      # 结果88
5 print(stu_score["林俊节"])      # 结果77
```

字典的Key和Value可以是任意数据类型 (Key不可为字典)

姓名	语文	数学	英语
王力鸿	77	66	33
周杰轮	88	86	55
林俊节	99	96	66

代码:

```
1 stu_score = {"王力鸿": {"语文": 77, "数学": 66, "英语": 33}, "周杰轮": {"语文": 88, "数学": 86, "英语": 55},
  "林俊节": {"语文": 99, "数学": 96, "英语": 66}}
```

优化一下可读性, 可以写成:

```
1 stu_score = {
2     "王力鸿": {"语文": 77, "数学": 66, "英语": 33},
3     "周杰轮": {"语文": 88, "数学": 86, "英语": 55},
4     "林俊节": {"语文": 99, "数学": 96, "英语": 66}
5 }
```

字典获取

```
1 stu_score = {
2     "王力鸿": {"语文": 77, "数学": 66, "英语": 33},
3     "周杰轮": {"语文": 88, "数学": 86, "英语": 55},
4     "林俊节": {"语文": 99, "数学": 96, "英语": 66}
5 }
6 print(stu_score["王力鸿"])          # 结果: {"语文": 77, "数学": 66, "英语": 33}
7 print(stu_score["王力鸿"]["语文"])    # 结果: 77
8 print(stu_score["周杰轮"]["数学"])    # 结果: 86
```

```
std1 = {"name": "Michael", "score": 98}
std2 = {"name": "Bob", "score": 81}

1 usage
def print_score(std):
    print("%s:%s" % (std["name"], std["score"]))

print_score(std1)
```

test1 ×

E:\python\黑马python练习\pythonProject1\.venv\Script

Michael:98

# 字典操作

2024年2月11日 19:37

编号	操作	说明
1	字典[Key]	获取指定Key对应的Value值
2	字典[Key] = Value	添加或更新键值对
3	字典.pop(Key)	取出Key对应的Value并在字典内删除此Key的键值对
4	字典.clear()	清空字典
5	字典.keys()	获取字典的全部Key，可用于for循环遍历字典
6	len(字典)	计算字典内的元素数量

- 新增元素

语法: 字典[Key] = Value, 结果: 字典被修改, 新增了元素

```
1 stu_score = {  
2     "王力鸿": 77,  
3     "周杰轮": 88,  
4     "林俊节": 99  
5 }  
6 # 新增: 张学油的考试成绩  
7 stu_score['张学油'] = 66  
8 print(stu_score)      # 结果: {'王力鸿': 77, '周杰轮': 88, '林俊节': 99, '张学油': 66}  
  
stu_score = {  
    "wanglihong": {"chinese": 77, "math": 66, "english": 33},  
    "dudu": {"chinese": 77, "math": 62, "english": 33},  
    "chougoupi": {"chinese": 71, "math": 66, "english": 33}  
}  
stu_score["lulu"] = {"chinese": 71, "math": 66, "english": 33}  
print(stu_score)
```

- 更新元素

语法: 字典[Key] = Value, 结果: 字典被修改, 元素被更新

注意: 字典Key不可以重复, 所以对已存在的Key执行上述操作, 就是更新Value值

```
1 stu_score = {  
2     "王力鸿": 77,  
3     "周杰轮": 88,  
4     "林俊节": 99  
5 }  
6 # 更新: 王力鸿的考试成绩  
7 stu_score['王力鸿'] = 100  
8 print(stu_score)      # 结果: {'王力鸿': 100, '周杰轮': 88, '林俊节': 99}
```

- 删除元素

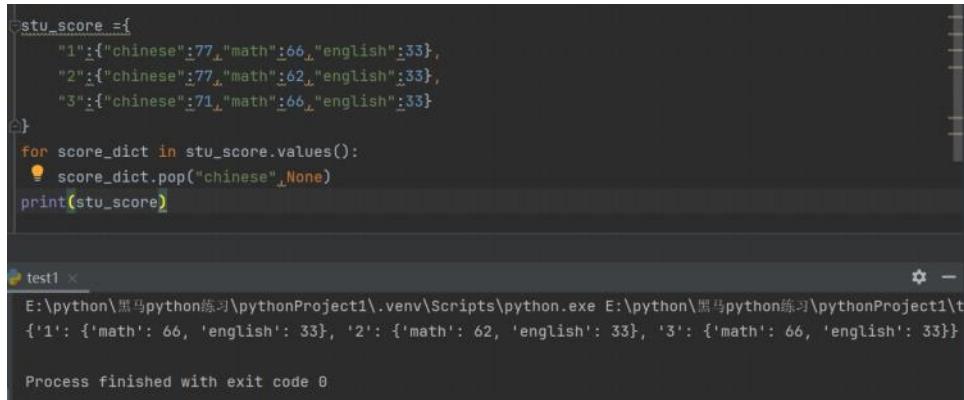
语法: 字典.pop(Key), 结果: 获得指定Key的Value, 同时字典被修改, 指定Key的数据被删除

```

1 stu_score = {
2     "王力鸿": 77,
3     "周杰轮": 88,
4     "林俊节": 99
5 }
6 value = stu_score.pop("王力鸿")
7 print(value)      # 结果: 77
8 print(stu_score)  # 结果: {"周杰轮": 88, "林俊节": 99}

```

### 删除嵌套



```

stu_score = {
    "1": {"chinese": 77, "math": 66, "english": 33},
    "2": {"chinese": 77, "math": 62, "english": 33},
    "3": {"chinese": 71, "math": 66, "english": 33}
}
for score_dict in stu_score.values():
    score_dict.pop("chinese", None)
print(stu_score)

```

test1

```

E:\python\黑马python练习\pythonProject1\.venv\Scripts\python.exe E:\python\黑马python练习\pythonProject1\t
{'1': {'math': 66, 'english': 33}, '2': {'math': 62, 'english': 33}, '3': {'math': 66, 'english': 33}}

```

Process finished with exit code 0

- 清空字典

语法: 字典.clear(), 结果: 字典被修改, 元素被清空

```

1 stu_score = {
2     "王力鸿": 77,
3     "周杰轮": 88,
4     "林俊节": 99
5 }
6 stu_score.clear()
7 print(stu_score)      # 结果: {}

```

- 获取全部的key

语法: 字典.keys(), 结果: 得到字典中的全部Key

```

1 stu_score = {
2     "王力鸿": 77,
3     "周杰轮": 88,
4     "林俊节": 99
5 }
6 keys = stu_score.keys()
7 print(keys)      # 结果: dict_keys(['王力鸿', '周杰轮', '林俊节'])

```

- 遍历字典

只能用for循环, 字典不支持下标索引, 所以不可以用while循环

- 计算字典内的全部元素 (键值对) 数量

语法: len(字典)

结果: 得到一个整数, 表示字典内元素 (键值对) 的数量

```
1 stu_score = {  
2     "王力鸿": 77,  
3     "周杰轮": 88,  
4     "林俊节": 99  
5 }  
6 print(len(stu_score))      # 结果: 3
```

# 容器总结

2024年2月11日 20:51

除了下标索引这个共性外，还可以通用类型转换

`List()`

将给定容器转换为列表除了下标索引这个共性外表

`str(容器)`

将给定容器转换为字符串

`tuple(容器)`

将给定容器转换为元组

`set(容器)`

将给定容器转换为集合

通用排序功能

`sorted(容器, [reverse=True])`

默认是`False`

将给定容器进行排序

注意，排序后都会得到列表（list）对象。

正向排序：

```
# 进行容器的排序
my_list = [3, 1, 2, 5, 4]
my_tuple = (3, 1, 2, 5, 4)
my_str = "bdcefga"
my_set = {3, 1, 2, 5, 4}
my_dict = {"key3": 1, "key1": 2, "key2": 3, "key5": 4, "key4": 5}

print(f"列表对象的排序结果: {sorted(my_list)}")
print(f"元组对象的排序结果: {sorted(my_tuple)}")
```

元组转集合的结果是: {1, 2, 3, 4, 5}  
字符串转集合结果是: {'b', 'c', 'd', 'e', 'f', 'a', 'g'}  
集合转集合的结果是: {1, 2, 3, 4, 5}  
字典转集合的结果是: {'key4', 'key5', 'key1', 'key2', 'key3'}  
列表对象的排序结果: [1, 2, 3, 4, 5]  
元组对象的排序结果: [1, 2, 3, 4, 5]  
字符串对象的排序结果: ['a', 'b', 'c', 'd', 'e', 'f', 'g']  
集合对象的排序结果: [1, 2, 3, 4, 5]  
字典对象的排序结果: ['key1', 'key2', 'key3', 'key4', 'key5']

之后是不是12345 abcdefg

结果反转

```
print(f"列表对象的反向排序结果: {sorted(my_list, reverse=True)}")  
print(f"元组对象的反向排序结果: {sorted(my_tuple, reverse=True)}")  
print(f"字符串对象反向的排序结果: {sorted(my_str, reverse=True)}")  
print(f"集合对象的反向排序结果: {sorted(my_set, reverse=True)}")  
print(f"字典对象的反向排序结果: {sorted(my_dict, reverse=True)}")
```

15\_数据容器通用功能

```
元组对象的排序结果: [1, 2, 3, 4, 5]  
字符串对象的排序结果: ['a', 'b', 'c', 'd', 'e', 'f', 'g']  
集合对象的排序结果: [1, 2, 3, 4, 5]  
字典对象的排序结果: ['key1', 'key2', 'key3', 'key4', 'key5']  
列表对象的反向排序结果: [5, 4, 3, 2, 1]  
元组对象的反向排序结果: [5, 4, 3, 2, 1] I  
字符串对象反向的排序结果: ['g', 'f', 'e', 'd', 'c', 'b', 'a']  
集合对象的反向排序结果: [5, 4, 3, 2, 1]  
字典对象的反向排序结果: ['key5', 'key4', 'key3', 'key2', 'key1']
```

功能	描述
通用for循环	遍历容器（字典是遍历key）
max	容器内最大元素
min()	容器内最小元素
len()	容器元素个数
list()	转换为列表
tuple()	转换为元组
str()	转换为字符串
set()	转换为集合
sorted(序列, [reverse=True])	排序, reverse=True表示降序 得到一个排好序的列表

# 函数

2024年2月7日 20:49

函数：是组织好的，可重复使用的，用来实现特定功能的代码段。

input()、print()、str()、int()等都是Python的内置函数

函数的定义：

```
def 函数名(传入参数):  
    函数体  
    return 返回值
```

函数的调用：

函数名(参数)

1. 传入参数如不需要，可以省略，但括号必须有
  1. 返回值如不需要，可以省略，但括号必须有
  2. 函数必须先定义后使用

## 传入参数

基于函数的定义语法：

```
def 函数名(传入参数):  
    函数体  
    return 返回值
```

可以有如下函数定义：

```
def add(x, y):  
    result = x + y  
    print(f" {x} + {y} 的结果是: {result}")
```

实现了，每次计算的是 $x + y$ ，而非固定的 $1 + 2$

$x + y$ 的值，可以在调用函数的时候指定。

函数定义中，提供的 $x$ 和 $y$ ，称之为：形式参数（形参），表示函数声明将要使用2个参数

参数之间使用逗号进行分隔

函数调用中，提供的5和6，称之为：实际参数（实参），表示函数执行时真正使用的参数值

传入的时候，按照顺序传入数据，使用逗号分隔

## 必选参数和默认参数

<https://www.liaoxuefeng.com/wiki/1016959663602400/10172616304258>

## 88

power(x, n)函数，可以计算任意n次方：

```
def power(x, n):  
    s = 1  
    while n > 0:  
        n = n - 1  
        s = s * x  
    return s
```

新的power(x, n)函数定义没有问题，但是，旧的调用代码失败了，原因是增加了  
一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

Python的错误信息很明确：调用函数power()缺少了一个位置参数n。

这个时候，默认参数就派上用场了。由于我们经常计算 $x^2$ ，所以，完全可以把第二个  
参数n的默认值设定为2：

```
def power(x, n=2):  
    s = 1  
    while n > 0:  
        n = n - 1  
        s = s * x  
    return s
```

这样，当我们调用power(5)时，相当于调用power(5, 2)：

```
>>> power(5)  
25  
>>> power(5, 2)  
25
```

设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，当函数有多个参数时，把变化大的参数放前面，  
变化小的参数放后面。变化小的参数就可以作为默认参数。{}

# 函数的说明文档

2024年2月8日 19:15

给函数添加说明文档，辅助理解函数的作用。

```
def func(x, y):
    """
    函数说明
    :param x: 形参x的说明
    :param y: 形参y的说明
    :return: 返回值的说明
    """


```

函数体

```
    return 返回值
```

内容应写在函数体之前

# 函数的嵌套

2024年2月8日 19:19

```
def func_b():
    print("---2---")

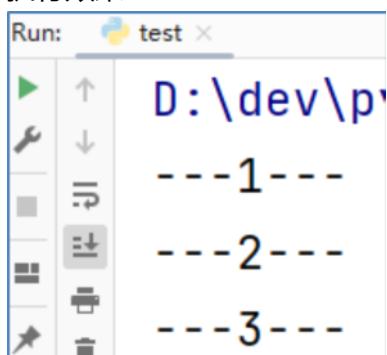
def func_a():
    print("---1---")

    func_b()

    print("---3---")

# 调用函数func_a
func_a()
```

执行效果



```
Run: test
D:\dev\p
---1---
---2---
---3---
```

# 函数的多种参数

2024年2月14日 15:53

**位置参数**: 调用函数时根据函数定义的参数位置来传递参数

```
def user_info(name, age, gender):  
    print(f'您的名字是{name}, 年龄是{age}, 性别是{gender}')  
  
user_info('TOM', 20, '男')
```

注意:

传递的参数和定义的参数的顺序及个数必须一致

**关键字参数**: 函数调用时通过“键=值”形式传递参数.

**作用**: 可以让函数更加清晰、容易使用，同时也清除了参数的顺序需求.

```
def user_info(name, age, gender)  
    print(f"您的名字是: {name}, 年龄是: {age}, 性别是: {gender}")  
  
# 关键字传参  
user_info(name="小明", age=20, gender="男")  
# 可以不按照固定顺序  
user_info(age=20, gender="男", name="小明")  
# 可以和位置参数混用，位置参数必须在前，且匹配参数顺序  
user_info("小明", age=20, gender="男")
```

注意:

函数调用时，如果有位置参数时，**位置参数必须在关键字参数的前面，但关键字参数之间不存在先后顺序**

## 缺省参数(默认参数)

缺省参数: 缺省参数也叫默认参数，用于定义函数，为参数提供默认值，调用函数时可不传该默认参数的值 (注意: 所有位置参数必须出现在默认参数前，包括函数定义和调用) .

**作用**: 当调用函数时没有传递参数，就会使用默认是用缺省参数对应的值.

