# Python数据类型

*Python3*中有六种标准的数据类型

## 数字（Number）

Python 数字数据类型用于存储数值。

Python 支持三种不同的数值类型：整型（int）、浮点型（float）、复数（complex）

### 整型：

**通常称为整数，是整数或者负数，不带小数点。python3整型类型没有大小限制。**int型数字：0、10 -780、-089、-0x270、0x90。

### 浮点型：

**浮点型由整数部分与小数部分组成。**

float型数字：0.0、15.2、-2.9、-32.5e10、32+e18。

浮点数可以用科学计算法表示：2.5e2 = 2.5 x 10^2 = 250

### 复数型：

**复数由实数部分和虚数部分构成，可以用a + bj,或者complex(a,b)表示， 复数的实部a和虚部b都是浮点型。**

complex:3.14j、9.23e-35j、4.53e-7j。

### python数字类型相互转换

int(x) 将x转换为一个整数。 float(x) 将x转换到一个浮点数。 complex(x) 将x转换到一个复数，实数部分为 x，虚数部分为 0。complex(x, y) 将 x 和 y 转换到一个复数，实数部分为 x，虚数部分为 y。x 和 y 是数字表达式。

### python数字运算

**和其他不一样的是： / ：返回的是浮点数、// :向下取整数、 \*\* ： 幂运算**

>>>7/2  
3.5  
>>17//3  
5  
>>>2\*\*3  
8

## 字符串（String）

**双引号或者单引号中的数据，就是字符串。**

str1 = "hello,world"  
str2 = "Jim"'

**单字符**在python中也是当作一个字符串来使用。

### python访问字符串中的值

var1 = "hello world"  
var2 = "python3"  
print("var1[0]",var1[0]  #var1[0] h  
print("var2[3:6]",var2[3:6])  #var2[3:6] hon

### python字符串的分片

通过上边界不包括在内。

str = "string"  
str[1:3] # "tr"   
#获取从偏移为1到偏移为3的字符串，不包括偏移为3的字符  
​  
str[1:] # "tring"  
#获取从偏移为1到最后的一个字符，不包括最后一个字符  
​  
str[:3] #"str"  
# 获取从偏移为0的字符一直到偏移为3的字符串，不包括偏移为3的字符串  
​  
str[:-1] #strin"  
#获取从偏移为0的字符一直到最后一个字符（不包括最后一个字符串）   
​  
str[:] #"string"  
#获取字符串从开始到结尾的所有元素    
​  
str[-3:-1] #"in"  
#获取偏移为-3到偏移为-1的字符，不包括偏移为-1的字符   
​  
str[::-1] # "gnirts"  
#反转输出'

**“+”：实现字符串的拼接**

string = "hello"+"world" #"hello world"'

### 字符串的方法

#### replace()方法

Python replace() 方法把字符串中的 old（旧字符串） 替换成 new(新字符串)，如果指定第三个参数max，则替换不超过 max 次。

str = "This is A Test"  
print(str.replace("is", "was")) # Thwas was A Test"  
print(str.replace("is", "was", 1)) #Thwas is A Test'

#### find()方法

find() 方法检测字符串中是否包含子字符串 str ，如果指定 beg（开始） 和 end（结束） 范围，则检查是否包含在指定范围内，如果指定范围内如果包含指定索引值，返回的是索引值在字符串中的起始位置。如果不包含索引值，返回-1。

语法：

str.find(str,beg = 0, end = len(str))

**参数：**

***str*** -- 指定检索的字符串 ***beg*** -- 开始索引，默认为0。 ***end*** -- 结束索引，默认为字符串的长度。

**返回值：**

*如果包含子字符串返回开始的索引值，否则返回-1*。

str1 = "python web: www.python.org"

str2 = "we"

print(str1.find(str2)) # 7

print(str1.find(str2, 3)) # 7

print(str1.find(str2, 8)) # -1'

#### index()方法

和find()方法相似，唯一的区别就是find方法不包含索引值会返回-1，而index()不包含索引值会抛出异常。

#### join()方法

连接字符串数组。将字符串、元组、列表中的元素以指定的字符(分隔符)连接生成一个新的字符串。join是重要的字符串方法，用来在队列中添加元素。

**注意：需要添加的元素必须都是字符串。**

语法

str.join(sequence)

str = "-"

a = [1,2,4]

b = ["a","b","c"]

print(str.join(a)) #wrong

print(str.join(b)) #a-b-c

#### split()方法

和join方法相反，split方法把字符串分成序列。

语法

str.split(str="", num=string.count(str)).

str -- 分隔符，默认为所有的空字符，包括空格、换行(\n)、制表符(\t)等。 num -- 分割次数。 返回值 返回分割后的字符串列表。

str = "ithis is string examplei"

print(str.split()) #['ithis', 'is', 'string', 'examplei']

print(str.split("s",1)) #['ithi', ' is string examplei']

print(str.split("i")) #['', 'th', 's ', 's str', 'ng example', '']'

记住一点儿：str1.split(str2)按照 str2分割时候，就把str2 换成 "," 形式。以" i ipython oi" 为例，把字符串的所有 i 换成 ","形式，结果为：" ", " ","python o"," "

#### len()方法

Python len() 方法返回对象（字符、列表、元组等）长度或项目个数。

str = "i love python"

print(len(str)) #13

List = [1,3,5,7]

print(len(List)) #4'

#### lower()方法

Python lower() 方法转换字符串中**所有大写字符**为**小写**。

#### upper()方法

Python lower() 方法转换字符串中**所有小写字符**为**大写**。

#### isnumeric()方法

如果 **string 中只包含数字字符**，则返回 True，否则返回 False

#### isspace()方法

如果 **string 中只包含空格**，则返回 True，否则返回 False.

str = "This Is A Test"

print(str.lower()) # this is a test

print(str.upper()) # THIS IS A TEST

str = "tt44333"

print(str.isnumeric()) #False

str = "44333"

print(str.isnumeric()) #True

print(str.isspace()) #False

str=" "

print(str.isspace()) #True'

### 字符串删除字符

在 Python 中，字符串是不可变的，这意味着一旦创建，就不能修改它们。然而，我们可以通过一些方法来删除字符串中的字符。以下是一些常用的字符串删除字符的方法及其详细说明。

#### 1. 使用 str.replace()

##### 语法

str.replace(old, new[, count])

##### 参数

* **old**: 要被替换的旧字符串。
* **new**: 用于替换的新字符串。
* **count**: 可选，指定替换的次数。如果不指定，默认替换所有。

##### 作用

此方法返回一个新字符串，其中的旧字符串被新字符串替换。

##### 示例

s = "hello world"  
result = s.replace("o", "")  # 删除字符 'o'  
print(result)  # 输出: hell wrld

#### 2. 使用 str.translate()

##### 语法

str.translate(table)

##### 参数

* **table**: 一个映射表，通常使用 str.maketrans() 创建。

##### 作用

此方法根据给定的映射表替换字符串中的字符。

##### 示例

s = "hello world"  
# 创建一个映射表，删除字符 'o' 和 'l'  
table = str.maketrans("", "", "ol")  
result = s.translate(table)  
print(result)  # 输出: he wrd

#### 3. 使用列表推导式

##### 语法

new\_str = ''.join([char for char in original\_str if char not in chars\_to\_remove])

##### 参数

* **original\_str**: 原始字符串。
* **chars\_to\_remove**: 要删除的字符集合。

##### 作用

通过列表推导式生成一个新的字符串，包含原始字符串中不在要删除字符集合中的字符。

##### 示例

s = "hello world"  
chars\_to\_remove = "lo"  
result = ''.join([char for char in s if char not in chars\_to\_remove])  
print(result)  # 输出: he wrd

#### 4. 使用 str.strip(), str.lstrip(), str.rstrip()

##### 语法

str.strip([chars])  
str.lstrip([chars])  
str.rstrip([chars])

##### 参数

* **chars**: 可选，指定要删除的字符集合。如果不指定，默认删除空白字符。

##### 作用

* strip(): 删除字符串两端的指定字符。
* lstrip(): 删除字符串左端的指定字符。
* rstrip(): 删除字符串右端的指定字符。

##### 示例

s = " hello world "  
result\_strip = s.strip()  # 删除两端空白  
result\_lstrip = s.lstrip()  # 删除左端空白  
result\_rstrip = s.rstrip()  # 删除右端空白  
print(result\_strip)  # 输出: hello world  
print(result\_lstrip)  # 输出: hello world    
print(result\_rstrip)  # 输出:   hello world

#### 5. 使用 str.split() 和 str.join()

##### 语法

str.split([sep[, maxsplit]])  
str.join(iterable)

##### 参数

* **sep**: 可选，指定分隔符。
* **maxsplit**: 可选，指定分割的最大次数。
* **iterable**: 要连接的可迭代对象。

##### 作用

通过分割字符串并重新连接来删除特定字符。

##### 示例

s = "hello world"  
result = ''.join(s.split('o'))  # 删除字符 'o'  
print(result)  # 输出: hell wrld

### 字符格式化操作

age = 10

str = "python"

print("今年%d岁"%age,"the string %s"%str)'

## 列表list

列表是一个**有序**且**可更改**的集合，并且是最常用的 Python 数据类型。在 Python 中，列表是使用方括号 [] 编写的。

### 创建列表

在Python中，列表中元素的数据类型可以不同，可以包含整数、浮点数和复数等，当然，也可以包含列表、元组、字典和集合等。

#### 使用 [ ] 创建列表

创建一个列表，只需要把逗号分隔的不同数据项使用方括号 **[]** 括起来即可。

* 创建**空列表**

list0 = []

* 创建**非空列表**

list1 = ['Baidu', 'Alibaba', 'Tencent']

#### 使用 list() 函数 创建(转换为)列表

* 可以使aaaa用 list() 构造函数创建列表：

this\_list = list(('apple', 'banana', 'cherry'))

注意：在使用 **list()** 函数创建列表时，一定要注意**双括号**。

* 在Python中，我们可以使用 list() 函数将字符串、元组、字典和集合等其他类似对象转换为列表。**具体用法见下方内置函数：**

### 访问列表

和列表一样，我们既可以使用**下标索引**访问列表中的某个元素（得到一个元素的值），也可以使用**切片**访问列表中的一组元素（得到是子列表）。

#### 下标索引访问

**下标索引**访问元组分为两大类，即**正向索引**和**反向索引**，格式为 list\_name[i] ，其中，list\_name 表示列表名，i表示索引值，i可以是正数（正向索引）也可以是负数（反向索引）。 可以得知，list\_name[0]表示列表的**第一个元素**，list\_name[-1]则表示列表的**最后一个元素**。

list\_name = ['wzq', 'lgl', 'gz', 'whl', 'sj', 'hxw']

print(list\_name[0])

print(list\_name[-1])

wzq

hxw

正向索引：从第一个（下标0）开始、第二个（下标1）…

反向索引：从倒数第一个（下标-1）、倒数第二个（下标-2）…

#### 切片访问

如若对上方描述不太理解，可参考下表：

| **元组值** | **wzq** | **lgl** | **gz** | **whl** | **sj** | **hxw** |
| --- | --- | --- | --- | --- | --- | --- |
| 正向索引 | 0 | 1 | 2 | 3 | 4 | 5 |
| 反向索引 | -6 | -5 | -4 | -3 | -2 | -1 |

使用**切片**访问列表的格式为 list\_name[strat : end : step] ，其中，start 表示起始索引，end 表示结束索引，step 表示步长。

list\_name = ['wzq', 'lgl', 'gz', 'whl', 'sj', 'hxw']

print(list\_name[1:5:2])

print(list\_name[-6:-1:3])

['lgl', 'whl']

['wzq', 'whl']

在使用切片访问列表元素时，list\_name[strat : end : step]，[start：end] 是左闭右开区间，即访问不了 end 代表的元素。

#### for 循环遍历列表

可以使用 **for** 循环遍历列表中的项目：

fruit\_list = ['apple', 'pear', 'cherry']

for i in fruit\_list:

print(i)

apple

pear

cherry

#### 检查项目是否存在

要确定列表中是否存在指定的项，我们可以使用 **in** 关键字：

# 检查列表中是否存在'apple'

fruit\_list = ['apple', 'pear', 'cherry']

print('apple' in fruit\_list)

True

使用 **in** 关键字检查列表中是否存在指定项时，如果存在，则返回 **True** ；反之，则返回 **False** 。

#### 更改列表值

当我们创建列表后，我们可以对列表中的数据项进行修改或更新，当然我们也可以使用 **append()** 方法来添加列表项。

fruit\_list = ['apple', 'pear', 'cherry']

fruit\_list[2] = 'banana'

print(fruit\_list)

['apple', 'pear', 'banana']

注意：**元组**一旦创建后，其值将无法被更改，但是有其他解决方法。

#### 列表连接(合并)/复制

与字符串一样，列表之间可以使用 + 号和 \* 号实现元组的**连接**和**复制**，这就意味着它们可以生成一个新的列表。 1、+连接（合并）

x = [1, 2, 3]

y = [4, 5, 6]

print(x + y)

[1, 2, 3, 4, 5, 6]

2、\*复制

x = ['Hello']

print(x \* 5)

['Hello', 'Hello', 'Hello', 'Hello', 'Hello']

#### 嵌套列表

使用**嵌套列表**即在列表里面创建其他列表。

x = [1, 2, 3]

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

z = [x, y]

print(z)

[[1, 2, 3], ['a', 'b', 'c']]

#### 列表比较

列表比较需要引入 **operator** 模块的 **eq** 方法。

# 导入 operator 模块

import operator

a = [1, 2]

b = [2, 3]

c = [2, 3]

print("operator.eq(a, b):", operator.eq(a, b))

print("operator.eq(b, c):", operator.eq(b, c))

operator.eq(a, b): False

operator.eq(b, c): True

### 内置函数

#### 打印输出 print()

print()函数 **print()** 函数的功能我们已经非常熟悉了，就是打印输出。

my\_list = ['pink', True, 1.78, 65]

print(my\_list)

['pink', True, 1.78, 65]

#### 确定列表项目 len()

len()函数 当我们要确定一个列表有多少项目（元素）时，可以使用len()函数。

fruit\_list = ['apple', 'banana', 'cherry']

print(len(fruit\_list))

3

#### 返回变量类型 type()

type()函数 使用 **type()** 函数可以确定变量是什么类型（字符串、列表、元组、字典或集合）。

info\_list = ['name', 'gender', 'age', 'height', 'weight']

print(type(info\_list))

<class 'list'>

当对info\_list使用 **type()** 确定变量类型时，会返回<class 'list'>，表明这是一个**列表**。

#### 转换为列表 list()

tuple()函数 **tuple()** 函数的功能是，将其他类型转换为元组类型，详细用法如下：

* 将字符串转换为列表

str1 = 'Hello Python'

print(list(str1))

['H', 'e', 'l', 'l', 'o', ' ', 'P', 'y', 't', 'h', 'o', 'n']

* 将元组转换为列表

tuple1 = ('Hello', 'Python')

print(list(tuple1))

['Hello', 'Python']

* 将字典转换为列表

dict1 = {'Hello': 'Python', 'name': 'pink'}

print(list(dict1))

['Hello', 'name']

* 将集合转换为列表

set1 = {'Hello', 'Python', 'name', 'pink'}

print(list(set1))

['Python', 'name', 'pink', 'Hello']

* 将区间转换为列表

range1 = range(1, 6)

print(list(range1))

[1, 2, 3, 4, 5]

#### 元组元素最大/小值 max()、min()

max()函数和min()函数 **max()** 函数的作用是返回列表中元素**最大值**。**min()** 函数的作用是返回列表中元素**最小值**。

list1 = [4, 6, 2, 0, -5]

print(max(list1))

print(min(list1))

list2 = ['a', 'z', 'A', 'Z']

print(max(list2))

print(min(list2))

6

-5

z

A

#### 删除列表 del

* 删除单个元素 我们可以使用 del list\_name[i] 来删除某个指定元素，其中 **list\_name** 表示列表名，**i** 表示指定值的索引值。

list\_de = ['Baidu', 'Alibaba', 'Tencent', 'Bytedance']

del list\_de[1]

print(list\_de)

123

['Baidu', 'Tencent', 'Bytedance']

1

* 删除列表 **del** 函数不仅可以实现删除某个元素，还可以删除整个列表。

list\_de = ['Baidu', 'Alibaba', 'Tencent', 'Bytedance']

del list\_de

12

当我们使用 **del** 函数删除某列表后，再使用 **print()** 函数打印输出时，会报错NameError: name 'list\_de' is not defined,表明该列表未被定义。

### 内置方法

#### 添加元素

#### append()方法

**append()** 方法用于在列表末尾添加新的对象。

##### 语法

list.append(element)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *element* | 必需。任何类型（字符串、数字、对象等）的元素。 |

##### 实例

###### 添加元素

fruit\_list = ['apple', 'banana', 'cherry']

fruit\_list.append('pear')

print(fruit\_list)

['apple', 'banana', 'cherry', 'pear']

###### 添加列表

x = [1, 2, 3]

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

x.append(y)

print(x)

[1, 2, 3, ['A', 'B', 'C']]

#### insert()方法

**insert()** 方法用于将指定对象插入列表的指定位置。

##### 语法

list.insert(position, element)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *position* | 必需。数字，指定在哪个位置插入值。 |
| *element* | 必需。元素，任何类型（字符串、数字、对象等）。 |

##### 实例

* 把值 “orange” 作为第二个元素插入 fruits 列表：

fruits = ['apple', 'banana', 'cherry']

fruits.insert(1, "orange")

print(fruits)

['apple', 'orange', 'banana', 'cherry']

* 将列表 y 插入到列表 x 中

x = [1, 2, 3]

y = ['a', 'c']

x.insert(0, y)

print(x)

[['a', 'c'], 1, 2, 3]

**append()** 只能在**末尾**处添加元素或列表，**insert()** 可以在**任意位置**添加元素或列表。

#### extend()方法

**extend()** 方法用于在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）。

##### 语法

list.extend(iterable)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *iterable* | 必需。任何可迭代对象（列表、集合、元组等）。 |

##### 实例

aver = ['A', 'B', 'C']

###### 添加字符串元素到列表末尾

str1 = 'Hello'

aver.extend(str1)

print(aver)

['A', 'B', 'C', 'H', 'e', 'l', 'l', 'o']

###### 添加列表元素到列表末尾

list1 = [1, 2, 3]

aver.extend(list1)

print(aver)

['A', 'B', 'C', 1, 2, 3]

###### 添加元组元素到列表末尾

tuple1 = (1, 2, 3)

aver.extend(tuple1)

print(aver)

['A', 'B', 'C', 1, 2, 3]

###### 添加字典元素到列表末尾

dict1 = {'name': 'pink', 'gender': True}

aver.extend(dict1)

print(aver)

['A', 'B', 'C', 'name', 'gender']

###### 添加集合元素到列表末尾

set1 = {1, 2, 3}

aver.extend(set1)

print(aver)

['A', 'B', 'C', 1, 2, 3]

###### 添加区间到列表末尾

range1 = range(1,10)

aver.extend(range1)

print(aver)

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

#### 元素出现次数 count()

**count()** 方法用于统计某个元素在列表中出现的次数。

##### 语法

list.count(value)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *value* | 必需。任何类型（字符串、数字、列表、元组等）。要搜索的值。 |

##### 实例

num = [1, 4, 2, 9, 7, 8, 9, 3, 1]

print(num.count(9))

#### 指定值索引 index()

**index()** 方法用于从列表中找出某个值第一个匹配项的索引位置。

##### 语法

list.index(element)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *element* | 必需。任何类型（字符串、数字、列表等）。要搜索的值。 |

##### 实例

num = [4, 55, 64, 32, 16, 32]

print(num.index(32))

3

当被搜索值在列表中多次出现时，仅返回**首次**出现的位置。

#### 对列表排序 sort()

**sort()** 方法用于对原列表进行排序，如果指定参数，则使用比较函数指定的比较函数。

##### 语法

list.sort(reverse=True|False, key=myFunc)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| reverse | 可选。reverse=True 将对列表进行降序排序。默认是 reverse=False。 |
| key | 可选。指定排序标准的函数。 |

##### 实例

###### 以字母顺序对列表进行排序：

words = ['Name', 'Gender', 'Age', 'Height', 'Weight']

words.sort()

print(words)

['Age', 'Gender', 'Height', 'Name', 'Weight']

###### 对列表进行降序排序：

words = ['Name', 'Gender', 'Age', 'Height', 'Weight']

words.sort(reverse=True)

print(words)

['Weight', 'Name', 'Height', 'Gender', 'Age']

###### 按照值的长度对列表进行排序：

# 返回值的长度的函数：

def myfunc(e):

return len(e)

words = ['a', 'bb', 'ccc', 'dddd', '']

words.sort(key=myfunc)

print(words)

['', 'a', 'bb', 'ccc', 'dddd']

###### 根据字典的 “year” 值对字典列表进行排序：

# 返回 'year' 值的函数：

def myfunc(e):

return e['year']

words = [

{'char': 'a', 'year': 1963},

{'char': 'b', 'year': 2010},

{'char': 'c', 'year': 2019}

]

words.sort(key=myfunc)

print(words)

[{'char': 'a', 'year': 1963}, {'char': 'b', 'year': 2010}, {'char': 'c', 'year': 2019}]

###### 按照值的长度对列表进行降序排序：

# 返回值的长度的函数：

def myfunc(e):

return len(e)

words = ['aa', 'b', 'ccc', 'dddd']

words.sort(reverse=True, key=myfunc)

print(words)

['dddd', 'ccc', 'aa', 'b']

###### 指定列表中的元素排序来输出列表：

# 获取列表的第二个元素

def takeSecond(elem):

return elem[1]

# 列表

random = [(2, 2), (3, 4), (4, 1), (1, 3)]

# 指定第二个元素排序

random.sort(key=takeSecond)

# 输出类别

print('排序列表：', random)

排序列表： [(4, 1), (2, 2), (1, 3), (3, 4)]

#### 复制列表 copy()

**copy()** 方法用于复制列表，类似于 **a[:]**。

##### 语法

list.copy()

##### 参数值

无参数

##### 实例

fruits = ['apple', 'banana', 'cherry', 'orange']

x = fruits.copy()

print(x)

['apple', 'banana', 'cherry', 'orange']

复制（制作副本）的另一种方法是使用内置函数 **list()** ，如下：

list1 = ['apple', 'banana', 'cherry']

list\_2 = list(list1)

#### 颠倒列表顺序 reverse()

**reverse()** 方法用于反向列表中元素。

##### 语法

list.reverse()

**参数值** 无参数 **实例**

fruits = ['apple', 'banana', 'cherry']

fruits.reverse()

print(fruits)

['cherry', 'banana', 'apple']

#### 删除元素

#### pop()方法

**pop()** 方法用于移除列表中的一个元素（默认最后一个元素），并且返回该元素的值。

##### 语法

list.pop(pos)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *pos* | 可选。数字，指定需删除元素的位置。默认值 -1，返回最后的项目。 |

##### 实例

* 当 **pos** 未指定时，默认删除最后的元素

fruits = ['apple', 'banana', 'cherry']

fruits.pop()

print(fruits)

['apple', 'banana']

* **pos** 指定要删除元素的位置

fruits = ['apple', 'banana', 'cherry']

fruits.pop(1)

print(fruits)

['apple', 'cherry']

#### remove()方法

**remove()** 方法用于移除列表中某个值的第一个匹配项。

##### 语法

list.remove(element)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *element* | 必需。需删除的任何类型（字符串、数字、列表等）的元素。 |

##### 实例

num = [1, 3, 2, 8, 3]

num.remove(3)

print(num)

[1, 2, 8, 3]

当被删除的元素在列表中存在多个时，默认删除**首次**出现的那个。

#### clear()方法

**clear()** 方法用于清空列表，类似于 **del a[:]**。

##### 语法

list.clear()

##### 参数值

无参数

##### 实例

word = ['A', 'B', 'C']

word.clear()

print(word)

[]

**clear()** 方法的作用是**清空**列表，执行结束后对其使用 print() 打印输出时，会输出 [] ,说明列表还存在，只是**空列表**而已。

**del** 函数的作用是**删除**列表，执行结束后对其使用 print() 打印输出时，会报错 NameError: name 'word' is not defined.。

### 总结

| **函数** | **描述** |
| --- | --- |
| print() | 打印输出 |
| len() | 确定列表项目 |
| type() | 返回变量类型 |
| list() | 转换为列表 |
| max() | 返回列表元素最大值 |
| min() | 返回列表元素最小值 |
| del | 删除列表 |
| 方法 | 描述 |
| ------------------------------ | ------------------------------------------------------------ |
| append(obj) | 在列表末尾添加新的对象 |
| insert(index, obj) | 在指定位置添加元素 |
| extend(seq) | 将列表元素（或任何可迭代的元素）添加到当前列表的末尾 |
| count(obj) | 统计某个元素在列表中出现的次数 |
| index(obj) | 返回具有指定值的第一个元素的索引 |
| sort( key=None, reverse=False) | 对原列表进行排序 |
| copy() | 复制列表 |
| reverse() | 颠倒列表的顺序 |
| pop([-1]) | 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值 |
| remove(obj) | 移除列表中某个值的第一个匹配项 |
| clear() | **清空列表** |

## 元组tuple

元组是**有序**且**不可更改**的集合。在Python中，元组使用圆括号 () 编写的。

### 创建元组

元组的创建很简单，使用圆括号 **()** 直接创建或者使用 **tuple()** 函数创建，只需要在圆括号中添加元素，并使用逗号隔开即可。

#### 使用 () 创建元组

通过 **()** 创建元组后，使用 **=** 将它赋值给变量，格式如下：

tuple\_name = (element\_1, element\_2, ..., element\_n)

其中，tuple\_name 表示变量名，element\_1 ~ element\_n 表示元组内的元素。

#### 使用 tuple()函数 创建元组

我们除了可以使用 **()** 创建元组，还可以使用 **tuple()** 函数创建元组，但 tuple() 偏向于将某个类型转换为元组，*具体用法见下述：*

cre\_tuple = tuple(('a', 'b', 'c')) # 注意双括号

print(cre\_tuple)

print(type(cre\_tuple))

('a', 'b', 'c')