```
def user_info(name, age, gender='男'):  
    print(f'您的名字是{name}, 年龄是{age}, 性别是{gender}')  
  
user_info('TOM', 20)  
user_info('Rose', 18, '女')
```

注意：

函数调用时，如果为缺省参数传值则修改默认参数值，否则使用这个默认值

## 不定长参数

不定长参数：不定长参数也叫可变参数。用于不确定调用的时候会传递多少个参数（不传参也可以）的场景。

作用：当调用函数时不确定参数个数时，可以使用不定长参数

\* args 和 kwargs 是规范要求，想用别的替代也可以

不定长参数的类型：

① 位置传递

② 关键字传递

```
def user_info(*args):  
    print(args)  
  
# ('TOM',)  
user_info('TOM')  
# ('TOM', 18)  
user_info('TOM', 18)
```

注意：

传进的所有参数都会被 args 变量收集，它会根据传进参数的位置合并为一个 **元组(tuple)**，  
**args 是元组类型，这就是位置传递**

## 关键字传递

```
def user_info(**kwargs):  
    print(kwargs)  
  
# {'name': 'TOM', 'age': 18, 'id': 110}  
user_info(name='TOM', age=18, id=110)
```

**kwargs 是关键词传递**

只要按照 key=value 的值传递就可以了

注意：

参数是“键=值”形式的形式的情况下,所有的“键=值”都会被kwargs接受,同时会根据“键=值”组成**字典**.

# 高阶函数

2024年2月14日 19:42

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

举例：

```
def add (x,y,f):  
    return f(x)+f(y)  
  
print(add(-5,6,abs))
```

根据函数定义，我们可以推导计算过程为：

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

# map函数和reduce函数

2024年2月14日 19:57

**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。**

举例说明，比如我们有一个函数 $f(x)=x^2$ ，要把这个函数作用在一个list [1, 2, 3, 4, 5, 6, 7, 8, 9]上，就可以用map()实现如下：

```
def f(x):  
...     return x * x  
...  
>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])  
>>> print(list(r))  
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

比如，把这个list所有数字转为字符串：

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

## reduce函数

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，其效果就是：

$\text{reduce}(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)$

比方说对一个序列求和，就可以用reduce实现：

```
from functools import reduce  
def add(x, y):  
    return x + y  
print(reduce(add, [1, 3, 5, 7, 9]))
```

## map函数练习

1.利用map()函数，把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入：['adam', 'LISA', 'barT']，输出：['Adam', 'Lisa', 'Bart']：

```
L1 = ['adam', 'LISA', 'barT']  
def normalize_name(L1):  
    return L1.lower().capitalize()
```

```
L2 = list(map(normalize_name, L1))  
print(L2)
```

**#在 Python 中，map() 函数返回一个迭代器，而不是一个列表。这意味着当你使用 map() 时，它不会立即对输入的序列进行操作并生成结果列表，而是返回一个特殊的迭代器对象，这个对象会在你迭代它时逐一产生结果。**

当你需要将 map() 的结果保存到一个列表中时，你需要将迭代器转换为列表。这就是为什么在表达式 `normalized_names = list(map(normalize_name, names))` 中需要加 list 的原因。

## reduce函数练习

1. Python提供的sum()函数可以接受一个list并求和, 请编写一个prod()函数, 可以接受一个list并利用reduce()求积:

```
from functools import reduce
```

```
def muliply(x,y):
```

```
    return x * y
```

```
def prod(lst):
```

```
    return reduce(muliply,lst)
```

```
print(prod([1,2,3,4]))
```

# filter函数

2024年2月14日 21:13

Python内建的filter()函数用于过滤序列。

和map()类似，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。

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

```
def is_odd(n):  
    return n % 2 == 1
```

```
list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
```

把一个序列中的空字符串删掉，可以这么写：

```
def not_empty(s):  
    return s and s.strip()  
list(filter(not_empty, ['A', '', 'B', None, 'C', '']))
```

注意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，它会强制迭代器完成所有计算并将结果存储在一个列表中。

练习：

1.用filter求素数

# sorted函数

2024年2月15日 10:36

## 排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个dict呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。

来自 <<https://www.liaoxuefeng.com/wiki/1016959663602400/1017408670135712>>

Python内置的sorted()函数就可以对list进行排序：

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

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，例如按绝对值大小排序：

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

我们再看一个字符串排序的例子：

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

默认情况下，对字符串排序，是按照ASCII的大小比较的，由于'Z' < 'a'，结果，大写字母Z会排在小写字母a的前面。

现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能用一个key函数把字符串映射为忽略大小写排序即可。忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

这样，我们给sorted传入key函数，即可实现忽略大小写的排序：

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

要进行反向排序，不必改动key函数，可以传入第三个参数reverse=True：

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

练习：

假设我们用一组tuple表示学生名字和成绩：

```
L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]
```

# 匿名函数

2024年2月14日 16:18

函数作为参数传递

```
def test_func(compute):  
    result = compute(1, 2)  
    print(result)  
  
def compute(x, y):  
    return x + y  
  
test_func(compute)      # 结果: 3  
  
def test_func(compute):  
    result = compute(1, 2)  
    print(result)  
  
def compute(x, y):  
    return x * y  
  
test_func(compute)      # 结果: 2  
  
def test_func(compute):  
    result = compute(1, 2)  
    print(result)  
  
def compute(x, y):  
    return x - y  
  
test_func(compute)      # 结果: -1
```

函数compute，作为参数，传入了test\_func函数中使用。

test\_func需要一个函数作为参数传入，这个函数需要接收2个数字进行计算，计算逻辑由这个被传入函数决定

compute函数接收2个数字对其进行计算，compute函数作为参数，传递给了test\_func函数使用最终，在test\_func函数内部，由传入的compute函数，完成了对数字的计算操作所以，这是一种，计算逻辑的传递，而非数据的传递。

就像上述代码那样，不仅仅是相加，相减、相除、等任何逻辑都可以自行定义并作为函数传入。

# 偏函数

2024年2月15日 18:41

Python的functools模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下：

int()函数可以把字符串转换为整数，当仅传入字符串时，int()函数默认按十进制转换：

```
int('12345')  
12345
```

但int()函数还提供额外的base参数，默认值为10。如果传入base参数，就可以做N进制的转换：

```
int('12345', base=8)  
5349  
int('12345', 16)  
74565
```

假设要转换大量的二进制字符串，每次都传入int(x, base=2)非常麻烦，于是，我们想到，可以定义一个int2()的函数，默认把base=2传进去：

```
def int2(x, base=2):  
    return int(x, base)
```

这样，我们转换二进制就非常方便了：

```
>>> int2('1000000')  
64  
>>> int2('1010101')  
85
```

functools.partial就是帮助我们创建一个偏函数的，不需要我们自己定义int2()，可以直接使用下面的代码创建一个新的函数int2：

```
import functools  
int2 = functools.partial(int, base=2)  
print(int2("100000"))
```

简单总结**functools.partial的作用就是，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。

注意到上面的新的int2函数，仅仅是把base参数重新设定默认值为2，但也可以在函数调用时传入其他值：

```
int2('1000000', base=10)  
1000000
```

创建偏函数时，实际上可以接收函数对象、\*args和\*\*kw这3个参数，当传入：

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

实际上固定了int()函数的关键字参数base，也就是：

```
int2('10010')
```

相当于：

```
kw = { 'base': 2 }
```

```
int('10010', **kw)
```

函数调用等价于 int('10010', base=2)。

再比如：

```
max2 = functools.partial(max, 10)
```

functools.partial(max, 10) 的用法是创建一个新的函数，该函数在调用时会将 max 函数的第一个参数固定为 10。换句话说，这个新的函数将始终比较其接收到的参数和 10，并返回两者之间的最大值。

还比如：

```
# 定义一个取余函数，默认和2取余；
```

```
def mod(x,y=2):
```

```
    # 返回 True 或 False
```

```
    return x % y == 0
```

```
# 假设我们要计算和3取余，如果不使用partial()函数，那么我们每次调用mod()函
```

```
数时，都要写y=3
```

```
mod(4,y=3)
```

```
mod(6,y=3)
```

```
# 使用partial()函数
```

```
from functools import partial
```

```
mod_3 = partial(mod,y=3)
```

```
mod_3(4)
```

# json数据格式

2024年2月19日 11:08

JSON是一种轻量级的数据交互格式。可以按照JSON指定的格式去组织和封装数据

JSON本质上是一个带有特定格式的字符串

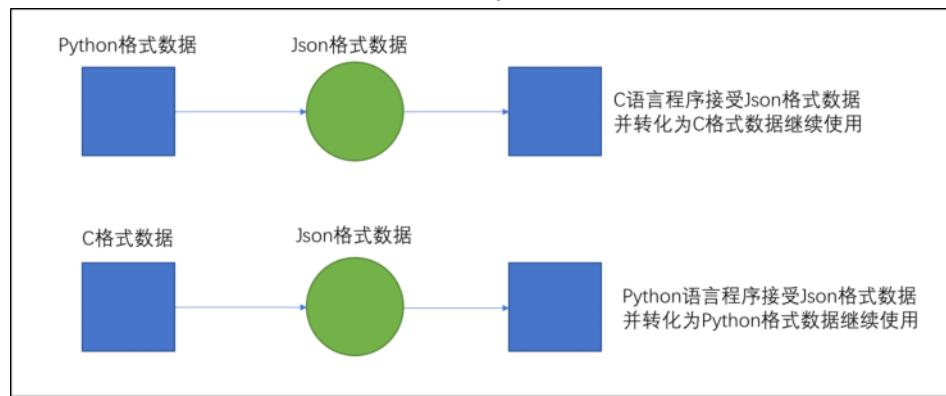
主要功能：json就是一种在各个编程语言中流通的数据格式，负责不同编程语言中的数据传递和交互。类似于：

国际通用语言-英语

中国56个民族不同地区的通用语言-普通话

各种编程语言存储数据的容器不尽相同，在Python中有字典dict这样的数据类型，而其它语言可能没有对应的字典。

为了让不同的语言都能够相互通用的互相传递数据，JSON就是一种非常良好的中转数据格式。如下图，以Python和C语言互传数据为例：



## Python数据和Json数据的相互转化

```
# 导入json模块
import json

# 准备符合格式json格式要求的python数据
data = [{"name": "老王", "age": 16}, {"name": "张三", "age": 20}]

# 通过 json.dumps(data) 方法把python数据转化为了 json数据，如果有
# 中文可以带上ensure_ascii = False
data = json.dumps(data)

# 通过 json.loads(data) 方法把json数据转化为了 python数据
data = json.loads(data)
```

# 属性函数

2024年2月20日 10:39

getattr(), setattr(), 和 hasattr() 是 Python 中的内置函数，它们允许你动态地获取、设置和检查对象的属性。

## 1getattr(object, name[, default])

"get" 和 "attr" 两个词组合而成的。"get" 通常用于表示获取或检索操作，而 "attr" 是 "attribute" (属性) 的缩写。因此，getattr() 函数用于获取对象的属性。

object: 对象，你想从中获取属性的对象。

name: 字符串，属性的名称。

default (可选) : 如果属性不存在，则返回此默认值。

描述: getattr() 函数返回对象指定的属性值。如果属性不存在并且没有提供默认值，则会引发 AttributeError 异常。

示例:

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

p1 = Person(name="dudu", age=23)
print(getattr(p1, "name"))
print(getattr(p1, "age"))
print(getattr(p1, "gender", "Unknown"))
```

test1 ×

E:\python\黑马python练习\pythonProject1\pythonProject1\test1.py

dudu

23

Unknown

## 2setattr(object, name, value)

这个函数名由 "set" 和 "attr" 两个词组合而成。"set" 通常用于表示设置或赋值操作，而 "attr" 同样是 "attribute" 的缩写。因此，setattr() 函数用于设置对象的属性。

object: 对象，你想设置属性的对象。

name: 字符串，属性的名称。

value: 你想设置的新属性值。

描述: setattr() 函数对应地设置对象的属性值。如果属性已经存在，则其值会被更新；如果属性不存在，则会创建该属性。

示例：

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

p1 = Person(name="dudu", age=23)
setattr(p1, "name", "aouou")
setattr(p1, "gender", "female")
print(p1.name)
print(p1.gender)
```

test1 ×  
E:\python\黑马python练习\pythonProject1\.ve  
  \pythonProject1\test1.py  
aouou  
female

### 3.hasattr(object, name)

这个函数名由 "has" 和 "attr" 两个词组合而成。"has" 通常用于表示存在性检查，而 "attr" 同样是 "attribute" 的缩写。因此，hasattr() 函数用于检查对象是否具有某个属性。

object: 对象，你想检查属性的对象。

name: 字符串，你想检查的属性名称。

描述: hasattr() 函数检查对象是否具有指定的属性。如果属性存在，则返回 True；否则返回 False。

示例：

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

p1 = Person(name="dudu", age=23)
print(hasattr(p1, "name"))
print(hasattr(p1, "gender"))


```

test1

```
E:\python\黑马python练习\pythonProject1\pythonProject1\test1.py
True
False
```

要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以直写：

```
sum = obj.x + obj.y
```

就不要写：

```
sum = getattr(obj, 'x') + getattr(obj, 'y')
```

一个正确的用法的例子如下：

```
def readImage(fp):
    if hasattr(fp, 'read'):
        return readData(fp)
    return None
```

假设我们希望从文件流fp中读取图像，我们首先要判断该fp对象是否存在read方法，如果存在，则该对象是一个流，如果不存在，则无法读取。hasattr()就派上了用场。

# 文件读写

2024年2月16日 17:33

# 文件操作

2024年2月16日 9:25

操作	功能
文件对象 = open(file, mode, encoding)	打开文件获得文件对象
文件对象.read(num)	读取指定长度字节 不指定num读取文件全部
文件对象.readline()	读取一行
文件对象.readlines()	读取全部行, 得到列表
for line in 文件对象	for循环文件行, 一次循环得到一行数据
文件对象.close()	关闭文件对象
with open() as f	通过with open语法打开文件, 可以自动关闭

## 打开/创建文件

使用open函数, 可以打开一个已经存在的文件, 或者创建一个新文件, 语法如下

**open(name, mode, encoding)**

name: 是要打开的目标文件名的字符串(可以包含文件所在的具体路径)。

mode: 设置打开文件的模式(访问模式): 只读、写入、追加等。

encoding: 编码格式 (推荐使用UTF-8)

### 示例

```
f = open('python.txt', 'r', encoding='UTF-8')
```

# 实际上open函数在encoding之前还有一个buffering=-1的形参, 省略了, 所以encoding的顺序不是第三位, 所以不能用位置参数, 用关键字参数直接指定

注意: 此时的`f`是`open`函数的文件对象, 对象是Python中一种特殊的数据类型, 拥有属性和方法, 可以使用对象.属性或对象.方法对其进行访问, 后续面向对象课程会给大家进行详细的介绍。

模式	描述
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
w	打开一个文件只用于写入。如果该文件已存在则打开文件, 并从开头开始编辑, <b>原有内容会被删除</b> 。 如果该文件不存在, <b>创建新文件</b> 。
a	打开一个文件用于追加。如果该文件已存在, <b>新的内容将会被写入到已有内容之后</b> 。 如果该文件不存在, <b>创建新文件</b> 进行写入。

## 读取文件

read()方法:

### 1.文件对象.read(num)

num表示要从文件中读取的数据的长度（单位是字节），如果没有传入num，那么就表示读取文件中所有的数据。

```
f=open(r'C:\Users\嘟嘟\Desktop\word.txt','r')
print(f.read().count("itheima"))
```

## 2.readlines()方法：

readlines可以按照行的方式把整个文件中的内容进行一次性读取，并且返回的是一个列表，其中每一行的数据为一个元素。

```
f = open('python.txt')
content = f.readlines()

# ['hello world\n', 'abcdefg\n', 'aaa\n', 'bbb\n', 'ccc']
print(content)

# 关闭文件
f.close()
```

## 3.readline()方法：一次读取一行内容

```
f = open('python.txt')

content = f.readline()
print(f'第一行: {content}')

content = f.readline()
print(f'第二行: {content}')

# 关闭文件
f.close()
```

## 4.for循环读取文件行

```
for line in open("python.txt", "r"):
    print(line)

# 每一个line临时变量，就记录了文件的一行数据
```

## 关闭文件

```
f = open("python.txt", "r")

f.close()

# 最后通过close，关闭文件对象，也就是关闭对文件的占用
# 如果不调用close,同时程序没有停止运行，那么这个文件将一直被Python程序占用。
```

## with open

```
with open("python.txt", "r") as f:
    f.readlines()
```

```
# 通过在with open的语句块中对文件进行操作
# 操作完成后自动关闭close文件，避免遗忘掉close方法
```

# 文件的写入

2024年2月16日 11:06

**文件如果不存在，使用“w”模式，会创建新文件**  
**文件如果存在，使用“w”模式，会将原有内容清空**

```
# 1. 打开文件
f = open('python.txt', 'w')

# 2. 文件写入
f.write('hello world')

# 3. 内容刷新
f.flush()
```

注意：

- 直接调用write，内容并未真正写入文件，而是会积攒在程序的内存中，称之为缓冲区
- 当调用flush的时候，内容会真正写入文件
- 这样做是避免频繁的操作硬盘，导致效率下降（攒一堆，一次性写磁盘）

# 文件的追加写入

2024年2月16日 11:17

```
# 1. 打开文件，通过a模式打开即可
f = open('python.txt', 'a')
```

```
# 2. 文件写入
f.write('hello world')
```

```
# 3. 内容刷新
f.flush()
```

**a模式，文件不存在会创建文件**

**a模式，文件存在会在最后，追加写入文件**

# 异常的捕获

2024年2月16日 17:33

## 捕获常规异常

基本语法：

```
try:  
    可能发生错误的代码  
except:  
    如果出现异常执行的代码
```

示例：

```
try:  
    f = open('linux.txt', 'r')  
except:  
    f = open('linux.txt', 'w')
```

## 捕获制定异常

基本语法：