<class 'tuple'>

#### 元组(单个元素)

当元组中只包含一个元素时，需要在元素后面添加逗号, ，否则括号会被当作运算符使用：

tuple\_1 = (20)

print(type(tuple\_1)) # <class 'int'>

tuple\_2 = (20,)

print(type(tuple\_2)) # <class 'tuple'>

12345

当元组中**只含一个元素未添加逗号**时，使用 type() 打印其类型时，会输出 <class 'int'>。

当元组中**只含一个元素并添加逗号**时，使用type() 打印其类型时，会输出 <class 'tuple'>。

#### 元组 VS 列表

* 在Python中，元组与列表相似，不同之处在于**元组的元素不能修改**，而**列表的元素可以修改**。
* 元组使用 小括号()，列表使用 中括号[]。

### 访问元组

和列表一样，我们既可以使用**下标索引**访问元组中的某个元素（得到一个元素的值），也可以使用**切片**访问元组中的一组元素（得到是子元组）。

#### 下标索引访问

**下标索引**访问元组分为两大类，即**正向索引**和**反向索引**，格式为 tuple\_name[i] ，其中，tuple\_name 表示元组名，i表示索引值，i可以是正数（正向索引）也可以是负数（反向索引）。 可以得知，tuple\_name[0]表示元组内**第一个元素**，tuple\_name[-1]则表示元组内**最后一个元素**。

tuple\_name = ('wzq', 'lgl', 'gz', 'whl', 'sj', 'hxw')

print(tuple\_name[0])

print(tuple\_name[-1])

wzq

hxw

正向索引：从第一个（下标0）开始、第二个（下标1）…

反向索引：从倒数第一个（下标-1）、倒数第二个（下标-2）…

#### 切片访问

如若对上方描述不太理解，可参考下表：

| **元组值** | **wzq** | **lgl** | **gz** | **whl** | **sj** | **hxw** |
| --- | --- | --- | --- | --- | --- | --- |
| 正向索引 | 0 | 1 | 2 | 3 | 4 | 5 |
| 反向索引 | -6 | -5 | -4 | -3 | -2 | -1 |

使用**切片**访问元组的格式为 tuple\_name[strat : end : step] ，其中，start 表示起始索引，end 表示结束索引，step 表示步长。

tuple\_name = ('wzq', 'lgl', 'gz', 'whl', 'sj', 'hxw')

print(tuple\_name[1:5:2])

print(tuple\_name[-6:-1:3])

('lgl', 'whl')

('wzq', 'whl')

在使用切片访问元组元素时，tuple\_name[strat : end : step]，[start：end] 是左闭右开区间，即访问不了 end 代表的元素。

#### for 循环遍历元组

可以使用 **for** 循环遍历元组中的项目：

fruit\_tuple = ('apple', 'pear', 'cherry')

for i in fruit\_tuple:

print(i)

apple

pear

cherry

#### 检查项目是否存在

要确定元组中是否存在指定的项，我们可以使用 **in** 关键字：

# 检查元组中是否存在'apple'

fruit\_tuple = ('apple', 'pear', 'cherry')

print('apple' in fruit\_tuple)

True

使用 **in** 关键字检查元组中是否存在指定项时，如果存在，则返回 **True** ；反之，则返回 **False** 。

#### 更改元组值

创建元组后，我们将无法更改其值。元组是不可变的，或者也被称为恒定的。 但是有一种解决方法：可以**先将元组转换为列表，更改列表值，然后再将其转换回元组**。

fruit\_tuple = ('apple', 'pear', 'cherry')

fruit\_list = list(fruit\_tuple)

fruit\_list[2] = 'banana'

fruit\_tuple = tuple(fruit\_list)

print(fruit\_tuple)

('apple', 'pear', 'banana')

#### 元组连接(合并)/复制

与字符串一样，元组之间可以使用 + 号和 \* 号实现元组的**连接**和**复制**，这就意味着它们可以生成一个新的元组。

##### +连接

x = (1, 2, 3)

y = (4, 5, 6)

print(x + y)

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

##### \*复制

x = ('Hello',)

print(x \* 5)

('Hello', 'Hello', 'Hello', 'Hello', 'Hello')

#### 元组不可变性

所谓元组的不可变性指的是**元组所指向的内存中的内容不可变。**

tup = ('A', 'B', 'C')

# tup[0] = 'a'

不支持修改元素，会报错 TypeError: 'tuple' object does not support item assignment 。

tup = ('A', 'B', 'C')

print(id(tup))

# 查看内存地址

tup = (1, 2, 3)

print(id(tup))

2671405067264

2671405072256

从以上实例可以看出，重新赋值的元组 tup ，绑定到了新的对象了，不是修改了原来的对象。

### 内置函数

#### 打印输出 print()

print()函数 **print()** 函数的功能我们已经非常熟悉了，就是打印输出。

my\_tuple = ('name', True, 21)

print(my\_tuple)

('name', True, 21)

#### 确定元组项目 len()

len()函数 当我们要确定一个元组有多少项目（元素）时，可以使用len()函数。

fruit\_tuple = ('apple', 'banana', 'cherry')

print(len(fruit\_tuple))

3

#### 返回变量类型 type()

type()函数 使用 **type()** 函数可以确定变量是什么类型（字符串、列表、元组、字典或集合）。

当对tup\_1 = (20)使用 **type()** 确定变量类型时，会返回<class 'int'>，表明这是一个**整型**。所以当一个元组只含有一个元素时，一定要加上逗号,。

当对tup\_1 = (20,)使用 **type()** 确定变量类型时，会返回<class 'tuple'>，表明这是一个**元组**。

#### 转换为元组 tuple()

tuple()函数 **tuple()** 函数的功能是，将其他类型转换为元组类型，详细用法如下：

##### 将字符串转换为元组

str1 = 'Hello Python'

print(tuple(str1))

('H', 'e', 'l', 'l', 'o', ' ', 'P', 'y', 't', 'h', 'o', 'n')

##### 将列表转换为元组

list1 = ['Hello', 'Python']

print(tuple(list1))

('Hello', 'Python')

##### 将字典转换为元组

dict1 = {'Hello': 'Python', 'name': 'pink'}

print(tuple(dict1))

('Hello', 'name')

##### 将集合转换为元组

set1 = {'Hello', 'Python', 'name', 'pink'}

print(tuple(set1))

('Hello', 'name', 'Python', 'pink')

##### 将区间转换为元组

range1 = range(1, 6)

print(tuple(range1))

(1, 2, 3, 4, 5)

#### 元组元素最大/小值 max()、min()

max()函数和min()函数 **max()** 函数的作用是返回元组中元素**最大值**。**min()** 函数的作用是返回元组中元素**最小值**。

tup1 = (4, 6, 2, 0, -5)

print(max(tup1))

print(min(tup1))

tup2 = ('a', 'z', 'A', 'Z')

print(max(tup2))

print(min(tup2))

6

-5

z

A

#### 删除元组 del

元组中的元素值是不允许删除的，但我们可以使用 **del** 函数来删除整个元组。

word\_tuple = ('a', 'b', 'c')

del word\_tuple

当我们使用 **del** 函数删除某元组后，再使用 **print()** 函数打印输出时，会报错NameError: name 'word\_tuple' is not defined,表明该元组未被定义。

### 内置方法

#### 元素出现次数 count()

**count()** 方法返回指定值在元组中出现的次数。

##### 语法

tuple.count(value)

##### 参数值

| **参数** | **秒速** |
| --- | --- |
| *value* | 必需。要检索的项目。 |

##### 实例

返回值 5 在元组中出现的次数：

num\_tuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5, 5)

print(num\_tuple.count(5))

3

#### 元素位置 index()

**index()** 方法查找指定值的第一次出现。

##### 语法

tuple.index(value)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *value* | 必需。要检索的项目。 |

##### 实例

num\_tuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5, 5)

print(num\_tuple.index(5))

5

注意：当被搜索值在元组中存在多个，则返回**第一次出现**的下标。

num\_tuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5, 5)

print(num\_tuple.index(999))

如果未找到该值，index() 方法将引发异常，ValueError: tuple.index(x): x not in tuple。

### 小结

| **函数** | **描述** |
| --- | --- |
| print() | 打印输出 |
| len() | 确定元组项目 |
| type() | 返回变量类型 |
| tuple() | 转换为元组 |
| max() | 返回元组元素最大值 |
| min() | 返回元组元素最小值 |
| del | 删除元组 |
| 方法 | 描述 |
| ------- | ---------------------------------------- |
| count() | 返回元组中指定值出现的次数 |
| index() | 在元组中搜索指定的值并返回它被找到的位置 |

## 集合（Set）

集合（set）是一个**无序**的**不重复**元素序列。

### 构造集合

可以使用大括号 **{ }** 或者 **set()** 函数创建集合，注意：创建一个空集合必须用 **set()** 而不是 **{ }**，因为 **{ }** 是用来创建一个空字典。

#### 使用大括号{}

basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}

print(basket) # {'apple', 'orange', 'banana', 'pear'}'

当使用print()函数打印输出时，会自动**将集合中重复的元素去除**，且每次打印**返回的顺序并不和初始的一致**。

#### 使用set()函数

fruit = set(("apple", "banana", "cherry")) # 请留意这个双括号

print(fruit) # {'banana', 'apple', 'cherry'}'

在使用set()函数创建集合时，一定要注意**双括号**。

### 基本计算

经过上述学习，我们已经知道如何创建集合。而两个集合之间的计算有4种，分别是-、|、&和^。

先创建两个集合，方便下面对其进行运算。

a = set('abcdefg')

b = set('abg')'

#### -

print(a - b) # {'e', 'd', 'f', 'c'}

**a - b** 表示：集合 a 中包含而集合 b 中不包含的元素

#### |

print(a | b) # {'d', 'b', 'e', 'c', 'g', 'f', 'a'}

**a | b** 表示：集合 a 或集合 b 中包含的所有元素

#### &

print(a & b) # {'a', 'b', 'g'}

**a & b** 表示：集合 a 和集合 b 都包含了的元素

#### ^(反交集)

print(a ^ b) # {'e', 'c', 'd', 'f'}

**a ^ b** 表示：不同时包含集合 a 和集合 b 的元素

### 访问项目

我们无法通过引用索引来访问 set 中的项目，因为 set 是**无序**的，项目**没有索引**。

但是可以使用 **for** 循环遍历 set 项目，或者使用 **in** 关键字查询集合中是否存在指定值。

#### 实例

遍历集合，并打印值

fruit = {'apple', 'banana', 'cherry'}

for key in fruit:

print(key)'

检查 set 中是否存在 “banana”

fruit = {'apple', 'banana', 'cherry'}

print('banana' in fruit)'

若输出True，则表示指定值存在于该集合中；若输出False，则表示指定值不存在于该集合中

### 内置函数

#### 打印输出 print()

print()函数

从上述构造集合中，我们可以得知 **print()** 的作用就是打印输出**小括号()** 里的内容或值。

#### 计算集合内元素 len()

len()函数

当我们想要获取集合 set 的长度，即确定集合中有多少项，就要使用 **len()** 方法。

print(len(basket)) # 4

print(len(fruit)) # 3

我们注意到，在上述构造集合 basket 和 fruit 时，在 **{}** 中填入**6个元素**，当我们使用**len()函数**定长时，会发现其自动**去除重复的元素**，再返回。

#### 返回变量类型 type()

type()函数

print(type(basket)) # <class 'set'>

当我们使用 **{}** 或 **set()** 构造好集合时，对其使用 **type()** 函数，会输出set，则表明这是一个集合。

#### 删除集合 del

del函数

我们已经构造好 basket 和 fruit 这两个集合，如果需要对其删除则使用**del函数**。

fruit = set(("apple", "banana", "cherry")) # 请留意这个双括号

del fruit

print(fruit) # NameError: name 'fruit' is not defined.

注意：使用完**del()\**函数删除集合后，如再用*\*print()函数**打印时，会报错NameError: name 'fruit' is not defined.，意为**fruit**未被定义，说明已删除。

### 内置方法

### 添加元素

#### add()方法

add() 方法用于给集合添加元素，如果添加的元素在集合中已存在，则不执行任何操作。

##### 语法

set.add(element)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *element* | 必需。要添加到集合的元素。 |

**实例**

向 fruits 集合添加一个元素：

fruits = {"apple", "banana", "cherry"}

fruits.add("orange")

print(fruits)'

输出结果：

{'apple', 'banana', 'orange', 'cherry'}

**如果该元素已存在，则 add() 方法就不会添加元素。**

fruits = {"apple", "banana", "cherry"}

fruits.add("apple")

print(fruits)'

输出结果：

{'apple', 'banana', 'cherry'}

#### updata()方法

update() 方法用于修改当前集合，可以添加新的元素或集合到当前集合中，如果添加的元素在集合中已存在，则该元素只会出现一次，重复的会忽略。

##### 语法

set.update(set)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set* | 必需。插入当前集合的集合。 |

##### 实例

合并两个集合，重复元素只会出现一次：

x = {"apple", "banana", "cherry"}

y = {"google", "runoob", "apple"}

x.update(y)

print(x)'

输出结果：

{'banana', 'apple', 'google', 'runoob', 'cherry'}

#### 移除元素

#### remove()方法

remove() 方法用于移除集合中的指定元素。

该方法不同于 **discard()** 方法，因为 **remove()** 方法在移除一个不存在的元素时会发生错误，而 **discard()** 方法不会。

##### 语法

set.remove(item)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *item* | 必需。需要检索并删除的项目。 |

##### 实例

移除元素 banana：

fruits = {"apple", "banana", "cherry"}

fruits.remove("banana")

print(fruits)'

输出结果：

{'cherry', 'apple'}

#### discard()方法

discard() 方法用于移除指定的集合元素。

该方法不同于 **remove()** 方法，因为 **remove()** 方法在移除一个不存在的元素时会发生错误，而 **discard()** 方法不会。

##### 语法

set.discard(value)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *value* | 必需。要检索并删除的项目。 |

##### 实例

移除集合中的元素 banana：

fruits = {"apple", "banana", "cherry"}

fruits.discard("banana")

print(fruits)'

输出结果：

{'cherry', 'apple'}

#### pop()方法

pop() 方法用于随机移除一个元素。

##### 语法

set.pop()

##### 参数值

无参数值。

##### 实例

随机移除一个元素：

fruits = {"apple", "banana", "cherry"}

fruits.pop()

print(fruits)'

输出结果：

{'apple', 'banana'}

使用**pop()方法**删除项目，但此方法将删除最后一项。

同时请记住，集合set是**无序**的，因此您也不知道被删除的是什么项目。

#### 清空集合 clear()

clear()方法

clear() 方法用于移除集合中的所有元素。

##### 语法

set.clear()

##### 参数值

无参数。

##### 实例

移除 fruits 集合中的所有元素：

fruits = {"apple", "banana", "cherry"}

fruits.clear()

print(fruits)'

输出结果：

set()

特别注意：clear() 功能是移除集合中所有元素，而del 功能是删除集合。

所以，当对一个集合使用clear()方法或del函数，并分别对其使用print()函数打印输出时，前者会返回一个空的集合即set()，而后者会报错即NameError: name 'fruit' is not defined.

clear() 从内存删除集合与清空集合，但**内存地址**不删除，del() 则会从内存中删除。

#### 合并集合

在Python中，有几种方法可以连接两个或多个集合。

可以使用 **union()** 方法返回包含两个集合中所有项目的新集合，也可以使用 **update()** 方法将一个集合中的所有项目插入另一个集合中。

#### union()方法

union() 方法返回两个集合的并集，即包含了所有集合的元素，重复的元素只会出现一次。

##### 语法

set.union(set1, set2 ...)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set1* | 必需。要整合的集合。 |
| *set2* | 可选。要整合的另一集合。您能够比较任意多的集合。集合由逗号分隔。 |

##### 实例

合并两个集合，重复元素只会出现一次：

set1 = {'a', 'b', 'c'}

set2 = {1, 2, 3, 'c'}

print(set1.union(set2))'

输出结果：

{1, 2, 'b', 3, 'c', 'a'}

合并多个集合：

set1 = {'a', 'b', 'c'}

set2 = {1, 2, 3}

set3 = {'A', 'B'}

print(set1.union(set2, set3))'

输出结果：

{1, 2, 3, 'B', 'b', 'A', 'a', 'c'}

#### update()方法

**update()** 方法在前面已经讲解过，在此仅作实例演示。

##### 实例

**update()** 方法将 **set2** 中的项目插入 **set1** 中：

set1 = {'a', 'b', 'c'}

set2 = {1, 2, 3}

set1.update(set2)

print(set1)'

输出结果：

{1, 2, 3, 'b', 'a', 'c'}

**注释**：

* union() 和 update() 都将排除任何重复项。
* 还有其他方法将两个集合连接起来，并且仅保留重复项，或者永远不保留重复项。

#### 拷贝集合 copy()

copy() 方法用于拷贝一个集合。

##### 语法

set.copy()

##### 参数值

无参数。

##### 实例

复制 fruits 集合：

fruits = {"apple", "banana", "cherry"}

print(fruits.copy())'

输出结果：

{'apple', 'banana', 'cherry'}

#### 判断是否包含相同元素 isdisjoint()

isdisjoint() 方法用于判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False。

##### 语法

set.isdisjoint(set)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set* | 必需。要在其中检索相同项目的集合。 |

##### 实例

判断集合 y 中是否有包含 集合 x 的元素：

x = {'a', 'b', 'c'}

y = {'x', 'y', 'z'}

print(x.isdisjoint(y))'

输出结果：

True

**注意**：**isdisjoint()** 方法在判断两个集合是否包含相同元素时，如果**不包含相同的元素则返回True**，而**包含相同的元素则返回False**

#### 返回交集

#### intersection()方法

intersection() 方法用于返回两个或更多集合中都包含的元素，即交集。

##### 语法

set.intersection(set1, set2 ... etc)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set1* | 必需。要在其中检索相同项目的集合。 |
| *set2* | 可选。其他需要在其中检索相同项目的集合。您可以比较任意多的集合。集合之间由逗号分隔。 |

##### 实例

返回一个新集合，该集合的元素既包含在集合 x 又包含在集合 y 中：

x = {'a', 'b', 'c'}

y = {'x', 'y', 'z', 'c'}

print(x.intersection(y))'

输出结果：

{'c'}

计算多个集合的交集：

x = {'a', 'b', 'c'}

y = {'x', 'y', 'z', 'c'}

z = {'c', 'a', 'd'}

print(x.intersection(y, z))'

输出结果：

{'c'}

返回的集合仅包含两个集合中都存在的项目，或者如果使用两个以上的集合进行比较，则在所有集合中都存在。

#### intersection\_update()方法

intersection\_update() 方法用于获取两个或更多集合中都重叠的元素，即计算交集。

##### 语法

set.intersection\_update(set1, set2 ... etc)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set1* | 必需。要在其中检索相等项目的集合。 |
| *set2* | 可选。要在其中检索相等项目的另一集合。您能够比较任意多的集合。集合由逗号分隔。 |

##### 实例

移除 x 集合中不存在于 y 集合中的元素：

x = {'a', 'b', 'c'} # y 集合不包含 a，被移除

y = {'x', 'y', 'b', 'c'}

x.intersection\_update(y)

print(x)'

输出结果：

{'c'}

intersection\_update() 方法不同于 intersection() 方法，因为 intersection() 方法是返回一个新的集合，而 intersection\_update() 方法是在原始的集合上移除不重叠的元素。

#### 判断子集

#### issubset()方法

issubset() 方法用于判断集合的所有元素是否都包含在指定集合中，如果是则返回 True，否则返回 False。

##### 语法

set.issubset(set)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set* | 必需。在其中检索相等项目的集合。 |

##### 实例

如果集合 y 中存在集合 x 中的所有项目，则返回 True：

x = {'a', 'b', 'c'}

y = {'x', 'y', 'z', 'a', 'b', 'c'}

print(x.issubset(y))'

输出结果：

True

如果没有全部包含返回 False：

x = {'a', 'b', 'c'}

y = {'x', 'y', 'z', 'a'}

print(x.issubset(y))'

输出结果：

False

注意：使用x.issubset(y)时，简单地说，当集合 x 中所有项目都存在于集合 y 中时，返回**True**，反之（集合 x 中有一个或多个都不存在于集合 y 中，就返回 **False**）。

#### issuperset()方法

issuperset() 方法用于判断指定集合的所有元素是否都包含在原始的集合中，如果是则返回 True，否则返回 False。

##### 语法

set.issuperset(set)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set* | 必需。要在其中检索相等项目的集合。 |

##### 实例

判断集合 y 的所有元素是否都包含在集合 x 中：

x = {'a', 'b', 'c', 'x', 'y', 'z'}

y = {'a', 'b', 'c'}

print(x.issuperset(y))'

输出结果：

True

如果没有全部包含返回 False：

x = {'a', 'x', 'y', 'z'}

y = {'a', 'b', 'c'}

print(x.issuperset(y))'

输出结果：

False

**issuperset()** 和 **issubset()** 类似，如果全都包含就返回True，反之（有一个或多个未包含）就返回False。

#### 差集

#### difference()方法

difference() 方法用于返回集合的差集，即返回的集合元素包含在第一个集合中，但不包含在第二个集合(方法的参数)中。

##### 语法

set.difference(set)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set* | 必需。要检查其中差异的项目。 |

##### 实例

返回一个集合，元素包含在集合 x ，但不在集合 y ：

x = {'a', 'b', 'c', 'x', 'y', 'z'}

y = {'a', 'b', 'c'}

print(x.difference(y))'

输出结果：

{'x', 'y', 'z'}

#### difference\_update()方法

difference\_update() 方法用于移除两个集合中都存在的元素。

##### 语法

set.difference\_update(set)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set* | 必需。要检查其中差异的集合。 |

##### 实例

移除两个集合都包含的元素：

x = {'a', 'b', 'c', 'x', 'y', 'z'}

y = {'a', 'b', 'c'}

x.difference\_update(y)

print(x)'

输出结果：

{'x', 'z', 'y'}

difference\_update() 方法与 difference() 方法的区别在于 difference() 方法**返回一个移除相同元素的新集合**，而 difference\_update() 方法是**直接在原来的集合中移除元素，没有返回值**。

#### 移除元素

#### symmetric\_difference()方法

symmetric\_difference() 方法返回两个集合中不重复的元素集合，即会移除两个集合中都存在的元素。

##### 语法

set.symmetric\_difference(set)

##### 参数值

| **参数** | **描述** |
| --- | --- |
| *set* | 必需。用于检查匹配项的集合。 |

##### 实例

返回两个集合组成的新集合，但会移除两个集合的重复元素：

x = {'a', 'b', 'c', 'x', 'y', 'z'}

y = {'a', 'b', 'c', 'Q', 'W', 'E'}

print(x.symmetric\_difference(y))'

输出结果：

{'x', 'Q', 'z', 'E', 'y', 'W'}

#### symmetric\_difference\_update()方法

symmetric\_difference\_update() 方法移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。

##### 语法

set.symmetric\_difference\_update(set)

**参数值**

| **参数** | **描述** |
| --- | --- |
| *set* | 必需。用于检查匹配项的集合。 |

##### 实例

在原始集合 x 中移除与 y 集合中的重复元素，并将不重复的元素插入到集合 x 中：

x = {'a', 'b', 'c', 'x', 'y', 'z'}

y = {'a', 'b', 'c', 'Q', 'W', 'E'}

x.symmetric\_difference\_update(y)

print(x)'

输出结果：

{'z', 'Q', 'x', 'W', 'E', 'y'}

### 小结

| **函数** | **描述** |
| --- | --- |
| print() | 打印输出 |
| len() | 计算集合内元素 |
| type() | 返回变量类型 |
| del | 删除集合 |

| **方法** | **描述** |
| --- | --- |
| add() | 为集合添加元素 |
| update() | 给集合添加元素 |
| remove() | 移除指定元素 |

| **discard()** | **删除集合中指定的元素** |
| --- | --- |
| pop() | 随机移除元素 |
| clear() | 移除集合中的所有元素 |
| union() | 返回两个集合的并集 |
| copy() | 拷贝一个集合 |

| **isdisjoint()** | **判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False** |
| --- | --- |
| intersection() | 返回集合的交集 |
| intersection\_update() | 返回集合的交集 |
| issubset() | 判断指定集合是否为该方法参数集合的子集 |
| issuperset() | 判断该方法的参数集合是否为指定集合的子集 |
| difference() | 返回多个集合的差集 |
| difference\_update() | 移除集合中的元素，该元素在指定的集合也存在 |
| symmetric\_difference() | 返回两个集合中不重复的元素集合。 |
| symmetric\_difference\_update() | 移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中 |

## 字典（Dictionary）

字典（Dictionary）是Python中非常重要和实用的数据结构，用于存储键值对（key-value pairs）。字典在很多编程场景中非常有用，因为它们提供了一种高效的方式来管理和访问数据。本文将详细讲解如何创建字典)、访问字典中的元素、修改字典元素、删除字典元素、查找元素是否是字典的键、标准类型操作符以及常用内置函数。

### 创建字典

字典是由一系列键值对组成的无序集合。键是唯一的，且必须是不可变的类型（如字符串、数字或元组），值可以是任何类型。字典使用花括号 {} 创建，键和值之间用冒号 : 分隔，键值对之间用逗号 , 分隔。

示例：

# 创建一个空字典

empty\_dict = {}

# 创建一个包含键值对的字典

person = {"name": "Alice", "age": 25, "city": "New York"}

# 使用 dict() 函数创建字典

person = dict(name="Alice", age=25, city="New York")

# 创建嵌套字典

nested\_dict = {"name": "Alice", "details": {"age": 25, "city": "New York"}}

print(empty\_dict) # 输出：{}

print(person) # 输出：{'name': 'Alice', 'age': 25, 'city': 'New York'}

print(nested\_dict) # 输出：{'name': 'Alice', 'details': {'age': 25, 'city': 'New York'}}

### 访问字典中的元素

字典中的元素可以通过键来访问。如果键不存在，会引发 KeyError 异常。可以使用 **get()** 方法在键不存在时返回默认值，而不是引发异常。

示例：

person = {"name": "Alice", "age": 25, "city": "New York"}

# 使用键访问值

print(person["name"]) # 输出：Alice

print(person["age"]) # 输出：25

# 使用 get() 方法访问值

print(person.get("city")) # 输出：New York

print(person.get("country")) # 输出：None

print(person.get("country", "USA")) # 输出：USA

### 修改字典元素

字典中的值可以通过键进行修改。如果键不存在，会添加一个新的键值对。

示例：

person = {"name": "Alice", "age": 25, "city": "New York"}

# 修改现有键的值

person["age"] = 26

print(person) # 输出：{'name': 'Alice', 'age': 26, 'city': 'New York'}

# 添加新的键值对

person["country"] = "USA"

print(person) # 输出：{'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}

### 删除字典元素

可以使用 del 语句或 pop() 方法删除字典中的元素。pop() 方法还会返回被删除的值。popitem() 方法删除并返回字典中的最后一个键值对（在Python 3.7+中，字典是有序的）。