```
try:  
    print(name)  
except NameError as e:  
    print('name变量名称未定义错误')  
#只捕获命名异常这一类异常，给命名异常指定别名e  
① 如果尝试执行的代码的异常类型和要捕获的异常类型不一致，则无法捕获异常。  
② 一般try下方只放一行尝试执行的代码。
```

## 捕获多个异常

当捕获多个异常时，可以把要捕获的异常类型的名字，放到except 后，并使用元组的方式进行书写。

```
try:  
    print(1/0)  
except (NameError, ZeroDivisionError):  
    print('ZeroDivision错误...')
```

## 捕获异常并输出描述信息

```
try:  
    print(num)  
except (NameError, ZeroDivisionError) as e:  
    print(e)
```

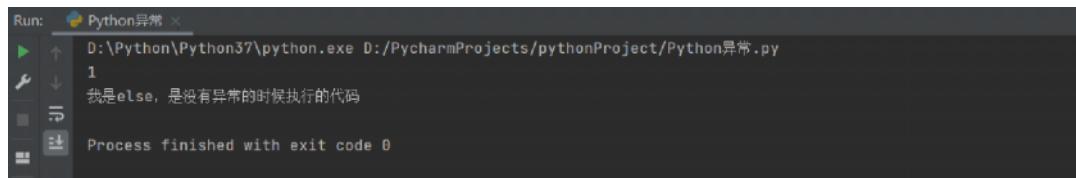
## 捕获所有异常

```
try:  
    print(num)  
except Exception as e:  
    print(e)
```

## 异常else(可选)

else表示的是如果没有异常要执行的代码。

```
try:  
    print(1)  
except Exception as e:  
    print(e)  
else:  
    print('我是else， 是没有异常的时候执行的代码')
```



```
Run: Python异常 ×  
D:\Python\Python37\python.exe D:/PycharmProjects/pythonProject/Python异常.py  
1  
我是else， 是没有异常的时候执行的代码  
Process finished with exit code 0
```

## 异常finally

finally表示的是无论是否异常都要执行的代码，例如关闭文件。

```
try:  
    f = open('test.txt', 'r')  
except Exception as e:  
    f = open('test.txt', 'w')  
else:  
    print('没有异常， 真开心')  
finally:  
    f.close()
```

# 异常的传递

2024年2月16日 17:58

```
def func01(): 异常在func01中没有被捕获
    print("这是func01开始")
    num = 1 / 0
    print("这是func01结束")

    异常在func02中没有被捕获
def func02():
    print("这是func02开始")
    func01() ←
    print("这是func02结束")

def main(): 异常在mian中被捕获
    try:
        func02() ←
    except Exception as e:
        print(e)

main()
```

利用异常具有传递性的特点, 当我们想要保证程序不会因为异常崩溃的时候, 就可以在main函数中设置异常捕获, 由于无论在整个程序哪里发生异常, 最终都会传递到main函数中, 这样就可以确保所有的异常都会被捕获

直接在最顶级的层级捕获:

```
def main():
    try:
        func2()
    except Exception as e:
        print(f"出现异常了, 异常的信息是: {e}")
```

# 模块

2024年2月16日 19:16

Python 模块(Module)，是一个 Python 文件，以 .py 结尾. 模块能定义函数，类和变量，模块里也能包含可执行的代码。

模块就是一个Python文件，里面有类、函数、变量等，我们可以拿过来用（导入模块去使用）。

## 语法

**[from 模块名] import [模块 | 类 | 变量 | 函数 | \*] [as 别名]**

import 模块名

from 模块名 import 类、变量、方法等

from 模块名 import \*

import 模块名 as 别名

from 模块名 import 功能名 as 别名

💡 **from可以省略，直接import即可**

💡 **as别名可以省略**

💡 **通过“.”来确定层级关系**

💡 **模块的导入一般写在代码文件的开头位置**

- **import 模块名**

import 模块名1, 模块名2

模块名.功能名()

例子：导入time模块

(按住ctrl点击time，可以看到time的源代码，ctrl+f搜索内容)

# 导入时间模块

import time

print("开始")

# 使用time里的sleep功能，让程序睡眠1秒(阻塞)

time.sleep(1)

print("结束")

- **from 模块名 import 功能名**

功能名()

案例：导入time模块中的sleep方法

```
# 导入时间模块中的sleep方法
from time import sleep

print("开始")
# 让程序睡眠1秒(阻塞)
sleep(1)

print("结束")
```

[**from 模块名 import [模块 | 类 | 变量 | 函数 | \*] [as 别名]**]

\*是全部功能的意思，和import time一样

- **from 模块名 import \***

```
from 模块名 import *
```

**功能名()**

# 使用 \* 导入time模块的全部功能

- ```
from time import *      # *表示全部的意思
print("你好")
sleep(5)
print("我好")
```

- **as 定义别名**

- **语法**

# 模块定义别名

```
import 模块名 as 别名
```

# 功能定义别名

```
from 模块名 import 功能 as 别名
```

**案例：**

# 模块别名

```
import time as tt
```

```
tt.sleep(2)
```

```
print('hello')
```

# 功能别名

```
from time import sleep as sl
```

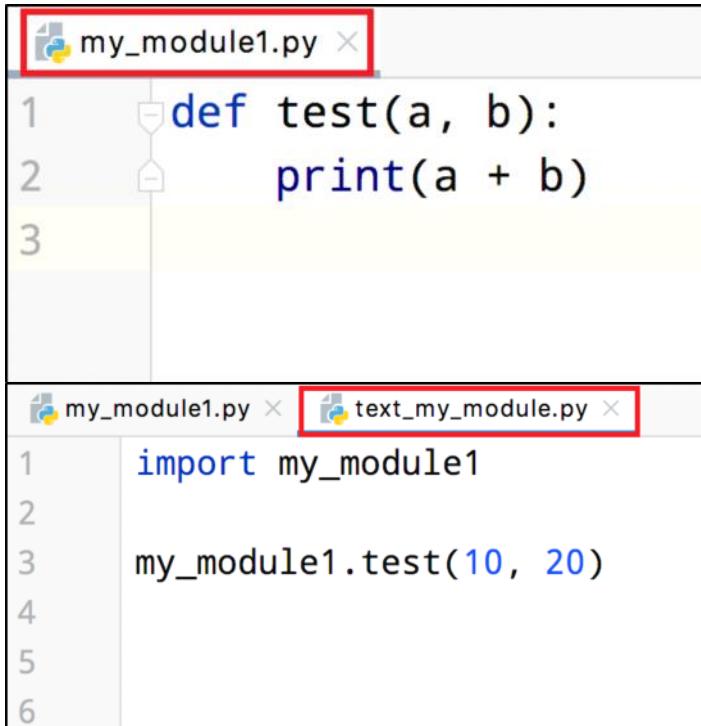
```
sl(2)
```

```
print('hello')
```

# 自定义模块

2024年2月16日 19:36

案例：新建一个Python文件，命名为my\_module1.py，并定义test函数



```
my_module1.py
1 def test(a, b):
2     print(a + b)
3

text_my_module.py
1 import my_module1
2
3 my_module1.test(10, 20)
4
5
6
```

注意事项：当导入多个模块的时候，且模块内有同名功能。当调用这个同名功能的时候，调用到的是后面导入的模块的功能

```
# 模块1代码
def my_test(a, b):
    print(a + b)

# 模块2代码
def my_test(a, b):
    print(a - b)

# 导入模块和调用功能代码
from my_module1 import my_test
from my_module2 import my_test

# my_test函数是模块2中的函数
my_test(1, 1)
```

## 测试模块：

在实际开发中，当一个开发人员编写完一个模块后，为了让模块能够在项目中达到想要的效果，

这个开发人员会自行在py文件中添加一些测试信息，例如，在my\_module1.py文件中添加测试代码test(1,1)

```
def test(a, b):  
    print(a + b)  
  
test(1, 1)
```

问题：

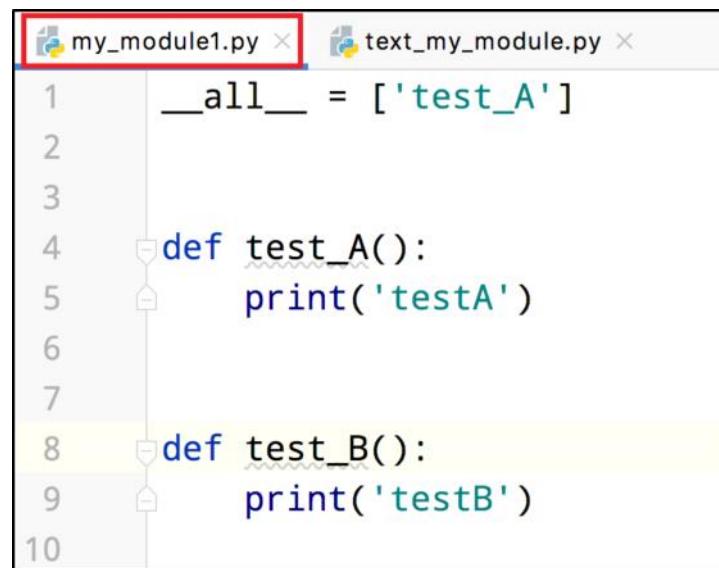
此时，无论是当前文件，还是其他已经导入了该模块的文件，在运行的时候都会自动执行`test`函数的调用

解决方案：

```
def test(a, b):  
    print(a + b)  
  
# 只在当前文件中调用该函数，其他导入的文件内不符合该条件，则不执行test函数调用  
if __name__ == '__main__':  
    test(1, 1)  
(写main回车就出来这个句子，就可以带数值测试了，但是导入文件的时候就什么事情就没有发生)
```

## \_\_all\_\_

如果一个模块文件中有`\_\_all\_\_`变量，当使用`from xxx import \*`导入时，只能导入这个列表中的元素



```
my_module1.py × text_my_module.py ×  
1  __all__ = ['test_A']  
2  
3  
4  def test_A():  
5      print('testA')  
6  
7  
8  def test_B():  
9      print('testB')  
10
```



```
my_module1.py x text_my_module.py x
1 from my_module1 import *
2
3 test|          这里只能使用test_A函数
4 f test_A()
5 ^↓ and ^↑ will move caret down and up in the editor Next Tip
6
7
```

```
# __all__ 变量
from my_module1 import test_b
test_a(1, 2)
test_b(2, 1)
```

但是手动主动还是可以的

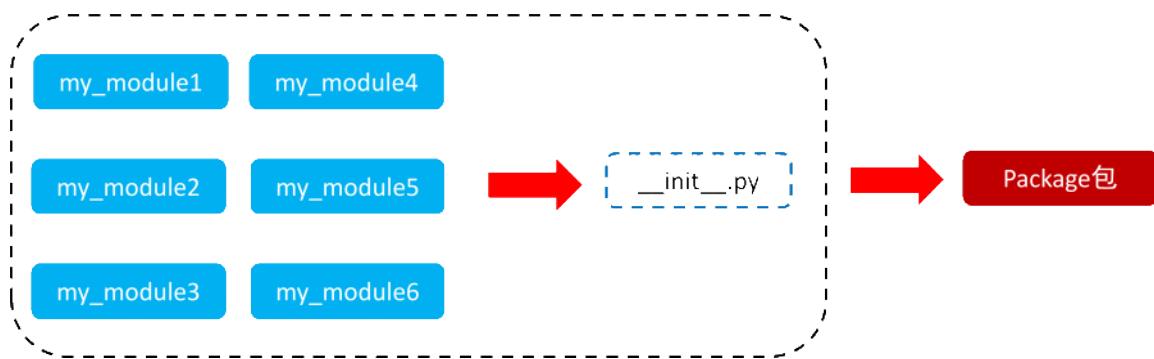
# 自定义python包

2024年2月16日 20:24

## 什么是Python包

从物理上看，包就是一个文件夹，在该文件夹下包含了一个 `__init__.py` 文件，该文件夹可用于包含多个模块文件

从逻辑上看，包的本质依然是模块

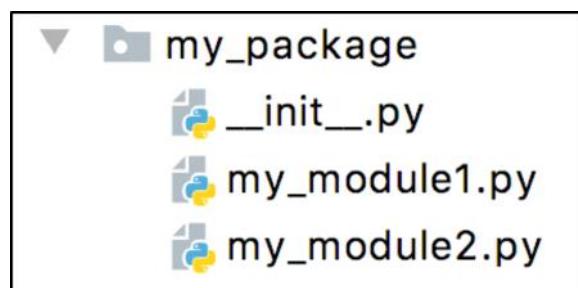


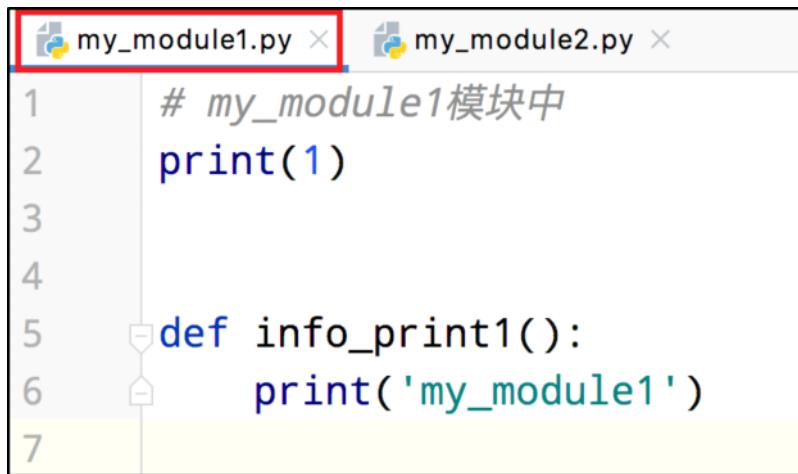
## 包的作用：

当我们的模块文件越来越多时，包可以帮助我们管理这些模块，包的作用就是包含多个模块，但包的本质依然是模块

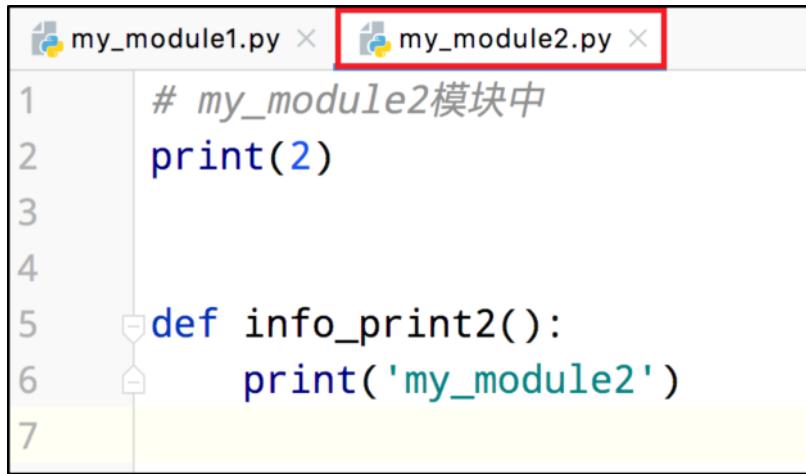
## 步骤如下：

- ① 新建包`my\_package`
- ② 新建包内模块：`my\_module1` 和 `my\_module2`
- ③ 模块内代码如下