示例：

person = {"name": "Alice", "age": 25, "city": "New York", "country": "USA"}

# 使用 del 语句删除元素

del person["country"]

print(person) # 输出：{'name': 'Alice', 'age': 25, 'city': 'New York'}

# 使用 pop() 方法删除元素

age = person.pop("age")

print(age) # 输出：25

print(person) # 输出：{'name': 'Alice', 'city': 'New York'}

# 使用 popitem() 方法删除并返回最后一个键值对

last\_item = person.popitem()

print(last\_item) # 输出：('city', 'New York')

print(person) # 输出：{'name': 'Alice'}

#### 使用 del 语句删除字典元素

del 语句用于从字典中删除指定的键及其对应的值。使用此方法时，如果尝试删除的键不存在，会引发 KeyError 异常。

person = {

"name": "Alice",

"age": 25,

"city": "New York",

"country": "USA"

}

# 使用 del 语句删除键 "country"

del person["country"]

print(person) # 输出：{'name': 'Alice', 'age': 25, 'city': 'New York'}

#### 使用 pop() 方法删除字典元素

pop() 方法用于删除指定的键，并返回该键对应的值。如果指定的键不存在，同样会引发 KeyError 异常。可以提供一个默认值参数，当键不存在时返回该默认值而不引发异常。

person = {

"name": "Alice",

"age": 25,

"city": "New York"

}

# 使用 pop() 方法删除键 "age" 并返回其值

age = person.pop("age")

print(age) # 输出：25

print(person) # 输出：{'name': 'Alice', 'city': 'New York'}

# 使用 pop() 方法删除键 "country" 并提供默认值

country = person.pop("country", "Not Found")

print(country) # 输出：Not Found

print(person) # 输出：{'name': 'Alice', 'city': 'New York'}

#### 使用 popitem() 方法删除并返回最后一个键值对

popitem() 方法删除并返回字典中的最后一个键值对。在 Python 3.7 及之后的版本中，字典是有序的，因此 popitem() 会删除字典中的最后一个元素。如果字典为空，调用 popitem() 会引发 KeyError 异常。

person = {

"name": "Alice",

"city": "New York"

}

# 使用 popitem() 方法删除并返回最后一个键值对

last\_item = person.popitem()

print(last\_item) # 输出：('city', 'New York')

print(person) # 输出：{'name': 'Alice'}

### 查找元素是否是字典的键

可以使用 in 操作符检查一个键是否存在于字典中。

示例：

person = {"name": "Alice", "age": 25, "city": "New York"}

# 检查键是否存在

print("name" in person) # 输出：True

print("country" in person) # 输出：False

# 检查键是否不存在

print("name" not in person) # 输出：False

print("country" not in person) # 输出：True

#### 检查键是否存在

使用 in 操作符可以直接检查键是否在字典中。

# 检查键 "name" 是否存在于字典 person 中

print("name" in person) # 输出：True

# 检查键 "country" 是否存在于字典 person 中

print("country" in person) # 输出：False

* "name" in person 返回 True，因为字典 person 中确实存在键 "name"。
* "country" in person 返回 False，因为字典 person 中不存在键 "country"。

#### 检查键是否不存在

同样地，我们可以使用 not in 操作符来检查键是否不在字典中。

# 检查键 "name" 是否不存在于字典 person 中

print("name" not in person) # 输出：False

# 检查键 "country" 是否不存在于字典 person

print("country" not in person) # 输出：True

* "name" not in person 返回 False，因为字典 person 中存在键 "name"。
* "country" not in person 返回 True，因为字典 person 中确实不存在键 "country"。

### 标准类型操作符

字典支持一些常用的标准类型操作符，例如键值对的长度、合并两个字典等。

### 获取字典长度

使用 len() 函数可以获取字典中键值对的数量。

示例：

person = {"name": "Alice", "age": 25, "city": "New York"}

print(len(person)) # 输出：3

### 合并两个字典

在Python 3.9+中，可以使用 | 操作符合并两个字典。

示例：

dict1 = {"name": "Alice"}

dict2 = {"age": 25, "city": "New York"}

merged\_dict = dict1 | dict2

print(merged\_dict) # 输出：{'name': 'Alice', 'age': 25, 'city': 'New York'}

在早期版本中，可以使用 update() 方法合并字典。

示例：

dict1 = {"name": "Alice"}

dict2 = {"age": 25, "city": "New York"}

dict1.update(dict2)

print(dict1) # 输出：{'name': 'Alice', 'age': 25, 'city': 'New York'}

### 常用内置函数

Python提供了一些内置函数，用于操作和处理字典。

#### keys() 方法

keys() 方法返回字典中所有的键。

示例：

person = {"name": "Alice", "age": 25, "city": "New York"}

print(person.keys()) # 输出：dict\_keys(['name', 'age', 'city'])

#### values() 方法

values() 方法返回字典中所有的值。

示例：

person = {"name": "Alice", "age": 25, "city": "New York"}

print(person.values()) # 输出：dict\_values(['Alice', 25, 'New York'])

#### items() 方法

items() 方法返回字典中所有的键值对。

示例：

person = {"name": "Alice", "age": 25, "city": "New York"}

print(person.items()) # 输出：dict\_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])

### 字典的遍历

可以使用 for 循环遍历字典的键、值或键值对。

示例：

person = {"name": "Alice", "age": 25, "city": "New York"}

# 遍历键

for key in person.keys():

print(key)

# 遍历值

for value in person.values():

print(value)

# 遍历键值对

for key, value in person.items():

print(f"{key}: {value}")

## Python操作符

python操作符分类：**算数操作符**；**比较操作符**；**逻辑操作符**；**成员操作符**；**身份操作符**；

### 算数操作符：

常用的算数操作符：+,−,∗,/,+,−,∗,/,;

python如何执行除法： 许多编程语言中整数除法执行的地板除（计算结果取比商小的最大整数，也就是舍弃小数）； 但是python对于整数间的除法返回的是浮点型的精确数值，但是python使用双斜杠代替地板除法：

>>>3/2  
1.5  
>>>3//2  
1

### 比较运算符：

常用的比较操作符：<,≤,≥,＞,＝＝,!＝

比较操作符根据表达式返回布尔类型的值;

### 逻辑操作符：

常用的逻辑操作符：and, or, not

**and操作符**：只有当操作符左边的操作数为真，且右边的操作符同时为真的时候，结果为真；

**or操作符**：左边或右边任意一边为真，结果都为真；只有当两边同时为假，结果才为假。；

**not操作符**：一元操作符，得到一个和操作数相反的布尔类型的值；

### 成员操作符：

成员操作符：in, not in;

**判断某个元素是否在数据类型里**

**in**: 如果在指定的序列中找到值返回True，否则返回False;

**not in**: 如果在指定的序列中没有找到值返回True，否则返回False;

### 身份操作符：

身份操作符：**判断两个对象是否相等**

**is**： 判断两个标识符是否引用一个对象；

**is not**： 判断两个标识符是否引用不同对象；

# Python循环

### for 语句

for 语句是编程中用于遍历序列（如列表、元组、字符串）或其他可迭代对象的一种循环结构。它允许你逐个访问序列中的每个元素，并对其进行操作。for 语句的基本语法如下：

#### 语法格式

for 变量 in 可迭代对象:  
   循环体

#### 参数要求和作用

* **变量**：每次循环迭代时，变量会被赋值为可迭代对象中的一个元素。
* **可迭代对象**：可以是列表、元组、字符串、字典、集合等任何可迭代对象。
* **循环体**：包含在for循环中的代码块，每次迭代时都会执行。

#### 具体例子

##### 1. 遍历列表

# 遍历列表中的每个元素  
fruits = ['apple', 'banana', 'cherry']  
for fruit in fruits:  
   print(fruit)

输出：

apple  
banana  
cherry

##### 2. 遍历字符串

# 遍历字符串中的每个字符  
message = "Hello, World!"  
for char in message:  
   print(char)

输出：

H  
e  
l  
l  
o  
,  
  
W  
o  
r  
l  
d  
!

##### 3. 遍历字典

# 遍历字典的键  
fruit\_color = {'apple': 'red', 'banana': 'yellow', 'cherry': 'black'}  
for fruit in fruit\_color:  
   print(fruit)

输出：

apple  
banana  
cherry

##### 4. 遍历字典的键值对

# 遍历字典的键值对  
for fruit, color in fruit\_color.items():  
   print(f"Fruit: {fruit}, Color: {color}")

输出：

Fruit: apple, Color: red  
Fruit: banana, Color: yellow  
Fruit: cherry, Color: black

##### 5. 使用range()函数

# 生成0到4的整数序列  
for i in range(5):  
   print(i)

输出：

0  
1  
2  
3  
4

##### 6. 使用enumerate()函数

# 同时获取索引和值  
for index, fruit in enumerate(fruits):  
   print(f"Index: {index}, Fruit: {fruit}")

输出：

Index: 0, Fruit: apple  
Index: 1, Fruit: banana  
Index: 2, Fruit: cherry

##### 7. 使用zip()函数

# 同时遍历多个列表  
colors = ['red', 'yellow', 'black']  
for fruit, color in zip(fruits, colors):  
   print(f"Fruit: {fruit}, Color: {color}")

输出：

Fruit: apple, Color: red  
Fruit: banana, Color: yellow  
Fruit: cherry, Color: black

##### 8. 使用reversed()函数

# 反向遍历列表  
for fruit in reversed(fruits):  
   print(fruit)

输出：

cherry  
banana  
apple

##### 9. 使用sorted()函数

# 按顺序遍历列表  
numbers = [3, 1, 4, 1, 5, 9, 2]  
for num in sorted(numbers):  
   print(num)

输出：

1

1

2

3

4

5

9

#### 嵌套for循环

for 语句可以嵌套使用，即在一个for循环内部再嵌套另一个for循环。这种结构常用于处理二维数据或需要多层遍历的情况。

##### 例子：遍历二维列表

matrix = [  
  [1, 2, 3],  
  [4, 5, 6],  
  [7, 8, 9]  
]  
​  
for row in matrix:  
   for element in row:  
       print(element, end=' ')  
   print()

输出：

1 2 3   
4 5 6   
7 8 9

#### for 循环中的控制语句

在for循环中，可以使用break和continue语句来控制循环的执行流程。

##### 1. break 语句

break 语句用于终止整个循环，跳出循环体。

# 查找列表中的第一个偶数  
numbers = [3, 5, 7, 2, 8, 4]  
for num in numbers:  
   if num % 2 == 0:  
       print(f"找到第一个偶数: {num}")  
       break

输出：

找到第一个偶数: 2

##### 2. continue 语句

continue 语句用于跳过当前循环的剩余部分，直接进入下一轮循环。

# 遍历列表，跳过值为'banana'的元素  
fruits = ['apple', 'banana', 'cherry']  
for fruit in fruits:  
   if fruit == 'banana':  
       continue  
   print(fruit)

输出：

apple  
cherry

#### 总结

for 语句是Python中非常强大且常用的循环结构，用于遍历序列或其他可迭代对象。通过合理使用for循环，可以简化代码逻辑，提高代码的可读性和效率

### while语句

#### 1. 语法格式

while 条件表达式:  
   循环体

#### 2. 参数要求和作用

* **条件表达式**：这是一个布尔表达式，当其值为True时，循环体中的代码会被执行。一旦条件表达式的值变为False，循环终止。

#### 3. 返回值和返回值的类型与作用

* while循环本身没有返回值，它是一个控制流语句，用于重复执行一段代码。
* 循环体中的代码可以有返回值，但这与while循环本身无关。

#### 4. 常用的搭配

* **循环变量**：通常在循环体内定义一个变量，用于控制循环的次数或条件。
* **初始化和更新**：在循环开始前初始化循环变量，在每次循环结束时更新循环变量。

#### 5. 具体例子

**打印1到10的数字**

i = 1  
while i <= 10:  
   print(i)  
   i += 1

**使用break跳出循环**

while True:  
   user\_input = input("请输入'exit'退出循环：")  
   if user\_input == 'exit':  
       break  
   print("输入的不是'exit'，继续循环。")

**使用continue跳过当前循环**

count = 0  
while count < 5:  
   count += 1  
   if count % 2 == 0:  
       continue  
   print("奇数：", count)

#### 6. 算法中的应用

**寻找素数**：使用while循环检查一个数是否为素数。

def is\_prime(num):  
   if num <= 1:  
       return False  
   divisor = 2  
   while divisor \* divisor <= num:  
       if num % divisor == 0:  
           return False  
       divisor += 1  
   return True  
​  
# 测试  
for number in range(2, 21):  
   if is\_prime(number):  
       print(f"{number} 是素数")

**计算阶乘**：使用while循环计算一个数的阶乘。

def factorial(n):  
   result = 1  
   i = 1  
   while i <= n:  
       result \*= i  
       i += 1  
   return result  
​  
# 测试  
print(factorial(5))  # 应输出120

**模拟时钟**：使用while循环模拟时钟的运行。

hour = 0  
minute = 0  
while hour < 24:  
   minute = 0  
   while minute < 60:  
       print(f"{hour:02d}:{minute:02d}")  
       minute += 1  
   hour += 1

通过这些例子和应用，我们可以看到while循环在Python中的强大和灵活性，它能够处理各种复杂的逻辑和算法问题。

# Python函数

在Python中，def语句用于定义函数。函数是可重用的代码块，用于执行特定任务。下面将详细讲解def语句的各种用法和方法，包括匿名函数、生成器函数、装饰器、递归函数、高阶函数、闭包、偏函数、类型提示和类型检查、文档字符串和注释、异常处理以及上下文管理器。

### 1. 基本函数定义

#### 语法格式

def 函数名(参数1, 参数2, ...):  
   """  
  文档字符串（可选）  
  """  
   函数体  
   return 返回值（可选）

#### 参数要求和作用

##### 位置参数

位置参数是按照顺序传递的参数，必须提供。

def add(a, b):  
   return a + b  
​  
result = add(3, 5)  # result = 8

##### 默认参数

默认参数在定义函数时提供默认值，调用时可以不传递。

def greet(name="World"):  
   return f"Hello, {name}!"  
​  
result1 = greet()  # result1 = "Hello, World!"  
result2 = greet("Alice")  # result2 = "Hello, Alice!"