```
1 # my_module1模块中
2 print(1)
3
4
5 def info_print1():
6     print('my_module1')
7
```

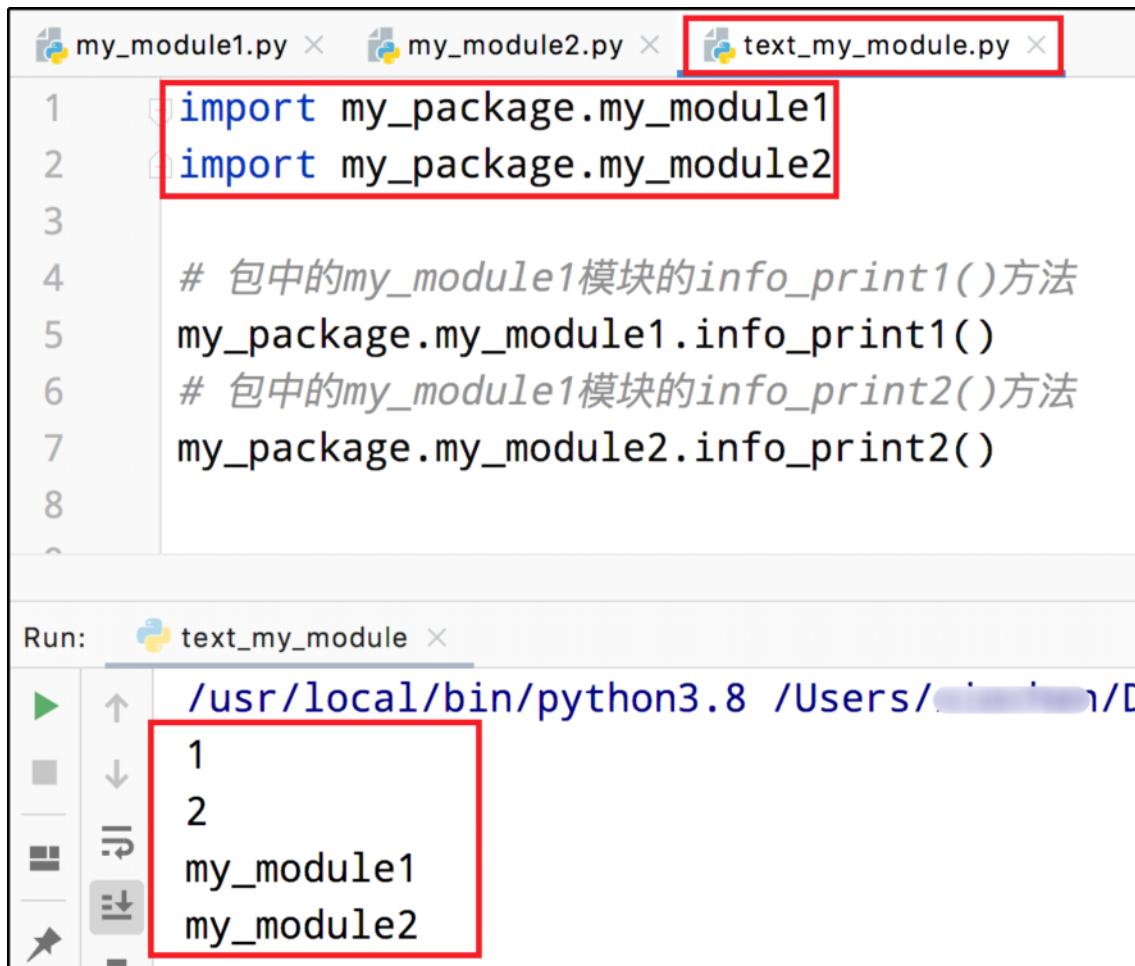


```
1 # my_module2模块中
2 print(2)
3
4
5 def info_print2():
6     print('my_module2')
7
```

## 导入包方式一

import 包名.模块名

包名.模块名.目标



```
my_module1.py x my_module2.py x text_my_module.py x
1 import my_package.my_module1
2 import my_package.my_module2
3
4 # 包中的my_module1模块的info_print1()方法
5 my_package.my_module1.info_print1()
6 # 包中的my_module1模块的info_print2()方法
7 my_package.my_module2.info_print2()

Run: text_my_module
/usr/local/bin/python3.8 /Users/.../D...
1
2
my_module1
my_module2
```

## 导入包方式二

from 包名 import 模块名

```
from my_package import my_module1
from my_package import my_module2
my_module1.info_print1()
my_module2.info_print2()
```

```
from my_package.my_module1 import info_print1
from my_package.my_module2 import info_print2
info_print1()
info_print2()
```

注意在`\_\_init\_\_.py`文件中添加`\_\_all\_\_ = []`，可以控制导入的模块列表



```
my_module1.py × my_module2.py × text_my_module.py × __init__.py ×
1 # 包中的__all__和模块中的__all__一样有着控制的功能
2 __all__ = ["my_module2"]
3 |
```

包中可以用的模块的名字



```
my_module1.py × my_module2.py × text_my_module.py × __init__.py ×
1 from my_package import *
2
3 # 包中的my_module1模块的info_print1()方法
4 my_module1.info_print1()
5 # 包中的my_module1模块的info_print2()方法
6 my_module2.info_print2()
```

注意:

\_\_all\_\_针对的是'from ... import \*'这种方式

对'import xxx'这种方式无效

# 安装第三方包

2024年2月16日 21:20

在Python程序的生态中，有许多非常多的第三方包（非Python官方），可以极大的帮助我们提高开发效率，如：

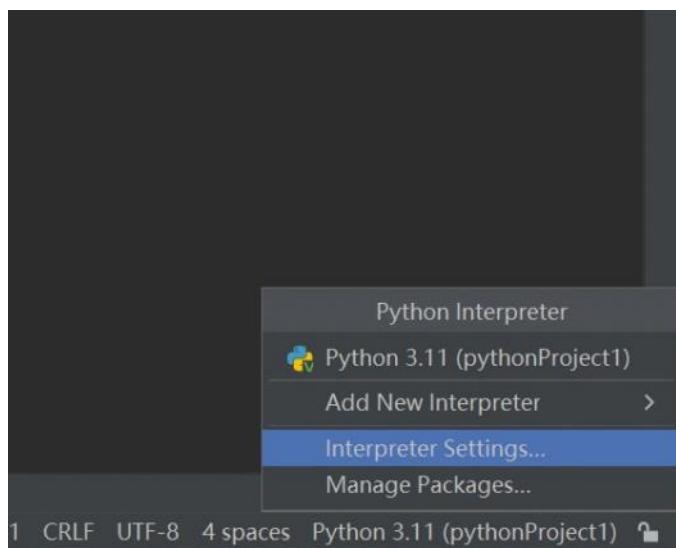
科学计算中常用的：numpy包

数据分析中常用的：pandas包

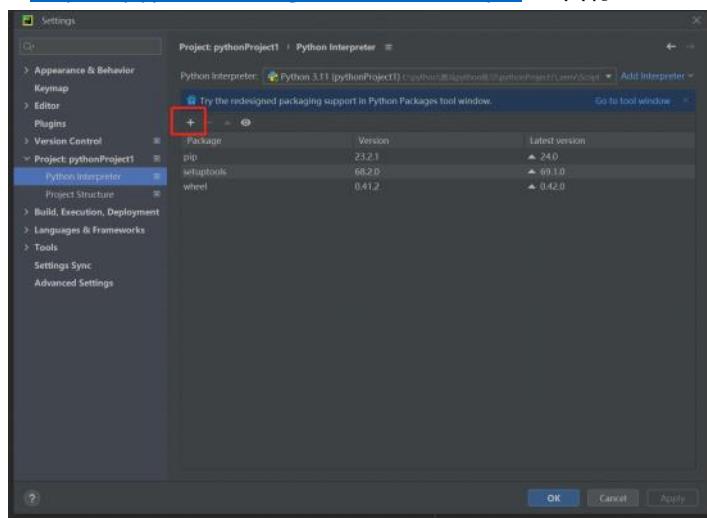
大数据计算中常用的：pyspark、apache-flink包

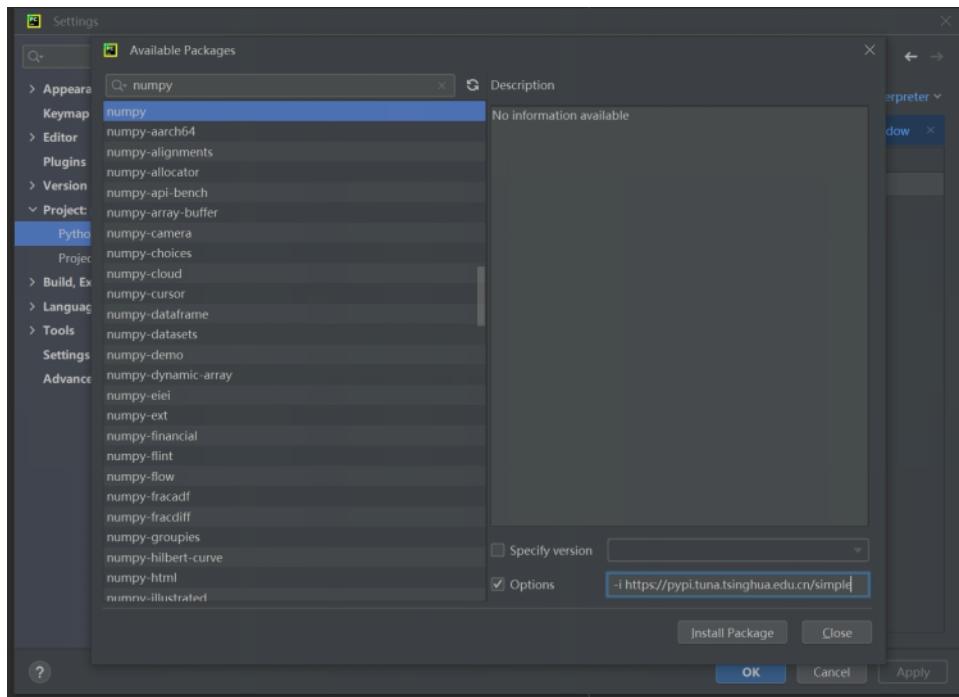
图形可视化常用的：matplotlib、pyecharts

人工智能常用的：tensorflow等



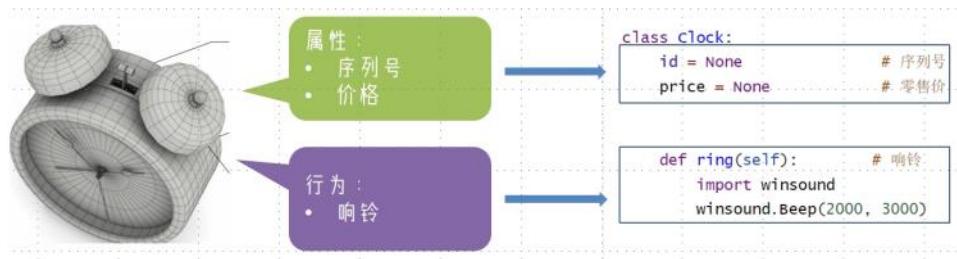
-i <https://pypi.tuna.tsinghua.edu.cn/simple> 包名称





# 面向对象编程

2024年2月18日 11:25



基于类创建对象



#1. 设计一个类（设计一张表）

```
class Student:  
    name=None  
    gender=None  
    nationality=None  
    native_place=None  
    age=None  
  
#2. 创建一个对象  
stu_1=Student() #stu_1是实例Instance, 获得Student的类别Class  
  
#3. 对象属性进行赋值  
stu_1.name="林俊杰"  
stu_1.gender="男"  
stu_1.nationality="中国"  
stu_1.native_place="山西省"  
stu_1.age=31  
  
#4. 输出  
print(stu_1.name)  
print(stu_1.gender)  
print(stu_1.nationality)  
print(stu_1.native_place)  
print(stu_1.age)
```

# 类和成员方法

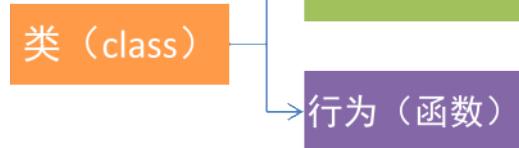
2024年2月18日 12:17

## 类的定义和使用

|            |                      |
|------------|----------------------|
| class 类名称: | class是关键字，表示要定义类了    |
| 类的属性       | 类的属性，即定义在类中的变量（成员变量） |
| 类的行为       | 类的行为，即定义在类中的函数（成员方法） |

创建类对象的语法：

对象 = 类名称()



在Python中，定义类是通过class关键字：

```
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，继承的概念我们后面再讲，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。

```
class student:  
    name = None      # 学生的姓名  
    age = None       # 学生的年龄  
  
    def say_hi(self):  
        print(f"Hi大家好，我是{self.name}")
```

类的属性

类的函数

那么，什么是类的行为（方法）呢？

```
class student:  
    name = None      # 学生的姓名  
    age = None       # 学生的年龄  
  
    def say_hi(self):  
        print(f"Hi大家好，我是{self.name}")
```

```
        stu = Student()  
        stu.name = "周杰伦"  
        stu.say_hi()      # 输出: Hi大家好，我是周杰伦
```

类中定义的属性（变量），我们称之为：成员变量  
类中定义的行为（函数），我们称之为：成员方法

成员方法的定义语法：

在类中定义成员方法和定义函数基本一致，但仍有细微区别：

```
def 方法名(self, 形参1, ..., 形参N):
```

    方法体

**在方法定义的参数列表中，有一个：self关键字**

**self关键字是成员方法定义的时候，必须填写的。**

- 它用来表示类对象自身的意思
- 当我们使用类对象调用方法的是，self会自动被python传入
- **在方法内部，想要访问类的成员变量，必须使用self**

self关键字，尽管在参数列表中，但是传参的时候可以忽略它。

```
class Student:  
    name = None  
  
    def say_hi(self):  
        print("Hello 大家好")  
  
    def say_hi2(self, msg):  
        print(f"Hello 大家好, {msg}")  
  
stu = Student()  
stu.say_hi() # 调用的时候无需传参  
stu.say_hi2("很高兴认识大家") # 调用的时候，需要传msg参数
```

可以看到，在传入参数的时候，self是透明的，可以不用理会它。

举例：

```
class Student:
```

name = None # 成员变量

def say\_hi(self):

print(f"hihi{self.name}") # 在方法内部，想要访问类的成员变量，必须

使用self

def say\_hi2(self, msg):

print(f"hihi{self.name},{msg}") # 在方法内部，想要访问类的成员变

量，必须使用self

```
stu = Student()
```

stu.name = "周杰伦"

stu.say\_hi()

stu.say\_hi2("啊啊")

```
stu2 = Student()
```

stu2.name = "林俊杰"

stu2.say\_hi()

stu2.say\_hi2("嗷嗷")

# 成员变量赋值

2024年2月18日 16:19

## 构造方法

Python类可以使用: `__init__()`方法, 称之为构造方法。

可以实现:

在创建类对象 (构造类) 的时候, 会自动执行。

在创建类对象 (构造类) 的时候, 将传入参数自动传递给`__init__`方法使用。

```
class Student:  
    name = None  
    age = None    可以省略  
    tel = None  
  
    def __init__(self, name, age, tel):  
        self.name = name  
        self.age = age  
        self.tel = tel  
        print("Student类创建了一个对象")  
  
stu = Student("周杰轮", 31, "18500006666")
```

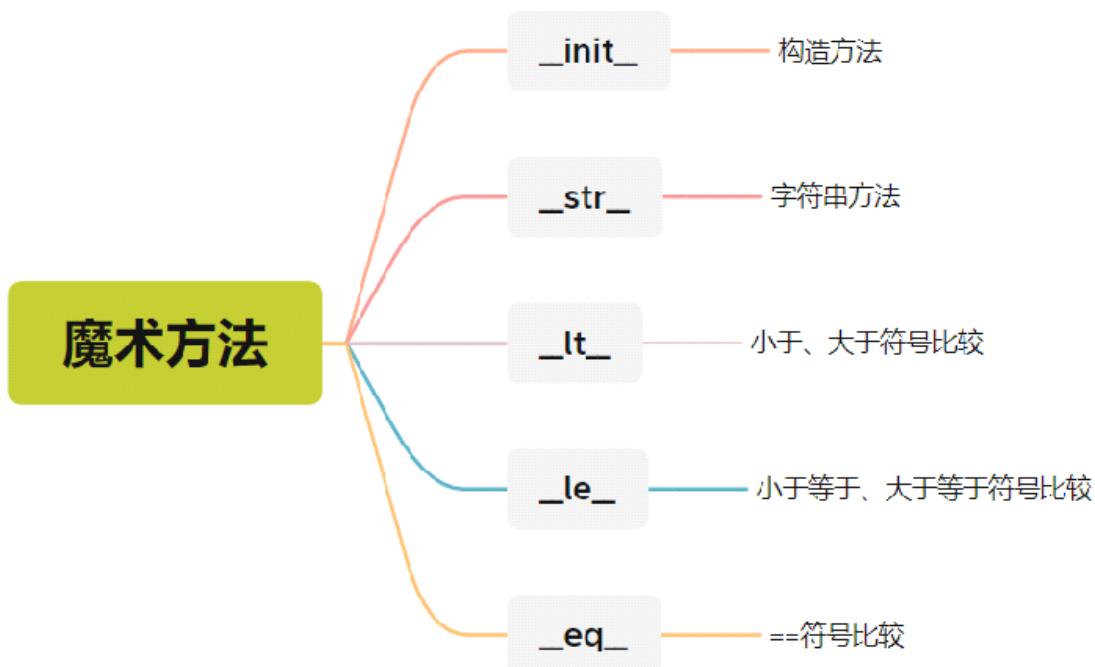
- 构建类时传入的参数会自动提供给`__init__`方法
- 构建类的时候`__init__`方法会自动执行