##### 可变参数

可变参数允许传递任意数量的位置参数，使用\*args表示。

def sum\_numbers(\*args):  
   return sum(args)  
​  
result = sum\_numbers(1, 2, 3, 4)  # result = 10

##### 关键字参数

关键字参数允许传递任意数量的关键字参数，使用\*\*kwargs表示。

def create\_profile(\*\*kwargs):  
   return kwargs  
​  
result = create\_profile(name="Alice", age=30)  # result = {'name': 'Alice', 'age': 30}

#### 返回值和返回值的类型与作用

函数可以返回一个值，使用return语句。返回值可以是任意类型，包括基本类型（如整数、字符串）、复合类型（如列表、字典）以及自定义类型（如类实例）。

##### 返回基本类型

def add(a, b):  
   return a + b  
​  
result = add(3, 5)  # result = 8

##### 返回复合类型

def get\_list():  
   return [1, 2, 3, 4]  
​  
result = get\_list()  # result = [1, 2, 3, 4]

##### 返回自定义类型

class Person:  
   def \_\_init\_\_(self, name, age):  
       self.name = name  
       self.age = age  
​  
def create\_person(name, age):  
   return Person(name, age)  
​  
person = create\_person("Alice", 30)  
print(person.name)  # 输出: Alice  
print(person.age)   # 输出: 30

### 2. 匿名函数（Lambda函数）

Lambda函数，也称为匿名函数，是Python中一种简洁的函数定义方式。它适用于那些只需要短暂使用一次的简单函数。Lambda函数使用lambda关键字定义，并且没有函数名。下面将详细讲解Lambda函数的用法、特点以及一些常见的应用场景。

#### 基本语法

Lambda函数的基本语法如下：

lambda 参数1, 参数2, ... : 表达式

* lambda：关键字，表示定义一个Lambda函数。
* 参数1, 参数2, ...：函数的参数列表，可以有多个参数，也可以没有参数。
* 表达式：函数的返回值，只能是一个表达式，不能包含语句。

#### 示例

##### 1. 基本用法

add = lambda x, y: x + y  
result = add(3, 5)  # result = 8

在这个例子中，lambda x, y: x + y定义了一个Lambda函数，它接受两个参数x和y，并返回它们的和。

##### 2. 无参数的Lambda函数

greet = lambda: "Hello, World!"  
result = greet()  # result = "Hello, World!"

这个Lambda函数没有参数，返回一个固定的字符串。

##### 3. 使用条件表达式

is\_even = lambda x: x % 2 == 0  
result = is\_even(4)  # result = True

这个Lambda函数接受一个参数x，返回x是否为偶数的布尔值。

#### 特点

* **简洁**：Lambda函数非常简洁，适合用于简单的操作。
* **匿名**：Lambda函数没有函数名，通常用于一次性操作。
* **单行表达式**：Lambda函数只能包含一个表达式，不能包含多行语句。
* **无需return**：表达式的结果会自动返回，无需使用return关键字。

#### 应用场景

##### 1. 与内置函数结合使用

Lambda函数常与内置函数如map、filter、sorted等结合使用，以简化代码。

# 使用map函数  
numbers = [1, 2, 3, 4, 5]  
squared\_numbers = list(map(lambda x: x\*\*2, numbers))  
# squared\_numbers = [1, 4, 9, 16, 25]  
​  
# 使用filter函数  
even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))  
# even\_numbers = [2, 4]  
​  
# 使用sorted函数  
students = [('Alice', 23), ('Bob', 20), ('Charlie', 25)]  
sorted\_students = sorted(students, key=lambda x: x[1])  
# sorted\_students = [('Bob', 20), ('Alice', 23), ('Charlie', 25)]

##### 2. 作为高阶函数的参数

Lambda函数可以作为高阶函数的参数，用于传递简单的操作。

def apply\_function(func, value):  
   return func(value)  
​  
result = apply\_function(lambda x: x \* 2, 5)  # result = 10

##### 3. 在字典中使用

Lambda函数可以用于字典的get方法，以提供默认值。

data = {'a': 1, 'b': 2}  
value = data.get('c', lambda: 0)()  # value = 0

#### 注意事项

* **可读性**：虽然Lambda函数简洁，但过度使用可能会降低代码的可读性。对于复杂的操作，建议使用常规的def定义函数。
* **调试困难**：由于Lambda函数没有名字，调试时可能会比较困难。
* **单行限制**：Lambda函数只能包含一个表达式，不能包含多行语句，这限制了它的功能。

通过这些详细的讲解，我们可以全面了解Lambda函数的用法、特点以及常见的应用场景。合理使用Lambda函数可以大大提高代码的简洁性和可读性。

### 3. 生成器函数

生成器函数是Python中一种特殊的函数，它使用yield关键字来产生一个序列的值，而不是使用return关键字返回一个单一的值。生成器函数在执行过程中能够暂停和恢复，这使得它们非常适合处理大数据集或无限序列，因为它们不需要一次性将所有数据加载到内存中。

#### 基本语法

生成器函数的基本语法如下：

def 生成器函数名():  
   # 函数体  
   yield 表达式

* def：定义函数的关键字。
* 生成器函数名：生成器函数的名称。
* yield：关键字，用于产生一个值，并暂停函数的执行，保存当前的状态。

#### 生成器的常用函数

##### 1. next(generator)

* **参数要求**：一个生成器对象。
* **作用**：获取生成器的下一个值。
* **返回值**：生成器的下一个值。
* **返回值类型**：生成器产生的值的类型。

gen = simple\_generator()  
print(next(gen))  # 输出: 1  
print(next(gen))  # 输出: 2

##### 2. send(value)

* **参数要求**：一个值。
* **作用**：向生成器发送一个值，并获取生成器的下一个值。
* **返回值**：生成器的下一个值。
* **返回值类型**：生成器产生的值的类型。

def counter():  
   count = 0  
   while True:  
       value = yield count  
       if value is not None:  
           count = value  
       count += 1  
​  
gen = counter()  
print(next(gen))  # 输出: 0  
print(gen.send(5))  # 输出: 6

##### 3. throw(type[, value[, traceback]])

* **参数要求**：异常类型，可选的异常值和回溯。
* **作用**：在生成器内部抛出一个异常。
* **返回值**：生成器的下一个值或异常。
* **返回值类型**：生成器产生的值的类型或异常。

def simple\_generator():  
   try:  
       yield 1  
   except ValueError:  
       yield "Caught an error"  
   yield 2  
​  
gen = simple\_generator()  
print(next(gen))  # 输出: 1  
print(gen.throw(ValueError))  # 输出: Caught an error  
print(next(gen))  # 输出: 2

##### 4. close()

* **参数要求**：无。
* **作用**：关闭生成器。
* **返回值**：无。

def simple\_generator():  
   yield 1  
   yield 2  
   yield 3  
​  
gen = simple\_generator()  
print(next(gen))  # 输出: 1  
gen.close()  
# print(next(gen)) # 抛出 StopIteration 异常

#### 示例

##### 1. 基本用法

def simple\_generator():  
   yield 1  
   yield 2  
   yield 3  
​  
gen = simple\_generator()  
print(next(gen))  # 输出: 1  
print(next(gen))  # 输出: 2  
print(next(gen))  # 输出: 3

在这个例子中，simple\_generator是一个生成器函数，它使用yield关键字产生三个值。通过调用next(gen)，我们可以逐个获取这些值。

##### 2. 斐波那契数列生成器

def fibonacci(n):  
   a, b = 0, 1  
   for \_ in range(n):  
       yield a  
       a, b = b, a + b  
​  
for num in fibonacci(10):  
   print(num)  # 输出: 0 1 1 2 3 5 8 13 21 34

##### 3. 无限序列生成器

def infinite\_sequence():  
   num = 0  
   while True:  
       yield num  
       num += 1  
​  
gen = infinite\_sequence()  
for \_ in range(5):  
   print(next(gen))  # 输出: 0 1 2 3 4

##### 4. 使用生成器进行惰性计算

def lazy\_square(numbers):  
   for num in numbers:  
       yield num \* num  
​  
numbers = [1, 2, 3, 4, 5]  
gen = lazy\_square(numbers)  
for value in gen:  
   print(value)  # 输出: 1 4 9 16 25

#### 生成器在算法中的应用

生成器在算法中的应用非常广泛，特别是在需要处理大量数据或需要惰性计算的场景中。例如，在处理大数据集时，生成器可以显著减少内存消耗。

##### 1. 归并排序中的生成器

def merge\_sorted\_generators(gen1, gen2):  
   value1 = next(gen1, None)  
   value2 = next(gen2, None)  
   while value1 is not None or value2 is not None:  
       if value1 is None:  
           yield value2  
           value2 = next(gen2, None)  
       elif value2 is None:  
           yield value1  
           value1 = next(gen1, None)  
       elif value1 < value2:  
           yield value1  
           value1 = next(gen1, None)  
       else:  
           yield value2  
           value2 = next(gen2, None)  
​  
gen1 = (x for x in [1, 3, 5])  
gen2 = (x for x in [2, 4, 6])  
merged = merge\_sorted\_generators(gen1, gen2)  
for value in merged:  
   print(value)  # 输出: 1 2 3 4 5 6

##### 2. 图遍历中的生成器

def depth\_first\_search(graph, start):  
   visited, stack = set(), [start]  
   while stack:  
       vertex = stack.pop()  
       if vertex not in visited:  
           yield vertex  
           visited.add(vertex)  
           stack.extend(graph[vertex] - visited)  
​  
graph = {  
   'A': {'B', 'C'},  
   'B': {'A', 'D', 'E'},  
   'C': {'A', 'F'},  
   'D': {'B'},  
   'E': {'B', 'F'},  
   'F': {'C', 'E'}  
}  
​  
for vertex in depth\_first\_search(graph, 'A'):  
   print(vertex)  # 输出: A B D E F C

通过这些详细的解释和示例代码，Python生成器的强大功能和灵活应用得到了全面的展示。生成器在处理大数据集、惰性计算和算法实现中具有重要作用，能够显著提高程序的性能和可维护性

#### 特点

* **延迟计算**：生成器函数只有在需要时才会计算下一个值，这使得它们非常适合处理大数据集或无限序列。
* **状态保存**：生成器函数在每次yield后会保存当前的状态，下次调用时会从上次暂停的地方继续执行。
* **内存高效**：由于生成器函数是逐个产生值，而不是一次性返回所有值，因此它们在内存使用上非常高效。
* **可迭代**：生成器函数返回的生成器对象是可迭代的，可以使用for循环或next函数来获取值。

#### 应用场景

##### 1. 处理大数据集

生成器函数非常适合处理大数据集，因为它们不需要一次性将所有数据加载到内存中。

def read\_large\_file(file\_path):  
   with open(file\_path, 'r') as file:  
       for line in file:  
           yield line  
​  
for line in read\_large\_file('large\_file.txt'):  
   print(line)  # 逐行处理大文件

##### 2. 生成无限序列

生成器函数可以用于生成无限序列，因为它们只在需要时计算下一个值。

def infinite\_sequence():  
   num = 0  
   while True:  
       yield num  
       num += 1  
​  
gen = infinite\_sequence()  
for \_ in range(10):  
   print(next(gen))  # 输出前10个自然数

##### 3. 与itertools结合使用

生成器函数可以与itertools模块结合使用，以实现更复杂的迭代逻辑。

import itertools  
​  
def squares():  
   for num in itertools.count(1):  
       yield num \*\* 2  
​  
for square in itertools.islice(squares(), 10):  
   print(square)  # 输出前10个自然数的平方

#### 注意事项

* **状态保存**：生成器函数在每次yield后会保存当前的状态，但这也意味着如果生成器函数中有大量状态需要保存，可能会占用较多的内存。
* **单向通信**：生成器函数与调用者之间的通信是单向的，即生成器函数只能通过yield产生值，而不能从外部接收值。
* **不能重复使用**：生成器函数一旦被耗尽（即所有值都被yield完毕），就不能再次使用。需要重新创建生成器对象。

通过这些详细的讲解，我们可以全面了解生成器函数的用法、特点以及常见的应用场景。合理使用生成器函数可以大大提高代码的效率和可读性。

### 4. 装饰器

装饰器是Python中一种非常强大的功能，它允许我们在不修改原有函数代码的情况下，增加或修改函数的行为。装饰器本质上是一个函数，它接受一个函数作为参数，并返回一个新的函数。装饰器通常用于日志记录、访问控制、性能计数等场景。

#### 基本语法

装饰器的基本语法如下：

@装饰器函数  
def 被装饰函数():  
   # 函数体  
   pass

等价于：

def 被装饰函数():  
   # 函数体  
   pass  
​  
被装饰函数 = 装饰器函数(被装饰函数)

#### 示例

##### 1. 简单装饰器

def my\_decorator(func):  
   def wrapper():  
       print("Something is happening before the function is called.")  
       func()  
       print("Something is happening after the function is called.")  
   return wrapper  
​  
@my\_decorator  
def say\_hello():  
   print("Hello!")  
​  
say\_hello()

输出：

Something is happening before the function is called.  
Hello!  
Something is happening after the function is called.