# 内置方法：init、str

2024年2月18日 19:09

上文学习的\_\_init\_\_ 构造方法，是Python类内置的方法之一。

这些内置的类方法，各自有各自特殊的功能，这些内置方法我们称之为：魔术方法



`__str__` 字符串方法return的一定是一个字符串,而不是一个元组

```
class Student:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
student = Student("周杰轮", 11)  
print(student)      # 结果: <__main__.Student object at 0x000002200CFD7040>  
print(str(student)) # 结果: <__main__.Student object at 0x000002200CFD7040>
```

当类对象需要被转换为字符串之时，会输出如上结果（内存地址）

内存地址没有多大作用，我们可以通过`__str__`方法，控制类转换为字符串的行为。

```
class Student:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def __str__(self):  
        return f"student类对象, name={self.name}, age={self.age}"
```

```
student = Student("周杰轮", 11)  
print(student)      # 结果: student类对象, name=周杰轮, age=11  
print(str(student)) # 结果: student类对象, name=周杰轮, age=11
```

也可以写成：

```

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))

```

\_\_lt\_\_ 小于符号比较方法

```

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

```

```
stu1 = Student("周杰轮", 11)
```

```
stu2 = Student("林军杰", 13)
```

```
print(stu1 < stu2)
```

```

Traceback (most recent call last):
  File "D:\python-learn\test.py", line 11, in <module>
    print(stu1 < stu2)
TypeError: '<' not supported between instances of 'Student' and 'Student'

```

直接对2个对象进行比较是不可以的，但是在类中实现\_\_lt\_\_方法，即可同时完成：小于符号 和 大于符号 2种比较

```

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

    def __lt__(self, other):
        return self.age < other.age

stu1 = Student("周杰轮", 11)
stu2 = Student("林军杰", 13)
print(stu1 < stu2) # 结果: True
print(stu1 > stu2) # 结果: False

```

- 方法名：\_\_lt\_\_
- 传入参数：other，另一个类对象
- 返回值：True 或 False
- 内容：自行定义

比较大于符号的魔术方法是：\_\_gt\_\_

不过，实现了 \_\_lt\_\_，\_\_gt\_\_ 就没必要实现了

## \_\_le\_\_ 小于等于比较符号方法

魔术方法: \_\_le\_\_ 可用于: <=、>=两种比较运算符上

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

    def __le__(self, other):
        return self.age <= other.age

stu1 = Student("周杰轮", 11)
stu2 = Student("林军杰", 13)
print(stu1 <= stu2) # 结果: True
print(stu1 >= stu2) # 结果: False
```

- 方法名: \_\_le\_\_
- 传入参数: other, 另一个类对象
- 返回值: True 或 False
- 内容: 自行定义

>=符号实现的魔术方法是: \_\_ge\_\_  
不过, 实现了 \_\_le\_\_, \_\_ge\_\_ 就没必要实现了

## \_\_eq\_\_, 比较运算符实现方法

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

    def __eq__(self, other):
        return self.age == other.age

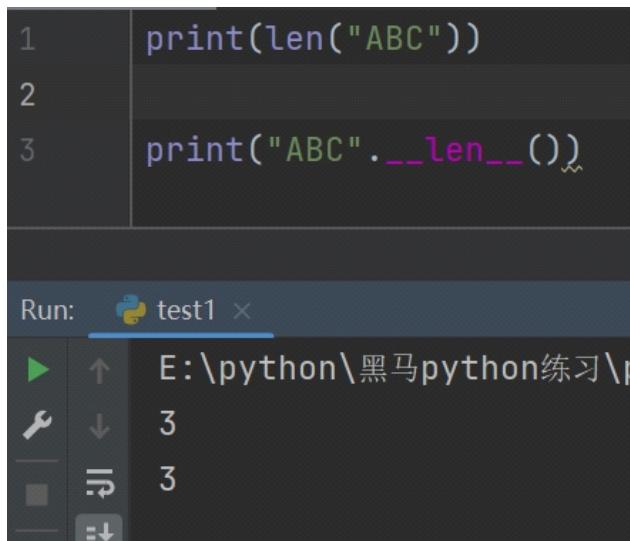
stu1 = Student("周杰轮", 11)
stu2 = student("林军杰", 11)
print(stu1 == stu2) # 结果: True
```

- 方法名: \_\_eq\_\_
- 传入参数: other, 另一个类对象
- 返回值: True 或 False
- 内容: 自行定义

不实现 \_\_eq\_\_ 方法, 对象之间可以比较, 但是是比较内存地址, 也即是: 不同对象==比较一定是 False 结果。

实现了 \_\_eq\_\_ 方法, 就可以按照自己的想法来决定2个对象是否相等了。

类似`__xxx__`的属性和方法在Python中都是有特殊用途的，比如`__len__`方法返回长度。在Python中，如果你调用`len()`函数试图获取一个对象的长度，实际上，在`len()`函数内部，它自动去调用该对象的`__len__()`方法，所以，下面的代码是等价的：



The screenshot shows a code editor with two snippets of Python code:

```
1 print(len("ABC"))
2
3 print("ABC).__len__()")
```

Below the code editor is a terminal window titled "Run: test1". The terminal shows the following output:

```
▶ 3
▶ 3
```

The terminal window has a dark theme with light-colored text. The "Run" tab is selected. The output shows two "3" characters, indicating that both code snippets produce the same result (the length of the string "ABC", which is 3).

# 封装：私有成员私有方法

2024年2月18日 19:33

面向对象编程，是许多编程语言都支持的一种编程思想。

简单理解是：基于模板（类）去创建实体（对象），使用对象完成功能开发。

面向对象包含3大主要特性：

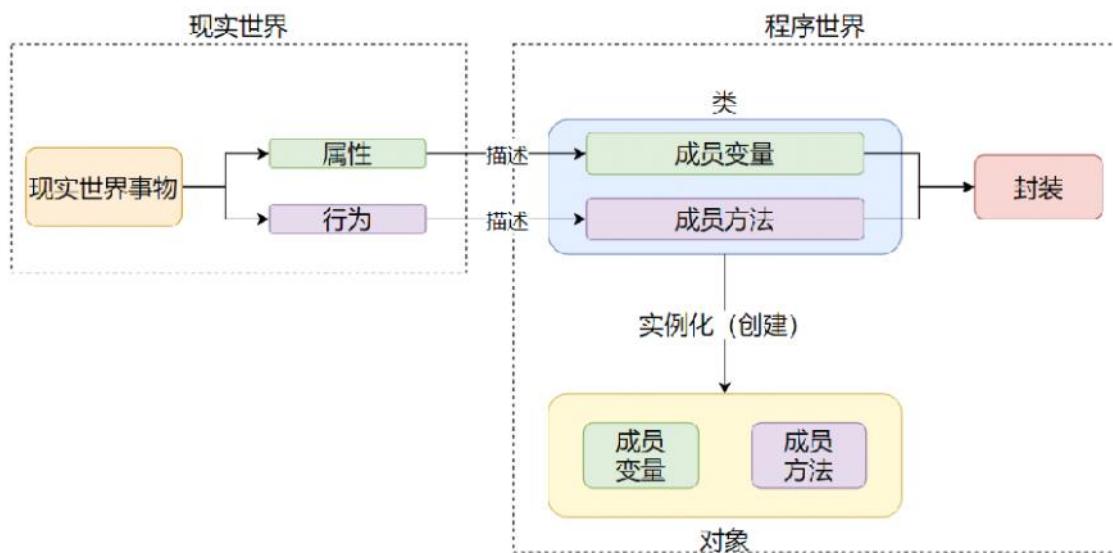
- 封装
- 继承
- 多态

封装表示的是，将现实世界事物的：

- 属性
- 行为

封装到类中，描述为：

- 成员变量
- 成员方法



## 私有成员

既然现实事物有不公开的属性和行为，那么作为现实事物在程序中映射的类，也应该支持。

类中提供了私有成员的形式来支持。

- 私有成员变量
- 私有成员方法

定义私有成员的方式非常简单，只需要：

- 私有成员变量：变量名以\_\_开头（2个下划线）
- 私有成员方法：方法名以\_\_开头（2个下划线）

即可完成私有成员的设置

```
class Phone:  
    IMEI = None # 序列号  
    producer = None # 厂商  
  
    __current_voltage = None # 当前电压  
    私有成员变量  
  
    def call_by_5g(self):  
        print("5g通话已开启")  
  
    def __keep_single_core(self):  
        print("让CPU以单核模式运行以节省电量")  
    私有成员方法
```

## 使用私有成员

### 私有方法无法直接被类对象使用

```
class Phone:  
    IMEI = None # 序列号  
    producer = None # 厂商  
  
    __current_voltage = None # 当前电压  
  
    def call_by_5g(self):  
        print("5g通话已开启")  
  
    def __keep_single_core(self):  
        print("让CPU以单核模式运行以节省电量")
```

```
phone = Phone() # 创建对象  
phone.__keep_single_core() # 使用私有方法
```

```
Traceback (most recent call last):  
  File "D:\python-learn\test.py", line 15, in <module>  
    phone.__keep_single_core() # 使用私有方法  
AttributeError: 'Phone' object has no attribute '__keep_single_core'  
私有变量无法赋值，也无法获取值
```

```

class Phone:
    IMEI = None          # 序列号
    producer = None      # 厂商

    __current_voltage = None  # 当前电压

    def call_by_5g(self):
        print("5g通话已开启")

    def __keep_single_core(self):
        print("让CPU以单核模式运行以节省电量")

phone = Phone()          # 创建对象
phone.__current_voltage = 33  # 私有变量赋值 不报错, 但无效
print(phone.__current_voltage) # 获取私有变量值 报错, 无法使用

```

私有成员无法被类对象使用, 但是可以被其它的成员使用。

```

class Phone:
    IMEI = None          # 序列号
    producer = None      # 厂商

    __current_voltage = None  # 当前电压

    def call_by_5g(self):
        if self.__current_voltage >= 1:
            self.__keep_single_core()  # 在成员方法内
            print("5g通话已开启")
        else:
            print("通话失败, 电量不足")

    def __keep_single_core(self):
        print("让CPU以单核模式运行以节省电量")

```

再如:

```

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))

```

改完后, 对于外部代码来说, 没什么变动, 但是已经无法从外部访问实例变量.\_\_name和实例变量.\_\_score了, 但是如果外部代码要获取name和score怎么办? 可以给Student类增加get\_name和get\_score这样的方法:

```

class Student(object):
    ...

```

```
def get_name(self):  
    return self.__name  
  
def get_score(self):  
    return self.__score
```

如果又要允许外部代码修改score怎么办？可以再给Student类增加set\_score方法：

```
def edit_score(self,score):  
    self.__score = score  
    return self.__score
```

# 继承父类

2024年2月19日 9:21

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为**子类（Subclass）**，而被继承的class称为**基类、父类或超类（Base class、Super class）**。

## 单继承

```
class Phone:  
    IMEI = None      # 序列号  
    producer = None # 厂商  
  
    def call_by_4g(self):  
        print("4g通话")
```

```
class Phone2022(Phone):  
    face_id = True    # 面部识别  
  
    def call_by_5g(self):  
        print("2022最新5g通话")
```

```
class 类名(父类名):  
    类内容体
```

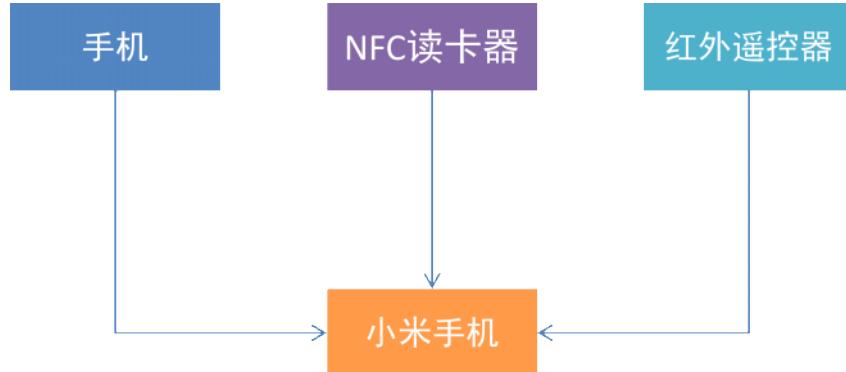
继承分为：单继承和多继承

使用如图语法，可以完成类的单继承。

继承表示：将从父类那里继承（复制）来成员变量和成员方法（不含私有）

## 多继承

Python的类之间也支持多继承，即一个类，可以继承多个父类



```
class 类名(父类1, 父类2, ..., 父类N):  
    类内容体
```

```
class Phone:
    IMEI = None          # 序列号
    producer = None      # 厂商

    def call_by_5g(self):
        print("5g通话")

class NFCReader:
    nfc_type = "第五代"
    producer = "HM"

    def read_card(self):
        print("读取NFC卡")

    def write_card(self):
        print("写入NFC卡")

class RemoteControl:
    rc_type = "红外遥控"

    def control(self):
        print("红外遥控开启")

class MyPhone(Phone, NFCReader, RemoteControl):
    pass
```

## 多继承注意事项

多个父类中，如果有同名的成员，那么默认以继承顺序（从左到右）为优先级。  
即：先继承的保留，后继承的被覆盖

```
class Phone:
    IMEI = None          # 序列号
    producer = None      # 厂商

    def call_by_5g(self):
        print("5g通话")

class NFCReader:
    nfc_type = "第五代"
    producer = "HM"

    def read_card(self):
        print("读取NFC卡")

    def write_card(self):
        print("写入NFC卡")

class Remotecontrol:
    rc_type = "红外遥控"

    def control(self):
        print("红外遥控开启")

class MyPhone(Phone, NFCReader, Remotecontrol):
    pass

my_phone = MyPhone()
print(my_phone.producer)      # 结果为None而非"HM"
```

# 复写父类成员和调用父类成员

2024年2月19日 9:54

## 复写

子类继承父类的成员属性和成员方法后，如果对其“不满意”，那么可以进行复写。  
即：在子类中重新定义同名的属性或方法即可。

```
class Phone:
    IMEI = None          # 序列号
    producer = "ITCAST" # 厂商

    def call_by_5g(self):
        print("父类的5g通话")

class MyPhone(Phone):
    producer = "ITHEIMA"      # 复写父类属性

    def call_by_5g(self):      # 复写父类方法
        print("子类的5g通话")
```

## 调用父类同名成员

一旦复写父类成员，那么类对象调用成员的时候，就会调用复写后的新成员  
如果需要使用被复写的父类的成员，需要特殊的调用方式：

### 方式1：

- 调用父类成员
  - 使用成员变量：父类名.成员变量
  - 使用成员方法：父类名.成员方法(self)

### 方式2：

- 使用super()调用父类成员
  - 使用成员变量：super().成员变量
  - 使用成员方法：super().成员方法()

只能在子类内调用父类的同名成员。  
子类的类对象直接调用会调用子类复写的成员

```
class Phone:
    IMEI = None          # 序列号
    producer = "ITCAST" # 厂商

    def call_by_5g(self):
        print("父类的5g通话")

class MyPhone(Phone):
    producer = "ITHEIMA"

    def call_by_5g(self):
        # 方式1调用父类成员
        print(f"父类的品牌是: {Phone.producer}")
        Phone.call_by_5g(self)

        # 方式2调用父类成员
        print(f"父类的品牌是: {super().producer}")
        super().call_by_5g()

    print("子类的5g通话")
```

# 注解

2024年2月19日 17:31

**基础语法:**

**变量: 类型注解 = 赋值**

# 变量的类型注解

2024年2月19日 10:11

## 类型注解：

Python在3.5版本的时候引入了类型注解，以方便静态类型检查工具，IDE等第三方工具。

类型注解：在代码中涉及数据交互的地方，提供数据类型的注解（显式的说明）。

## 主要功能：

- 帮助第三方IDE工具（如PyCharm）对代码进行类型推断，协助做代码提示
- 帮助开发者自身对变量进行类型注释

## 支持：

- 变量的类型注解
- 函数（方法）形参列表和返回值的类型注解

## 类型注解的语法：

为变量设置类型注解

## 基础语法：

**变量: 类型注解 = 赋值**

基础数据类型注解

```
var_1: int = 10
var_2: float = 3.1415926
var_3: bool = True
var_4: str = "itheima"
```

类对象类型注解

```
class student:
    pass
stu: student = student()
```

## 基础容器类型注解

```
my_list: list = [1, 2, 3]
my_tuple: tuple = (1, 2, 3)
my_set: set = {1, 2, 3}
my_dict: dict = {"itheima": 666}
my_str: str = "itheima"
```

## 容器类型详细注解

```
my_list: list[int] = [1, 2, 3]
my_tuple: tuple[str, int, bool] = ("itheima", 666, True)
my_set: set[int] = {1, 2, 3}
my_dict: dict[str, int] = {"itheima": 666}
```

注意：

- 元组类型设置类型详细注解，需要将每一个元素都标记出来
- 字典类型设置类型详细注解，需要2个类型，第一个是key第二个是value

除了使用 变量: 类型，这种语法做注解外，也可以在注释中进行类型注解。

语法：

```
# type: 类型
```

## 在注释中进行类型注解

```
class Student:
    pass

var_1 = random.randint(1, 10)    # type: int
var_2 = json.loads(data)        # type: dict[str, int]
var_3 = func()                 # type: Student
```

类型注解的语法：

为变量设置注解，显示的变量定义，一般无需注解：

```
var_1: int = 10
var_2: list = [1, 2, 3]
var_3: dict = {"itheima": 666}
var_4: Student = Student()
```

如图，就算不写注解，也明确的知晓变量的类型

```
class Student:  
    pass  
  
var_1: int = random.randint(1, 10)  
var_2: dict = json.loads(data)  
var_3: Student = func()
```

一般，无法直接看出变量类型之时  
会添加变量的类型注解

### 类型注解的限制

类型注解主要功能在于：

- 帮助第三方IDE工具（如PyCharm）对代码进行类型推断，协助做代码提示
- 帮助开发者自身对变量进行类型注释（备注）

并不会真正的对类型做验证和判断。

也就是，类型注解仅仅是提示性的，不是决定性的

```
var_1: int = "itheima"
```

```
var_2: str = 123
```

如图代码，是不会报错的哦。

# 函数和方法类型注解

2024年2月19日 16:33

## 函数（方法）的类型注解 - 形参注解

```
def func(data):
```

```
    data.app|
```

data

```
func()
```

如图所示：

- 在编写函数（方法），使用形参data的时候，工具没有任何提示
  - 在调用函数（方法），传入参数的时候，工具无法提示参数类型
- 这些都是因为，我们在定义函数（方法）的时候，没有给形参进行注解

## 函数和方法的形参类型注解语法：

```
def 函数方法名(形参名: 类型, 形参名: 类型, .....):  
    pass
```

```
def add(x: int, y: int):  
    return x + y
```

x: int, y: int

```
add()
```

```
def func(data: list):  
    data.app|
```

append(self, \_\_object) list

Ctrl+向左箭头 and Ctrl+向上箭头 will move caret down and up in the editor Next Tip

调用函数的时候，按ctrl+p就会弹出提示

## 对返回值进行类型注解：

```
def 函数方法名(形参: 类型, ..... , 形参: 类型) -> 返回值类型:  
    pass
```

```
def add(x: int, y: int) -> int:  
    return x + y
```

```
def func(data: list[int]) -> list[int]:  
    pass
```

```
def func(data:list) -> list:  
    return data
```

# Union联合类型注解

2024年2月19日 16:45

只有一种类型注解：

```
my_list: list[int] = [1, 2, 3]
```

```
my_dict: dict[str, int] = {"age": 11, "num": 3}
```

使用Union[类型, ..., 类型]

可以定义联合类型注解

```
from typing import Union
```

```
my_list: list[Union[str, int]] = [1, 2, "itheima", "itcast"]
```

```
my_dict: dict[str, Union[str, int]] = {"name": "周杰轮", "age": 31}
```

Union联合类型注解，在变量注解、函数（方法）形参和返回值注解中，均可使用。

```
my_list: list[Union(int, str)] = [1, 2, "itcast", "itheima"]
```

```
my_dict: dict[str, Union[str, int]] = {"name": "周杰轮", "age": 31}
```

```
def func(data: Union[int, str]) -> Union[int, str]:  
    pass
```

# 多态

2024年2月19日 17:10

多态，指的是：多种状态，即完成某个行为时，使用不同的对象会得到不同的状态。

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        print("汪汪汪")

class Cat(Animal):
    def speak(self):
        print("喵喵喵")

def make_noise(animal: Animal):
    animal.speak()

dog = Dog()
cat = cat()

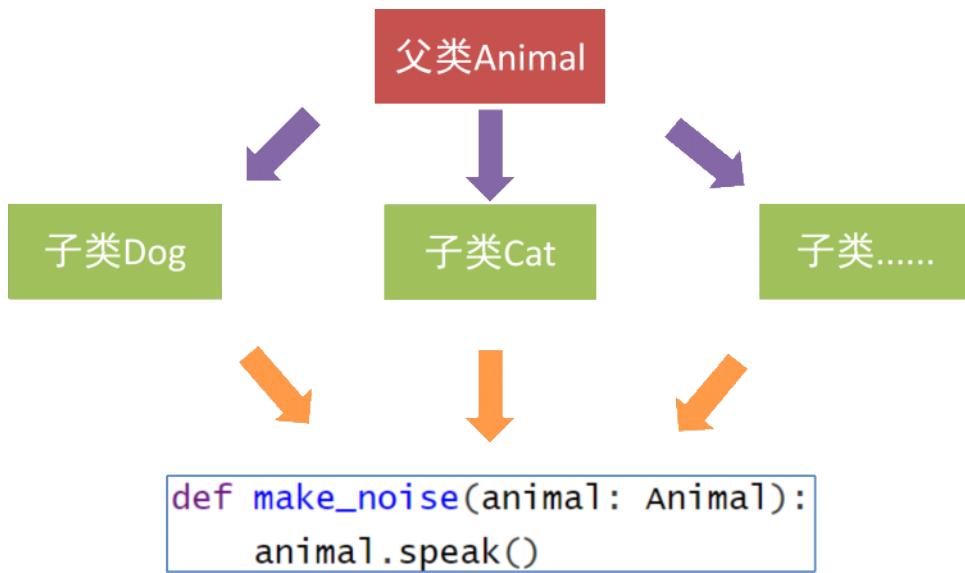
make_noise(dog)      # 输出: 汪汪汪
make_noise(cat)      # 输出: 喵喵喵
```

这个函数接受一个Animal类型的参数animal，并调用它的speak方法。

在make\_noise函数中，animal参数是一个Animal类的实例（或者是其子类的一个实例）。

当你调用animal.speak()时，你实际上是在调用该实例的speak方法。在这种情况下，animal就相当于类方法内部的self。

同样的行为（函数），传入不同的对象，得到不同的状态



多态常作用在继承关系上.

比如

- 函数(方法)形参声明接收父类对象
- 实际传入父类的子类对象进行工作

即:

- 以父类做定义声明
- 以子类做实际工作
- 用以获得同一行为, 不同状态

### 抽象类 (接口)

细心的同学可能发现了, 父类Animal的speak方法, 是空实现

```

class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        print("汪汪汪")

class Cat(Animal):
    def speak(self):
        print("喵喵喵")
  
```

这种设计的含义是:

- 父类用来确定有哪些方法
- 具体的方法实现，由子类自行决定

这种写法，就叫做抽象类（也可以称之为接口）

抽象类：含有抽象方法的类称之为抽象类

抽象方法：方法体是空实现的（pass）称之为抽象方法

```
class AC:  
    def cool_wind(self):  
        """制冷"""  
        pass  
  
    def hot_wind(self):  
        """制热"""  
        pass  
  
    def swing_l_r(self):  
        """左右摆风"""  
        pass
```

```
class Midea_AC(AC):  
    def cool_wind(self):  
        print("美的空调核心制冷科技")  
  
    def hot_wind(self):  
        print("美的空调电热丝加热")  
  
    def swing_l_r(self):  
        print("美的空调无风感左右摆风")
```

```
class GREE_AC(AC):
    def cool_wind(self):
        print("格力空调变频省电制冷")

    def hot_wind(self):
        print("格力空调电热丝加热")

    def swing_l_r(self):
        print("格力空调静音左右摆风")
```

配合多态，完成

- 抽象的父类设计（设计标准）
- 具体的子类实现（实现标准）

```
def make_cool(ac: AC):
    ac.cool_wind()
```

```
midea_ac = Midea_AC()
gree_ac = GREE_AC()
```

```
make_cool(midea_ac)      # 输出: 美的空调制冷
make_cool(gree_ac)       # 输出: 格力空调制冷
```

# 实例属性和类属性

2024年2月20日 11:14

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过self变量：

```
class Student(object):
    def __init__(self, name):
        self.name = name

s = Student('Bob')
s.score = 90
```

如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有：

```
class Student(object):
    name = 'Student'
```

当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下：

```
>>> 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
```

从上面的例子可以看出，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。

例题：

为了统计学生人数，可以给Student类增加一个类属性，每创建一个实例，该属性自动增加：

```
class Student(object):
    count = 0

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

```
Student.count += 1

#每当实例化一个对象, 类属性count就会自增
#注意区分类属性和实例属性, 这里如果改成self. count, 那就错了
```

# 限制实例属性

2024年2月20日 14:13

`__slots__`

只允许对Student实例添加name和age属性。

```
class Student(object):
    __slots__ = ('name', 'age')
```

```
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'
```

由于'score'没有被放到`__slots__`中，所以不能绑定score属性，试图绑定score将得到AttributeError的错误。

使用`__slots__`要注意，`__slots__`定义的属性仅对当前类实例起作用，对继承的子类是不起作用的：

```
class GraduateStudent(Student):
    ...     pass
    ...
>>> g = GraduateStudent()
>>> g.score = 9999
```

# 闭包

2024年2月20日 14:59

## 闭包

在函数嵌套的前提下，内部函数使用了外部函数的变量，并且外部函数返回了内部函数，我们把这个使用外部函数变量的内部函数称为闭包。

```
account_amount = 0      # 账户余额
def atm(num, deposit=True):
    global account_amount
    if deposit:
        account_amount += num
        print(f"存款:{+{num}}, 账户余额:{account_amount}")
    else:
        account_amount -= num
        print(f"取款:{-{num}}, 账户余额:{account_amount}")

atm(300)
atm(300)
atm(100, False)
```

简单闭包后：

```
def account_create(initial_amount=0):
    def atm(num, deposit=True):
        nonlocal initial_amount
        if deposit:
            initial_amount += num
            print(f"存款:{+{num}}, 账户余额:{initial_amount}")
        else:
            initial_amount -= num
            print(f"取款:{-{num}}, 账户余额:{initial_amount}")

    return atm

fn = account_create()
fn(300)
fn(200)
fn(-100)
```

```
fn(300)
fn(200)
fn(300, False)
D:\dev\Python\Python31
```

存款:+300, 账户余额:300

存款:+200, 账户余额:500

取款:-300, 账户余额:200

简单闭包



```
def outer(logo):  
    def inner(msg):  
        print(f"<{logo}>{msg}<{logo}>")  
    return inner
```

```
fn1 = outer("黑马程序员")  
fn1("hello")
```

test1 ×

E:\python\黑马python练习\pythonProject1\.ve

E:\python\黑马python练习\pythonProject1\te

<黑马程序员>hello<黑马程序员>

Process finished with exit code 0

思考：能不能在inner修改外部的logo？

可以的，用nonlocal就可以了

```
def outer(num1):  
    def inner(num2):  
        nonlocal num1  
        num1 += num2  
        print(num1)  
    return inner  
  
fn = outer(10)  
fn(10)  
fn(10)  
fn(10)  
fn(10)
```

```
test1 ×  
E:\python\黑马python练习\pythonProject1\.env  
E:\python\黑马python练习\pythonProject1\test1.py  
20  
30  
40  
50  
  
Process finished with exit code 0
```

这是因为 num1 在 inner 函数的作用域内是持续存在的，并且在每次调用 fn 时都会保持其状态。每次调用 fn 都会更新 num1 的值，并在下一次调用时保留这个更新的值。这就是为什么你看到的结果是累加的原因。

# 设计模式

2024年2月20日 19:45

# 单例模式

2024年2月20日 19:03

## 单例模式

```
class Tool:  
    pass  
  
t1 = Tool()  
t2 = Tool()  
print(t1)  
print(t2)
```

```
<__main__.Tool object at 0x0000027246FCCEE0>  
<__main__.Tool object at 0x0000027246FCD9F0>
```

创建类的实例后，就可以得到一个完整的、独立的类对象。

通过print语句可以看出，它们的内存地址是不相同的，即t1和t2是完全独立的两个对象。

**某些场景下，我们需要一个类无论获取多少次类对象，都仅仅提供一个具体的实例**

用以节省创建类对象的开销和内存开销

比如某些工具类，仅需要1个实例，即可在各处使用

**单例模式 (Singleton Pattern) 是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。**

在整个系统中，某个类只能出现一个实例时，单例对象就能派上用场。

定义：保证一个类只有一个实例，并提供一个访问它的全局访问点

适用场景：当一个类只能有一个实例，而客户可以从一个众所周知的访问点访问它时。

|                                                      |                                                                                           |                                                                                                                   |
|------------------------------------------------------|-------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| class StrTools:<br>pass<br><br>str_tool = StrTools() | from test import str_tool<br><br>s1 = str_tool<br>s2 = str_tool<br>print(s1)<br>print(s2) | <test.StrTools object at 0x0000013D0001DB910><br><test.StrTools object at 0x0000013D0001DB910><br><br>s1和s2是同一个对象 |
|------------------------------------------------------|-------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|

在一个文件中定义如上代码

在另一个文件中导入对象

s1和s2是同一个对象

# 工厂模式

2024年2月20日 19:14

当需要大量创建一个类的实例的时候， 可以使用工厂模式。

即，从原生的使用类的构造去创建对象的形式

迁移到，基于工厂提供的方法去创建对象的形式。

```
class Person:
```

```
    pass
```

```
class Worker(Person):
```

```
    pass
```

```
class Student(Person):
```

```
    pass
```

```
class Teacher(Person):
```

```
    pass
```

```
worker = Worker()
```

```
stu = Student()
```

```
teacher = Teacher()
```



使用工厂类的get\_person()方法去创建具体的类对象

```
class Person:  
    pass  
  
class Worker(Person):  
    pass  
class Student(Person):  
    pass  
class Teacher(Person):  
    pass  
  
class Factory:  
    def get_person(self, p_type):  
        if p_type == 'w':  
            return Worker()  
        elif p_type == 's':  
            return Student()  
        else:  
            return Teacher()  
  
factory = Factory()  
worker = factory.get_person('w')  
stu = factory.get_person('s')  
teacher = factory.get_person('t')
```

优点：

- 大批量创建对象的时候有统一的入口，易于代码维护
- 当发生修改，仅修改工厂类的创建方法即可
- 符合现实世界的模式，即由工厂来制作产品（对象）

## 多线程

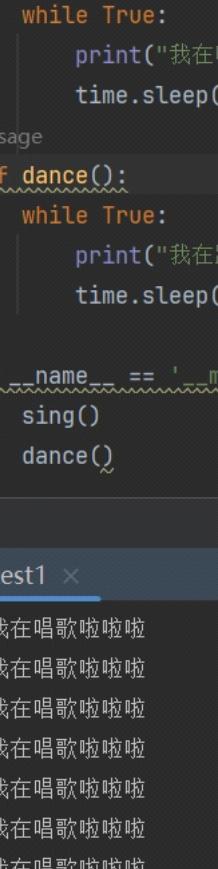
2024年2月20日 19:45

单线程：

```
import time
1 usage
def sing():
    while True:
        print("我在唱歌啦啦啦")
        time.sleep(1)

1 usage
def dance():
    while True:
        print("我在跳舞呱呱呱")
        time.sleep(1)

if __name__ == '__main__':
    sing()
    dance()