在这个例子中，my\_decorator是一个装饰器函数，它接受say\_hello函数作为参数，并返回一个新的函数wrapper。wrapper函数在调用say\_hello函数前后打印一些信息。

##### 2. 带参数的装饰器

如果被装饰的函数需要参数，我们可以在wrapper函数中使用\*args和\*\*kwargs来传递参数。

def my\_decorator(func):  
   def wrapper(\*args, \*\*kwargs):  
       print("Something is happening before the function is called.")  
       result = func(\*args, \*\*kwargs)  
       print("Something is happening after the function is called.")  
       return result  
   return wrapper  
​  
@my\_decorator  
def say\_hello(name):  
   print(f"Hello, {name}!")  
​  
say\_hello("Alice")

输出：

Something is happening before the function is called.  
Hello, Alice!  
Something is happening after the function is called.

##### 3. 带参数的装饰器函数

有时候，我们希望装饰器本身也能接受参数。这时，我们需要再嵌套一层函数。

def repeat(num\_times):  
   def decorator(func):  
       def wrapper(\*args, \*\*kwargs):  
           for \_ in range(num\_times):  
               result = func(\*args, \*\*kwargs)  
           return result  
       return wrapper  
   return decorator  
​  
@repeat(num\_times=3)  
def greet(name):  
   print(f"Hello, {name}!")  
​  
greet("Bob")

输出：

Hello, Bob!  
Hello, Bob!  
Hello, Bob!

在这个例子中，repeat是一个带参数的装饰器函数，它接受一个参数num\_times，并返回一个装饰器。

#### 特点

* **代码复用**：装饰器可以让我们在多个函数中复用相同的功能，而不需要重复编写代码。
* **不修改原有代码**：装饰器可以在不修改原有函数代码的情况下，增加或修改函数的行为。
* **灵活性**：装饰器可以接受参数，这使得它们非常灵活，能够适应各种不同的需求。
* **可读性**：使用装饰器可以让代码更加简洁和易读，特别是在需要添加日志记录、性能计数等功能时。

#### 应用场景

##### 1. 日志记录

def log\_decorator(func):  
   def wrapper(\*args, \*\*kwargs):  
       print(f"Calling function: {func.\_\_name\_\_}")  
       result = func(\*args, \*\*kwargs)  
       print(f"Function {func.\_\_name\_\_} finished.")  
       return result  
   return wrapper  
​  
@log\_decorator  
def add(a, b):  
   return a + b  
​  
print(add(3, 4))

输出：

Calling function: add  
Function add finished.  
7

##### 2. 性能计数

import time  
​  
def timer\_decorator(func):  
   def wrapper(\*args, \*\*kwargs):  
       start\_time = time.time()  
       result = func(\*args, \*\*kwargs)  
       end\_time = time.time()  
       print(f"Function {func.\_\_name\_\_} took {end\_time - start\_time} seconds to run.")  
       return result  
   return wrapper  
​  
@timer\_decorator  
def slow\_function():  
   time.sleep(2)  
   print("Function executed.")  
​  
slow\_function()

输出：

Function executed.  
Function slow\_function took 2.0001230239868164 seconds to run.

##### 3. 访问控制

def require\_admin(func):  
   def wrapper(\*args, \*\*kwargs):  
       if not is\_admin():  
           raise PermissionError("Admin access required.")  
       return func(\*args, \*\*kwargs)  
   return wrapper  
​  
@require\_admin  
def delete\_user(user\_id):  
   print(f"User {user\_id} deleted.")  
​  
def is\_admin():  
   # 模拟检查用户是否为管理员  
   return False  
​  
try:  
   delete\_user(123)  
except PermissionError as e:  
   print(e)

输出：

Admin access required.

#### 注意事项

* **函数签名**：装饰器会改变被装饰函数的签名，可以使用functools.wraps来保留原始函数的元数据。

from functools import wraps  
​  
def my\_decorator(func):  
   @wraps(func)  
   def wrapper(\*args, \*\*kwargs):  
       print("Something is happening before the function is called.")  
       result = func(\*args, \*\*kwargs)  
       print("Something is happening after the function is called.")  
       return result  
   return wrapper  
​  
@my\_decorator  
def say\_hello(name):  
   print(f"Hello, {name}!")  
​  
print(say\_hello.\_\_name\_\_)  # 输出: say\_hello

* **多层装饰器**：可以对一个函数应用多个装饰器，装饰器的顺序是从内到外。

@decorator1  
@decorator2  
def my\_function():  
   pass

等价于：

my\_function = decorator1(decorator2(my\_function))

通过这些详细的讲解，我们可以全面了解装饰器的用法、特点以及常见的应用场景。合理使用装饰器可以大大提高代码的复用性和可读性。

### 5. 递归函数

递归函数是Python中一种强大的编程技巧，它允许函数调用自身。递归函数在解决一些问题时非常有效，特别是那些可以分解为更小的相同问题的场景。下面详细讲解递归函数的定义、特点、参数要求、返回值、应用场景以及具体的例子。

#### 递归函数的定义和特点

##### 定义

递归函数是指在函数体内调用自身的函数。递归函数通常包含两个部分：

* **基准条件（Base Case）**：递归终止的条件，避免无限递归。
* **递归条件（Recursive Case）**：函数调用自身的部分。

##### 特点

* **简洁性**：递归函数可以使代码更简洁和易读。
* **逻辑性**：递归函数能够清晰地表达问题的分解过程。
* **内存消耗**：递归函数可能会消耗更多的内存，因为每次递归调用都会在调用栈上创建一个新的帧。

#### 参数要求和返回值

##### 参数要求

递归函数的参数通常包括：

* **输入数据**：用于计算的数据。
* **递归控制变量**：用于控制递归的深度或进度。

##### 返回值

递归函数的返回值通常是：

* **基准条件的返回值**：当满足基准条件时，返回一个确定的结果。
* **递归调用的返回值**：将递归调用的结果进行处理后返回。

#### 应用场景

递归函数适用于以下场景：

* **分治算法**：将问题分解为更小的子问题。
* **树和图的遍历**：如深度优先搜索（DFS）。
* **数学计算**：如阶乘、斐波那契数列等。

#### 具体例子

##### 1. 计算阶乘

阶乘是一个经典的递归问题。

def factorial(n):  
   # 基准条件  
   if n == 0 or n == 1:  
       return 1  
   # 递归条件  
   else:  
       return n \* factorial(n - 1)  
​  
print(factorial(5))  # 输出: 120

##### 2. 斐波那契数列

斐波那契数列也是一个经典的递归问题。

def fibonacci(n):  
   # 基准条件  
   if n == 0:  
       return 0  
   elif n == 1:  
       return 1  
   # 递归条件  
   else:  
       return fibonacci(n - 1) + fibonacci(n - 2)  
​  
print(fibonacci(10))  # 输出: 55

##### 3. 二分查找

二分查找是一种分治算法，可以通过递归实现。

def binary\_search(arr, low, high, target):  
   # 基准条件  
   if high >= low:  
       mid = (high + low) // 2  
       # 如果目标值在中间  
       if arr[mid] == target:  
           return mid  
       # 如果目标值小于中间值，则在左半部分查找  
       elif arr[mid] > target:  
           return binary\_search(arr, low, mid - 1, target)  
       # 如果目标值大于中间值，则在右半部分查找  
       else:  
           return binary\_search(arr, mid + 1, high, target)  
   else:  
       # 目标值不在数组中  
       return -1  
​  
arr = [2, 3, 4, 10, 40]  
target = 10  
result = binary\_search(arr, 0, len(arr) - 1, target)  
print(result)  # 输出: 3

##### 4. 汉诺塔问题

汉诺塔是一个经典的递归问题，用于演示递归的思想。

def hanoi(n, source, auxiliary, target):  
   if n == 1:  
       print(f"Move disk 1 from {source} to {target}")  
       return  
   hanoi(n - 1, source, target, auxiliary)  
   print(f"Move disk {n} from {source} to {target}")  
   hanoi(n - 1, auxiliary, source, target)  
​  
hanoi(3, 'A', 'B', 'C')

#### 注意事项

* **基准条件**：确保递归函数有明确的基准条件，避免无限递归。
* **递归深度**：递归深度过大可能会导致栈溢出，可以使用尾递归优化或迭代方法来解决。
* **性能**：递归函数的性能可能不如迭代方法，特别是在递归深度较大的情况下。

通过这些例子和解释，递归函数的定义、特点、参数要求、返回值、应用场景以及具体例子都得到了详细的讲解。递归函数是Python中一种强大的工具，能够简洁地解决许多复杂问题。

### 6. 高阶函数

#### 1. map 函数

##### 作用

map函数用于将一个函数应用到一个或多个可迭代对象的每个元素上，返回一个包含结果的迭代器。

##### 用法

map(function, iterable, ...)

##### 参数

* function：一个函数，用于对iterable中的每个元素进行操作。
* iterable：一个或多个可迭代对象（如列表、元组等）。

##### 返回值

返回一个迭代器，包含对iterable中每个元素应用function后的结果。

##### 示例

def square(x):  
   return x \* x  
​  
numbers = [1, 2, 3, 4, 5]  
squared\_numbers = map(square, numbers)  
print(list(squared\_numbers))  # 输出: [1, 4, 9, 16, 25]

#### 2. filter 函数

##### 作用

filter函数用于过滤可迭代对象中的元素，返回一个包含所有使函数返回True的元素的迭代器。

##### 用法

filter(function, iterable)

##### 参数

* function：一个函数，用于判断iterable中的每个元素是否满足条件。如果函数返回True，则保留该元素；否则，丢弃该元素。
* iterable：一个可迭代对象。

##### 返回值

返回一个迭代器，包含所有使function返回True的元素。

##### 示例

def is\_even(x):  
   return x % 2 == 0  
​  
numbers = [1, 2, 3, 4, 5]  
even\_numbers = filter(is\_even, numbers)  
print(list(even\_numbers))  # 输出: [2, 4]

#### 3. reduce 函数

##### 作用

reduce函数用于对一个可迭代对象中的元素进行累积计算，返回一个单一的结果。

##### 用法

reduce(function, iterable[, initializer])

##### 参数

* function：一个函数，接受两个参数，用于累积计算。
* iterable：一个可迭代对象。
* initializer（可选）：初始值，如果提供，则作为累积计算的初始值。

##### 返回值

返回一个单一的累积结果。

##### 示例

from functools import reduce

def add(x, y):

return x + y

numbers = [1, 2, 3, 4, 5]

sum\_of\_numbers = reduce(add, numbers)

print(sum\_of\_numbers) # 输出: 15

# 使用 initializer

sum\_with\_initializer = reduce(add, numbers, 10)

print(sum\_with\_initializer) # 输出: 25

#### 4. sorted 函数

##### 作用

sorted函数用于对一个可迭代对象进行排序，返回一个新的排序后的列表。

##### 用法

sorted(iterable, \*, key=None, reverse=False)

##### 参数

* iterable：一个可迭代对象。
* key（可选）：一个布尔值，如果为True，则按降序排序；默认为False。

##### 返回值

返回一个新的排序后的列表。

##### 示例

def by\_length(word):  
   return len(word)  
​  
words = ["apple", "banana", "cherry", "date"]  
sorted\_words = sorted(words, key=by\_length)  
print(sorted\_words)  # 输出: ['date', 'apple', 'banana', 'cherry']  
​  
# 降序排序  
sorted\_words\_desc = sorted(words, key=by\_length, reverse=True)  
print(sorted\_words\_desc)  # 输出: ['banana', 'cherry', 'apple', 'date']

#### 5. 自定义高阶函数

##### 示例1：接受函数作为参数

def apply\_function(func, value):  
   return func(value)  
​  
def square(x):  
   return x \* x  
​  
result = apply\_function(square, 5)  
print(result)  # 输出: 25

##### 示例2：返回函数

def create\_multiplier(factor):  
   def multiplier(x):  
       return x \* factor  
   return multiplier  
​  
double = create\_multiplier(2)  
triple = create\_multiplier(3)  
​  
print(double(5))  # 输出: 10  
print(triple(5))  # 输出: 15

#### 6. 装饰器

##### 作用

装饰器本质上是一个返回函数的高阶函数，通常用于修改或增强函数的行为。

##### 用法

装饰器通常使用@语法糖来应用。

##### 示例

def my\_decorator(func):  
   def wrapper(\*args, \*\*kwargs):  
       print("Something is happening before the function is called.")  
       result = func(\*args, \*\*kwargs)  
       print("Something is happening after the function is called.")  
       return result  
   return wrapper  
​  
@my\_decorator  
def say\_hello(name):  
   print(f"Hello, {name}!")  
​  
say\_hello("Alice")

输出：

Something is happening before the function is called.  
Hello, Alice!  
Something is happening after the function is called.