```

## 多线程：

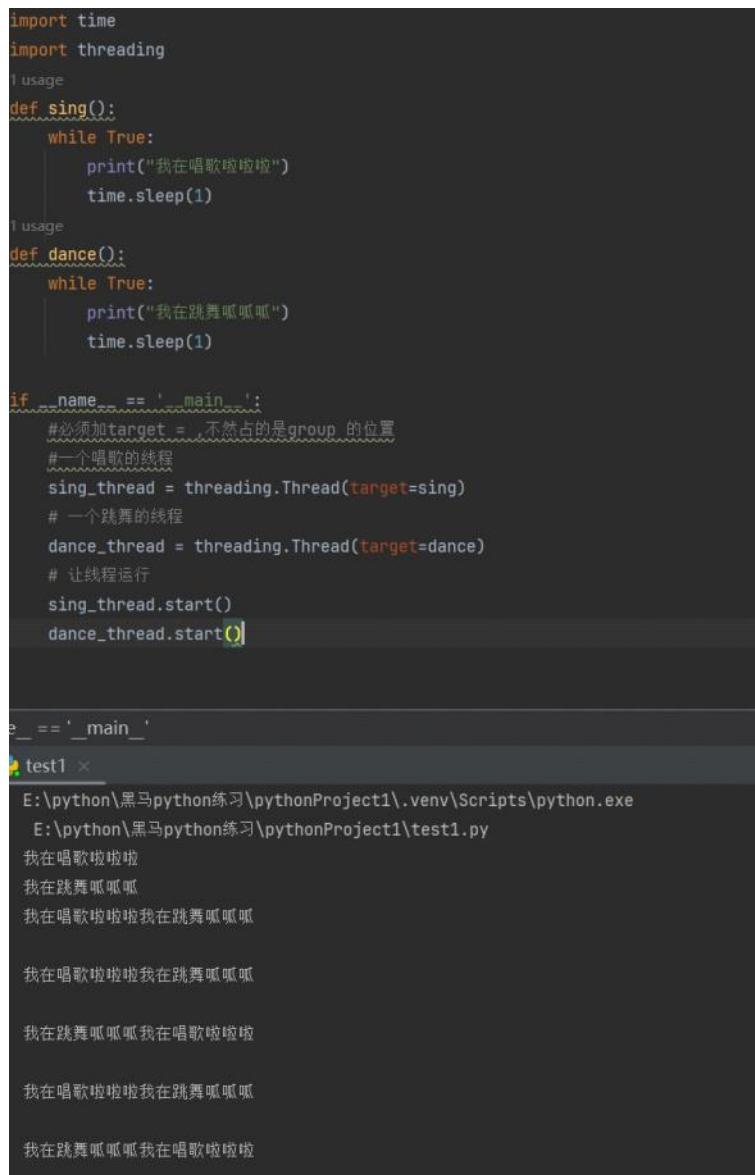
## threading模块

Python的多线程可以通过threading模块来实现。

```
import threading
```

```
thread_obj = threading.Thread([group [, target [, name [, args [, kwargs]]]]])  
- group: 暂时无用, 未来功能的预留参数  
- target: 执行的目标任务名  
- args: 以元组的方式给执行任务传参  
- kwargs: 以字典方式给执行任务传参  
- name: 线程名, 一般不用设置
```

```
# 启动线程，让线程开始工作
thread_obj.start()
如果需要等待线程完成其任务，可以调用线程对象的 join() 方法。
thread.join()
```



```
import time
import threading
# usage
def sing():
    while True:
        print("我在唱歌啦啦啦")
        time.sleep(1)
# usage
def dance():
    while True:
        print("我在跳舞呱呱呱")
        time.sleep(1)

if __name__ == '__main__':
    #必须加target =，不然占的是group 的位置
    #一个唱歌的线程
    sing_thread = threading.Thread(target=sing)
    # 一个跳舞的线程
    dance_thread = threading.Thread(target=dance)
    # 让线程运行
    sing_thread.start()
    dance_thread.start()
```

```
test1 ×
E:\python\黑马python练习\pythonProject1\.venv\Scripts\python.exe
E:\python\黑马python练习\pythonProject1\test1.py
我在唱歌啦啦啦
我在跳舞呱呱呱
我在唱歌啦啦啦我在跳舞呱呱呱

我在唱歌啦啦啦我在跳舞呱呱呱

我在跳舞呱呱呱我在唱歌啦啦啦

我在唱歌啦啦啦我在跳舞呱呱呱

我在跳舞呱呱呱我在唱歌啦啦啦
```

需要传参的话可以通过：  
args参数通过元组（按参数顺序）的方式传参  
或使用kwargs参数用字典的形式传参

```
import time
import threading
# usage
def sing(msg):
    while True:
        print(msg)
        time.sleep(1)

# usage
def dance(msg):
    while True:
        print(msg)
        time.sleep(1)

if __name__ == '__main__':
    #必须加target=，不然占的是group 的位置
    #元组必须加逗号，不然就是普通的括号
    #一个唱歌的线程
    sing_thread = threading.Thread(target=sing, args=("我要唱歌哈哈哈",))
    # 一个跳舞的线程
    dance_thread = threading.Thread(target=dance, kwargs={"msg": "我在跳舞 啦啦啦"})
    # 让线程运行
    sing_thread.start()
    dance_thread.start()

__name__ = '__main__'

test1 x
E:\python\黑马python练习\pythonProject1\.venv\Scripts\python.exe
E:\python\黑马python练习\pythonProject1\test1.py
我要唱歌哈哈哈
我在跳舞 啦啦啦
我要唱歌哈哈哈我在跳舞 啦啦啦

我在跳舞 啦啦啦我要唱歌哈哈哈

我要唱歌哈哈哈我在跳舞 啦啦啦

我在跳舞 啦啦啦我要唱歌哈哈哈

Process finished with exit code -1
```

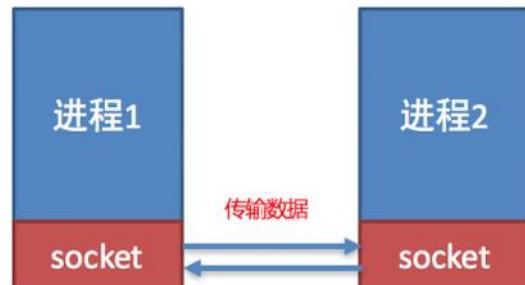
# 服务端开发

2024年2月21日 10:29

## Socket

socket (简称 套接字) 是进程之间通信一个工具，好比现实生活中的插座，所有的家用电器要想工作都是基于插座进行，进程之间想要进行网络通信需要socket。

Socket负责进程之间的网络数据传输，好比数据的搬运工。



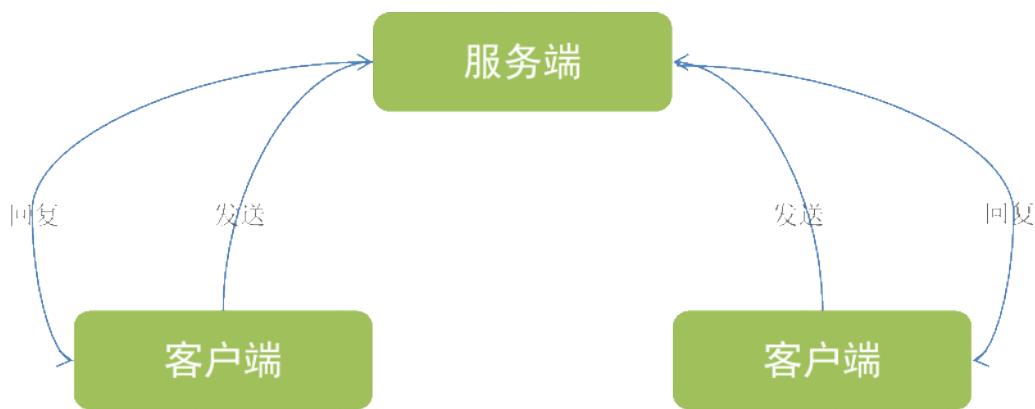
大多数软件都使用到了Socket进行网络通讯

## 客户端和服务端

2个进程之间通过Socket进行相互通讯，就必须有服务端和客户端

Socket服务端：等待其它进程的连接、可接受发来的消息、可以回复消息

Socket客户端：主动连接服务端、可以发送消息、可以接收到回复



## socket服务端编程

socket 模块是 Python 中用于实现网络编程的标准库之一。它提供了对低级网络通信的支持，允许开发者创建客户端和服务器应用程序，进行数据传输和通信。

socket 模块的主要功能包括：

1. 创建套接字 (Sockets)：套接字是网络通信的端点，socket 模块提供了创建

不同类型套接字的方法，如 TCP、UDP、原始套接字等。

2. 绑定地址和端口：通过 **bind()** 方法，可以将套接字绑定到特定的 IP 地址和端口上，从而指定网络通信的端点。
3. 监听和接受连接：对于服务器套接字，可以使用 **listen()** 方法开始监听传入的连接请求，并使用 **accept()** 方法接受客户端的连接请求。
4. 发送和接收数据：使用 **send()** 和 **recv()** 方法可以在客户端和服务器之间发送和接收数据。
5. 关闭套接字：在完成网络通信后，可以使用 **close()** 方法关闭套接字，释放相关资源。

主要分为如下几个步骤：

```
#导入socket包
import socket
#创建socket对象
socket_server=socket.socket()
#bind()方法需要一个元组作为参数,绑定ip地址和端口,加两层括号是因为元组
socket_server.bind(("localhost", 8888))
#监听端口,listen方法接受一个整数传参,表示接受的链接数量
socket_server.listen(1)
#等待客户端连接, accept返回结果是一个二元元组
conn, address = socket_server.accept()
#conn = result[0]客户端和服务端的链接对象
#address = result[1]客户端地址信息
#accpet()方法是阻塞的方法,等待客户端的链接,如果没有链接,就卡在这一行不向下执行了
print(f"接受到了客户端的链接, 客户端的信息是: {address}")
while True:
    #接受客户端信息, 要使用客户端和服务端的本次链接对象, 而非
    #socket_server对象
    #recv()方法用于从套接字接收数据。这个方法会尝试读取指定数量的字节,
    #并返回一个包含这些字节的字节串 (bytes)
    #1024指的是尝试从套接字中读取最多1024字节的数据。这个数字通常是一个经验值, 它代表了一个相对较大的数据块
    #但又不会太大以至于消耗过多的内存。
    #recv方法返回的是字节, decode()解码
    data = conn.recv(1024).decode("utf-8")
    print(f"客户端发来的信息是: {data}")
    #发送回复信息
    msg=input("请输入你和客户端回复的信息: ")
    if msg=="exit":
        break
    conn.send(msg.encode("utf-8"))
    #关闭链接
    conn.close()
    socket_server.close()
```



# 编码和解码

2024年2月21日 11:30

`decode()`解码：将字节串（bytes）解码为字符串（str）。

`encode()`编码：将字符串编码为字节串。

#编码一个字符串为字节串

```
encoded_bytes="HelloWorld".encode("utf-8")
print(encoded_bytes)
#解码字符串为字节串
decoded_string=encoded_bytes.decode("utf-8")
print(decoded_string)
```

# 正则表达式

2024年2月21日 12:46

## 正则表达式

正则表达式，又称规则表达式 (Regular Expression)，是使用单个字符串来描述、匹配某个句法规则的字符串，常被用来检索、替换那些符合某个模式 (规则) 的文本。

简单来说，正则表达式就是使用：字符串定义规则，并通过规则去验证字符串是否匹配。

比如，验证一个字符串是否是符合条件的电子邮箱地址，只需要配置好正则规则，即可匹配任意邮箱。

比如通过正则规则： `(^[\w-]+(\.[\w-]+)*@[\\w-]+(\.[\\w-]+)+$)` 即可匹配一个字符串是否是标准邮箱格式

但如果不用正则，使用if else来对字符串做判断就非常困难了。

## 正则的三个基础方法

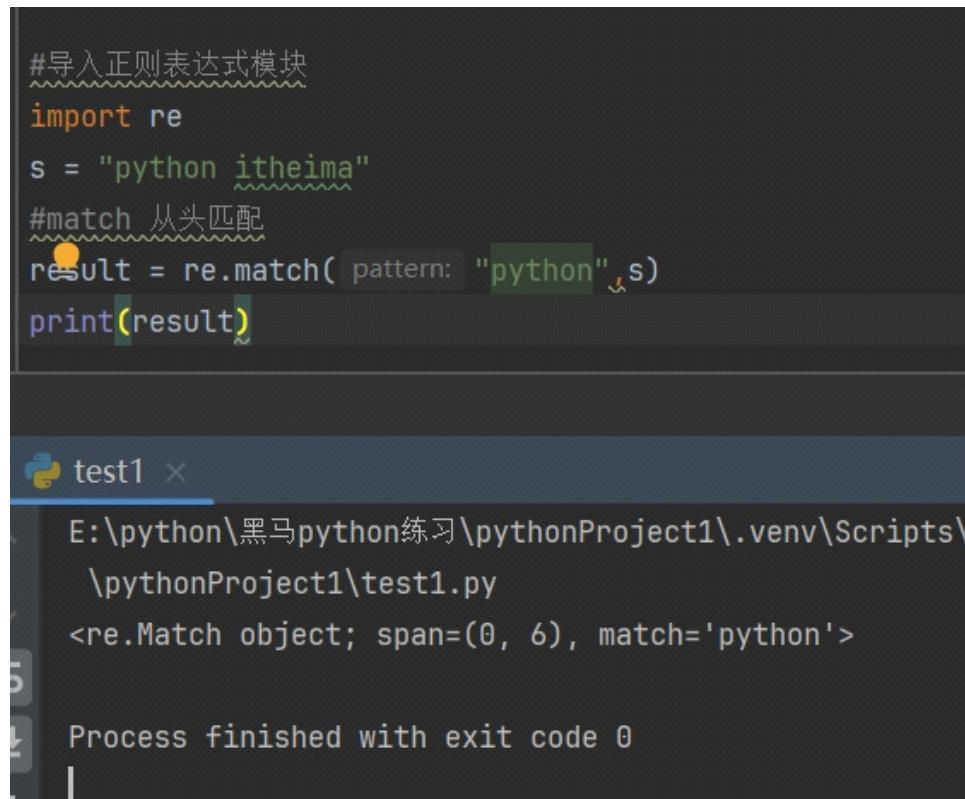
Python正则表达式，使用**re模块**，并基于re模块中三个基础方法来做正则匹配。

分别是：**match**、**search**、**findall** 三个基础方法。

### 1.re.match(匹配规则， 被匹配字符串)

从被匹配第一个字符串开始匹配，匹配成功返回匹配对象（包含匹配的信息），匹配不成功返回空。第一个空格前不匹配就结束了

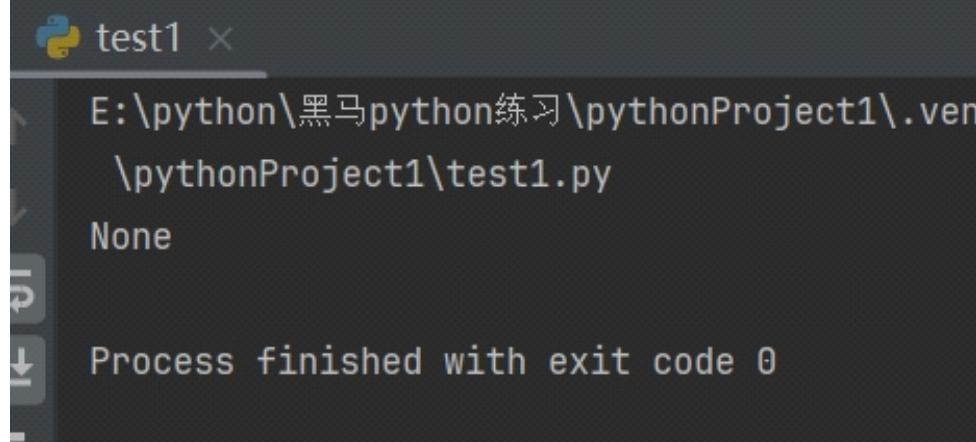
```
#导入正则表达式模块
import re
s = "python itheima"
#match 从头匹配
result = re.match(pattern: "python", s)
print(result)
```



The screenshot shows a terminal window titled 'test1' with the following content:

```
E:\python\黑马python练习\pythonProject1\.venv\Scripts\
\pythonProject1\test1.py
<re.Match object; span=(0, 6), match='python'>
Process finished with exit code 0
```

```
#导入正则表达式模块
import re
s = "1python itheima"
#match 从头匹配
result = re.match(pattern: "python", s)
print(result)
```

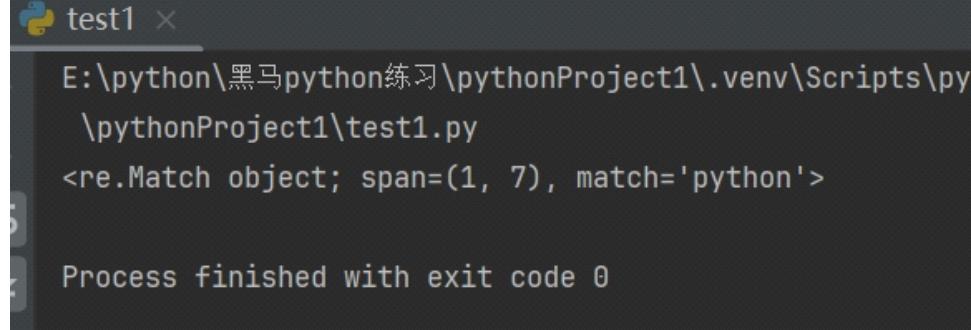


```
test1 ×
E:\python\黑马python练习\pythonProject1\.venv\pythonProject1\test1.py
None
Process finished with exit code 0
```

## 2.search(匹配规则, 被匹配字符串)

搜索整个字符串, 找出匹配的。从前向后, 找到第一个后, 就停止, 不会继续向后

```
#导入正则表达式模块
import re
s = "1python itheima"
#match 从头匹配
result = re.search(pattern: "python", s)
print(result)
```



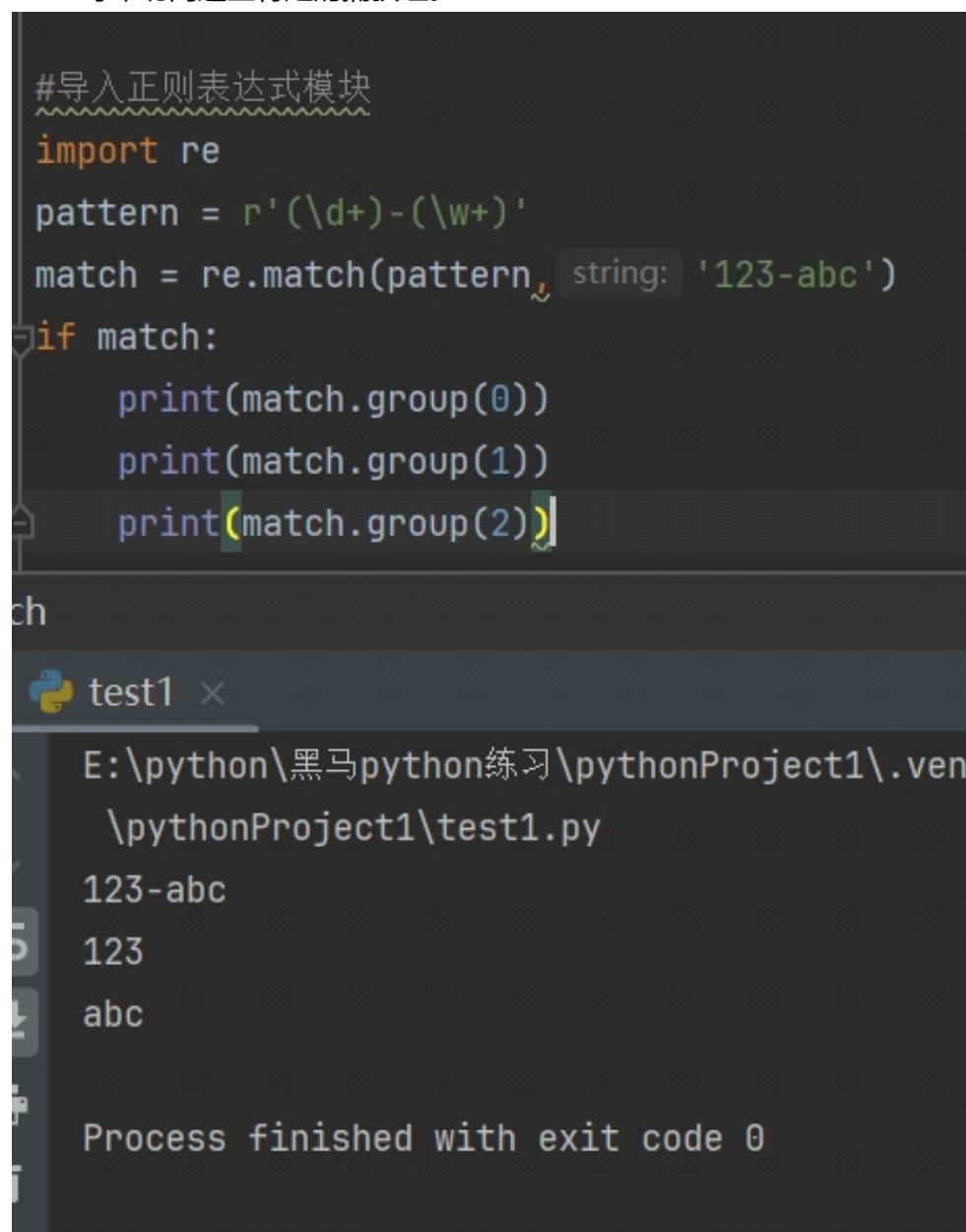
```
test1 ×
E:\python\黑马python练习\pythonProject1\.venv\Scripts\pythonProject1\test1.py
<re.Match object; span=(1, 7), match='python'>
Process finished with exit code 0
```

## group的用法

- 如果你没有在正则表达式中使用圆括号创建捕获组, 那么调用

group()、group(0)或groups()都会返回整个匹配的字符串。

- 如果你使用了圆括号创建了捕获组，那么你可以使用group(1)、group(2)等来访问这些特定的捕获组。



The screenshot shows a PyCharm interface. The code editor contains a script named 'test1.py' with the following content:

```
# 导入正则表达式模块
import re
pattern = r'(\d+)-(\w+)'  # 正则表达式，匹配以数字-字母的形式
match = re.match(pattern, string: '123-abc')
if match:
    print(match.group(0))
    print(match.group(1))
    print(match.group(2))
```

The terminal window below shows the script's output:

```
test1
E:\python\黑马python练习\pythonProject1\.venv\pythonProject1\test1.py
123-abc
123
abc

Process finished with exit code 0
```

3.findall(匹配规则, 被匹配字符串)

匹配整个字符串, 找出全部匹配项

找不到返回空list: []

```
#导入正则表达式模块
import re
s = "1python itheima"
#.findall 全部匹配
result = re.findall( pattern: "python", s)
print(result)
```

```
python test1 ×
E:\python\黑马python练习\pythonProject1\.
\pythonProject1\test1.py
['python']

Process finished with exit code 0
```

# 正则表达式-元字符匹配

2024年2月21日 15:39

## 单字符匹配：

| 字符 | 功能                       |
|----|--------------------------|
| .  | 匹配任意1个字符（除了\n）， \. 匹配点本身 |
| [] | 匹配[]中列举的字符               |
| \d | 匹配数字，即0-9                |
| \D | 匹配非数字                    |
| \s | 匹配空白，即空格、tab键            |
| \S | 匹配非空白                    |
| \w | 匹配单词字符，即a-z、A-Z、0-9、_    |
| \W | 匹配非单词字符                  |

## 示例：

字符串 s = "itheima1 @@python2 !!666 ##itcast3"

- 找出全部数字：re.findall(r '\d' , s)

字符串的r标记，表示当前字符串是原始字符串，即内部的转义字符无效而是普通字符

```
#导入正则表达式模块
import re
s = "1python@@ itheima!!!666###itccast3"
#.findall 全部匹配
result = re.findall( pattern: r"\d", s)
print(result)
```

```
test1 x
E:\python\黑马python练习\pythonProject1\.venv\Scripts\python.exe
  \pythonProject1\test1.py
  ['1', '6', '6', '6', '3']

Process finished with exit code 0
```

- 找出特殊字符:

```
re.findall(r'\W', s)
```

```
#导入正则表达式模块
import re
s = "1python@@ itheima!!!666###itccast3"
#.findall 全部匹配
result = re.findall( pattern: r'\W', s)
print(result)
```

```
test1 x
E:\python\黑马python练习\pythonProject1\.venv\Scripts\python.exe
  \pythonProject1\test1.py
  ['@', '@', ' ', '!', '!', '!', '#', '#', '#']

Process finished with exit code 0
```

- 找出全部英文字母:

```
re.findall(r'[a-zA-Z]', s)
```

[]内可以写: [a-zA-Z0-9] 这三种范围组合或指定单个字符如[aceDFG135]

```
#导入正则表达式模块
import re
s = "ipython@ itheima!!!666##itccast3ZA"
#.findall 全部匹配
result = re.findall( pattern: r'[a-zA-Z0-9]', s)
print(result)
```

### 数量匹配：

| 字符     | 功能                |
|--------|-------------------|
| *      | 匹配前一个规则的字符出现0至无数次 |
| +      | 匹配前一个规则的字符出现1至无数次 |
| ?      | 匹配前一个规则的字符出现0次或1次 |
| {m}    | 匹配前一个规则的字符出现m次    |
| {m,}   | 匹配前一个规则的字符出现最少m次  |
| {m, n} | 匹配前一个规则的字符出现m到n次  |

### 边界匹配：

| 字符 | 功能        |
|----|-----------|
| ^  | 匹配字符串开头   |
| \$ | 匹配字符串结尾   |
| \b | 匹配一个单词的边界 |
| \B | 匹配非单词边界   |

1.^如果多行模式 (re.MULTILINE) 被设置，^还会匹配每一行的开始位置。

```
import re
text = """
hello
world
hello again
"""

print(re.findall(r'^hello', text, re.MULTILINE))
# 输出: ['hello', 'hello']
```

2.\$

正则表达式r'world\$'会匹配任何以"world"结束的字符串：

```
import re
print(re.search(r'world$', 'hello world').group()) # 输出: world
print(re.search(r'world$', 'world hello')) # 不输出
```

**如果你想确保一个字符串完全由"hello"组成，你可以使用r'^hello\$'作为正则表达式。**

### 3.\b

\b 是一个单词边界匹配符，它匹配的是一个单词的开始或结束位置。换句话说，它匹配的是一个单词字符（通常是字母、数字或下划线）和一个非单词字符之间的位置。

例如，在字符串 "hello world" 中，\b 会匹配到以下位置：

- 在 "h" 前面（因为 "h" 是一个单词的开始）
- 在 "o" 后面（因为 "o" 是一个单词的结束）
- 在 " " 后面（因为它前面是一个单词结束，后面是一个非单词字符）
- 在 "w" 前面（因为它前面是一个非单词字符，后面是一个单词开始）

### 4.\B

匹配的是一个非单词边界的位置，要求所在的位置两侧都是单词字符或都不是单词字符。

r"\Bhello\B" 会匹配 "hello" 在 "123hello123" 中的位置，但不会匹配 "hello" 在 "hello world" 中的位置，因为 "hello" 的前后都有单词字符。

使用上面的例子，在字符串 "hello world" 中，\B 会匹配到以下位置：

- 在 "e" 和 "l" 之间（因为两侧都是单词字符）
- 在 "l" 和 "o" 之间（同样，两侧都是单词字符）
- 在 " " 中间（因为两侧都是非单词字符）

## 分组匹配：

| 字符  | 功能           |
|-----|--------------|
|     | 匹配左右任意一个表达式  |
| ( ) | 将括号中字符作为一个分组 |

### 1.竖线 |：

竖线 | 在正则表达式中用作“或”操作符，表示匹配其左侧或右侧的任何一个模式。

例如，如果你想匹配单词 "apple" 或 "orange"，你可以使用以下正则表达式：

```
import re
pattern = r'apple|orange'
print(re.search(pattern, 'I have an apple.').group()) # 输出: apple
print(re.search(pattern, 'I like oranges.').group()) # 输出: orange
```

### 2. ()

```
import re
pattern = r'(\d{4})-(\d{2})-(\d{2})'
```

```
match = re.search(pattern, '1999-12-31')
print(match.group(1)) # 输出: 1999
print(match.group(2)) # 输出: 12
print(match.group(3)) # 输出: 31
```

**捕获组**: 如果你不希望捕获某个组的内容, 可以使用 `?:` 前缀。

```
import re
pattern = r'(?:\d{4})-(\d{2})-(\d{2})'
match = re.search(pattern, '1999-12-31')
print(match.group(1)) # 引发 IndexError, 因为没有捕获组 1
print(match.group(2)) # 输出: 12
print(match.group(3)) # 输出: 31
```

# 正则表达式-案例

2024年2月22日 9:50

1. 匹配账号，只能由字母和数字组成，长度限制6到10位

规则为：

`^[0-9a-zA-Z]{6, 10}$`

2. 匹配QQ号，要求纯数字，长度5-11，第一位不为0规则为：

`^[1-9][0-9]{4, 10}$`

[1-9]匹配第一位，[0-9]匹配后面4到10位

3. 匹配邮箱地址，只允许qq、163、gmail这三种邮箱地址

`^[\w\.-]+(?:qq|163|gmail)\.com$`

这个正则表达式的解释如下：

`^` 表示字符串的开始。

`[\w\.-]+` 匹配一个或多个字母、数字、点(.) 或连字符(-)。

`@` 匹配@字符。

`(?:qq|163|gmail)` 是一个非捕获组，它匹配qq、163或gmail中的一个。

`\.` 匹配字符 (因为在正则表达式中是特殊字符，所以需要使用\进行转义)。

`com` 匹配com字符串。

`$` 表示字符串的结束。

# 递归

2024年2月22日 11:10

递归：即方法（函数）自己调用自己的一种特殊编程写法

```
def func():
```

```
    if ...:  
        func()  
  
    return ...
```

# zip函数

2024年4月6日 20:14

zip是一个内置函数，用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的对象，如果各个迭代器的元素个数不一致，则返回列表长度与最短的对象相同，利用 \* 号操作符，可以将元组解压为列表。

基础用法1：

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
zipped = zip(list1, list2)
print(list(zipped)) # 输出: [(1, 'a'), (2, 'b'), (3, 'c')]
```

不等长列表：

```
list1 = [1, 2, 3, 4]
list2 = ['a', 'b', 'c']
zipped = zip(list1, list2)
print(list(zipped)) # 输出: [(1, 'a'), (2, 'b'), (3, 'c')]
```

解压元组：

```
zipped = [(1, 'a'), (2, 'b'), (3, 'c')]
list1, list2 = zip(*zipped)
print(list1) # 输出: (1, 2, 3)
print(list2) # 输出: ('a', 'b', 'c')
```

与for循环结合使用：

```
List1=[1,2,3]
List2=['a','b','c']
For x,y in zip(list1,list2):
    print(x,y)
# 输出:
# 1 a
# 2 b
# 3 c
```

# os模块

2024年6月18日 20:48

os 是 Python 的一个标准库模块，提供了一系列用于与操作系统交互的函数。通过这个模块，你可以执行多种与操作系统相关的任务，例如：

- 访问文件和目录：例如，列出目录内容、创建和删除文件或目录等。
- 文件路径操作：可以拼接路径、获取文件扩展名、转换文件路径为绝对路径等。
- 环境变量管理：获取和设置环境变量。
- 操作系统信息：获取操作系统的名称、版本等信息。
- 进程管理：启动新的进程、等待进程结束等。

以下是一些使用 os 模块的常见示例：

```
import os

# 获取当前工作目录
current_directory = os.getcwd()

# 改变当前工作目录
os.chdir('/path/to/directory')

# 创建一个目录
os.mkdir('new_directory')

# 删除一个目录
os.rmdir('empty_directory')

# 列出一个目录的内容
entries = os.listdir('directory')

# 检查一个路径是否存在
path_exists = os.path.exists('path/to/file')

# 检查一个路径是否为文件
is_file = os.path.isfile('path/to/file')

# 检查一个路径是否为目录
is_directory = os.path.isdir('path/to/directory')

# 获取环境变量
environment_variable = os.getenv('ENV_VARIABLE_NAME')

比如：
api_key = os.getenv("API_KEY")  # 确保环境变量名是"API_KEY"

# 设置环境变量
os.environ['ENV_VARIABLE_NAME'] = 'value'

# 删除环境变量
```

```
os.environ.pop('ENV_VARIABLE_NAME')
```