通过这些详细的讲解，我们可以全面了解高阶函数的作用、用法、参数要求以及返回值。合理使用高阶函数可以大大提高代码的复用性和可读性。

### 7. 闭包

#### 闭包的工作原理

闭包的核心在于函数可以捕获并记住其定义时的环境。具体来说，当一个嵌套函数引用了其外部函数的变量时，这些变量会被保存在嵌套函数的闭包中。即使外部函数已经执行完毕，这些变量仍然可以被嵌套函数访问和修改。

#### 闭包的构成要素

* **外部函数**：定义了一个局部变量和一个嵌套函数。
* **嵌套函数**：引用了外部函数的局部变量。
* **返回嵌套函数**：外部函数返回嵌套函数，使得嵌套函数可以在其定义的作用域之外被调用。

#### 闭包的具体例子

##### 1. 简单的闭包

def outer\_function(x):  
   def inner\_function(y):  
       return x + y  
   return inner\_function  
​  
closure = outer\_function(10)  
print(closure(5))  # 输出: 15

在这个例子中，outer\_function 返回了 inner\_function，而 inner\_function 记住了 outer\_function 中的变量 x。即使 outer\_function 已经执行完毕，closure 仍然可以访问 x。

##### 2. 带状态的闭包

def make\_counter():  
   count = 0  
   def counter():  
       nonlocal count  
       count += 1  
       return count  
   return counter  
​  
counter1 = make\_counter()  
print(counter1())  # 输出: 1  
print(counter1())  # 输出: 2  
​  
counter2 = make\_counter()  
print(counter2())  # 输出: 1

在这个例子中，make\_counter 返回了一个 counter 函数，这个函数可以记住并更新 count 变量。每次调用 counter 函数时，count 都会增加。

##### 3. 函数工厂

def power\_generator(exp):  
   def power(base):  
       return base \*\* exp  
   return power  
​  
square = power\_generator(2)  
cube = power\_generator(3)  
​  
print(square(4))  # 输出: 16  
print(cube(4))    # 输出: 64

在这个例子中，power\_generator 返回了一个 power 函数，这个函数可以计算一个数的幂。通过不同的 exp 参数，可以创建不同的幂函数。

##### 4. 回调函数

def make\_multiplier(factor):  
   def multiplier(x):  
       return x \* factor  
   return multiplier  
​  
def apply\_callback(callback, value):  
   return callback(value)  
​  
multiplier\_by\_3 = make\_multiplier(3)  
result = apply\_callback(multiplier\_by\_3, 10)  
print(result)  # 输出: 30

在这个例子中，make\_multiplier 返回了一个 multiplier 函数，这个函数可以用来乘以一个特定的因子。apply\_callback 函数接受一个回调函数和一个值，并应用回调函数。

#### 闭包的高级技巧

##### 1. 使用 nonlocal 关键字

在闭包中修改自由变量时，需要使用 nonlocal 关键字。nonlocal 关键字允许嵌套函数修改外部函数的局部变量。

def outer\_function():  
   x = 0  
   def inner\_function():  
       nonlocal x  
       x += 1  
       return x  
   return inner\_function  
​  
closure = outer\_function()  
print(closure())  # 输出: 1  
print(closure())  # 输出: 2

##### 2. 闭包与装饰器

闭包可以与装饰器结合使用，实现一些高级的功能。装饰器本质上是一个返回函数的函数，它可以用来修改或增强其他函数的行为。

def my\_decorator(func):  
   def wrapper(\*args, \*\*kwargs):  
       print("Something is happening before the function is called.")  
       result = func(\*args, \*\*kwargs)  
       print("Something is happening after the function is called.")  
       return result  
   return wrapper  
​  
@my\_decorator  
def say\_hello(name):  
   return f"Hello, {name}!"  
​  
print(say\_hello("Alice"))

在这个例子中，my\_decorator 是一个装饰器，它返回了一个 wrapper 函数，这个函数在调用原函数之前和之后打印一些信息。

##### 3. 闭包与生成器

闭包可以与生成器结合使用，实现一些复杂的逻辑。生成器是一种特殊的函数，它可以暂停和恢复执行，返回一个迭代器。

def make\_generator(start):  
   def generator():  
       nonlocal start  
       while True:  
           yield start  
           start += 1  
   return generator  
​  
gen = make\_generator(0)()  
print(next(gen))  # 输出: 0  
print(next(gen))  # 输出: 1  
print(next(gen))  # 输出: 2

在这个例子中，make\_generator 返回了一个生成器函数，这个生成器函数可以生成一个无限序列。

#### 注意事项

* **自由变量的生命周期**：闭包中的自由变量在其定义的作用域之外仍然存在，因此需要注意内存管理。
* **性能**：闭包可能会导致一些性能问题，特别是在频繁调用的情况下。
* **作用域和命名空间**：闭包中的变量作用域和命名空间需要特别注意，避免变量冲突和意外覆盖。

通过这些详细的解释和例子，闭包的定义、工作原理、应用场景以及高级技巧都得到了深入的探讨。闭包是Python中一个非常强大的工具，能够实现许多高级的编程技巧。

### 8. 偏函数

#### 偏函数的作用

偏函数的主要作用是固定一个函数的部分参数，从而创建一个新的函数。这个新函数在调用时只需要提供剩余的参数即可。偏函数可以简化函数的调用，特别是在需要频繁调用某个函数且部分参数固定的情况下。

#### 偏函数的用法

偏函数通过functools模块中的partial函数来创建。partial函数的基本语法如下：

from functools import partial  
​  
new\_function = partial(original\_function, \*args, \*\*kwargs)

#### 参数要求

* original\_function：需要创建偏函数的原始函数。
* \*args：位置参数，用于固定原始函数的部分位置参数。
* \*\*kwargs：关键字参数，用于固定原始函数的部分关键字参数。

#### 返回值

partial函数返回一个新的函数（偏函数），这个新函数在调用时只需要提供剩余的参数即可。

#### 详细用法示例

##### 1. 固定位置参数

from functools import partial  
​  
def add(a, b):  
   return a + b  
​  
# 创建一个偏函数，固定第一个位置参数为 5  
add\_5 = partial(add, 5)  
​  
print(add\_5(3))  # 输出: 8

在这个例子中，add\_5是一个新的函数，它固定了add函数的第一个位置参数为5。调用add\_5时只需要提供第二个参数即可。

##### 2. 固定关键字参数

from functools import partial  
​  
def multiply(a, b):  
   return a \* b  
​  
# 创建一个偏函数，固定关键字参数 b 为 3  
multiply\_by\_3 = partial(multiply, b=3)  
​  
print(multiply\_by\_3(4))  # 输出: 12

在这个例子中，multiply\_by\_3是一个新的函数，它固定了multiply函数的关键字参数b为3。调用multiply\_by\_3时只需要提供参数a即可。

##### 3. 混合固定位置参数和关键字参数

from functools import partial  
​  
def power(base, exp):  
   return base \*\* exp  
​  
# 创建一个偏函数，固定位置参数 base 为 2，关键字参数 exp 为 3  
power\_2\_3 = partial(power, 2, exp=3)  
​  
print(power\_2\_3())  # 输出: 8

在这个例子中，power\_2\_3是一个新的函数，它固定了power函数的位置参数base为2，关键字参数exp为3。调用power\_2\_3时不需要提供任何参数。

##### 4. 结合类方法

from functools import partial  
​  
class Calculator:  
   def add(self, a, b):  
       return a + b  
​  
# 创建一个偏函数，固定类方法的部分参数  
calc = Calculator()  
add\_5 = partial(calc.add, 5)  
​  
print(add\_5(3))  # 输出: 8

在这个例子中，add\_5是一个新的函数，它固定了Calculator类实例的add方法的第一个位置参数为5。调用add\_5时只需要提供第二个参数即可。

##### 5. 结合装饰器

from functools import partial, wraps  
​  
def log\_decorator(func):  
   @wraps(func)  
   def wrapper(\*args, \*\*kwargs):  
       print(f"Calling {func.\_\_name\_\_} with args: {args}, kwargs: {kwargs}")  
       result = func(\*args, \*\*kwargs)  
       print(f"{func.\_\_name\_\_} returned: {result}")  
       return result  
   return wrapper  
​  
def add(a, b):  
   return a + b  
​  
# 创建一个偏函数，并应用装饰器  
add\_5 = partial(add, 5)  
add\_5 = log\_decorator(add\_5)  
​  
print(add\_5(3))  # 输出: 8

在这个例子中，我们首先创建了一个偏函数add\_5，然后使用装饰器log\_decorator来装饰这个偏函数，从而在调用时打印日志信息。

#### 注意事项

* **参数顺序**：使用偏函数时需要注意参数的顺序，特别是位置参数和关键字参数的混合使用。
* **性能**：偏函数可能会导致一些性能问题，特别是在频繁调用的情况下。
* **作用域和命名空间**：偏函数中的变量作用域和命名空间需要特别注意，避免变量冲突和意外覆盖。

通过这些详细的解释和例子，偏函数的定义、工作原理、应用场景以及高级技巧都得到了深入的探讨。偏函数是Python中一个非常强大的工具，能够简化函数的调用，特别是在需要固定部分参数的情况下。

# Python类和对象

## 1. 类的定义和基本用法

在 Python 中，类是创建对象的蓝图。类定义了对象的属性和方法。

class Dog:  
   def \_\_init\_\_(self, name, age):  
       self.name = name  
       self.age = age  
​  
   def bark(self):  
       return f"{self.name} is barking!"  
​  
dog1 = Dog("Buddy", 3)  
print(dog1.bark())  # 输出: Buddy is barking!

## 2. 类变量和实例变量

类变量是所有实例共享的变量，而实例变量是每个实例独有的。

class Dog:  
   species = "Canis familiaris"  # 类变量  
​  
   def \_\_init\_\_(self, name, age):  
       self.name = name  # 实例变量  
       self.age = age    # 实例变量  
​  
dog1 = Dog("Buddy", 3)  
dog2 = Dog("Max", 2)  
print(dog1.species)  # 输出: Canis familiaris  
print(dog2.species)  # 输出: Canis familiaris

## 3. 类方法和静态方法

类方法使用 @classmethod 装饰器，静态方法使用 @staticmethod 装饰器。

class Dog:  
   species = "Canis familiaris"  
​  
   def \_\_init\_\_(self, name, age):  
       self.name = name  
       self.age = age  
​  
   @classmethod  
   def from\_birth\_year(cls, name, birth\_year):  
       return cls(name, 2023 - birth\_year)  
​  
   @staticmethod  
   def is\_adult(age):  
       return age > 1  
​  
dog1 = Dog.from\_birth\_year("Buddy", 2020)  
print(dog1.age)  # 输出: 3  
print(Dog.is\_adult(3))  # 输出: True

## 4. 私有属性和方法

私有属性以双下划线 \_\_ 开头，只能在类内部访问。

class Dog:  
   def \_\_init\_\_(self, name, age):  
       self.\_\_name = name  # 私有属性  
       self.\_\_age = age    # 私有属性  
​  
   def get\_name(self):  
       return self.\_\_name  
​  
   def get\_age(self):  
       return self.\_\_age  
​  
dog1 = Dog("Buddy", 3)  
print(dog1.get\_name())  # 输出: Buddy  
print(dog1.get\_age())   # 输出: 3

## 5. 多重继承

多重继承允许一个类继承多个父类的属性和方法。

class Animal:  
   def eat(self):  
       return "eating"  
​  
class Mammal(Animal):  
   def breathe(self):  
       return "breathing"  
​  
class Dog(Mammal):  
   def bark(self):  
       return "barking"  
​  
dog1 = Dog()  
print(dog1.eat())    # 输出: eating  
print(dog1.breathe())  # 输出: breathing  
print(dog1.bark())   # 输出: barking

## 6. 抽象基类

抽象基类使用 abc 模块定义，强制子类实现特定方法。

from abc import ABC, abstractmethod  
​  
class Animal(ABC):  
   @abstractmethod  
   def make\_sound(self):  
       pass  
​  
class Dog(Animal):  
   def make\_sound(self):  
       return "Bark"  
​  
dog1 = Dog()  
print(dog1.make\_sound())  # 输出: Bark

## 7. 元类

元类是类的类，用于创建类。

class MyMeta(type):  
   def \_\_new\_\_(cls, name, bases, dct):  
       print(f"Creating class {name}")  
       return super().\_\_new\_\_(cls, name, bases, dct)  
​  
class MyClass(metaclass=MyMeta):  
   pass  
​  
# 输出: Creating class MyClass

## 8. \_\_str\_\_ 和 \_\_repr\_\_ 方法

在Python中，\_\_str\_\_ 和 \_\_repr\_\_ 是两个特殊的方法，用于定义对象的字符串表示形式。它们在不同的场景下被调用，具有不同的用途。下面是对这两个方法的详细解释，包括它们的语法格式、作用、基本用法和高级用法。

### \_\_str\_\_ 方法

#### 语法格式

def \_\_str\_\_(self):  
   # 方法体  
   return string

#### 作用

\_\_str\_\_ 方法用于定义对象的可读字符串表示形式。它通常在以下情况下被调用：

* 使用 str(obj) 或 print(obj) 时。
* 使用格式化字符串 f"{obj}" 时。

#### 参数要求

* self: 当前对象。

#### 返回值

\_\_str\_\_ 方法必须返回一个字符串。

#### 基本用法

class Person:  
   def \_\_init\_\_(self, name, age):  
       self.name = name  
       self.age = age  
​  
   def \_\_str\_\_(self):  
       return f"Person(name={self.name}, age={self.age})"  
​  
person = Person("Alice", 30)  
print(person)  # 输出: Person(name=Alice, age=30)

#### 高级用法

\_\_str\_\_ 方法可以用于提供对象的用户友好表示，特别是在调试和日志记录时。

### \_\_repr\_\_ 方法

#### 语法格式

def \_\_repr\_\_(self):  
   # 方法体  
   return string

#### 作用

\_\_repr\_\_ 方法用于定义对象的“官方”字符串表示形式。它通常在以下情况下被调用：

* 使用 repr(obj) 时。
* 在交互式解释器中直接输入对象名时。
* 如果没有定义 \_\_str\_\_ 方法，print(obj) 也会调用 \_\_repr\_\_。

#### 参数要求

* self: 当前对象。

#### 返回值

\_\_repr\_\_ 方法必须返回一个字符串。

#### 基本用法

class Person:  
   def \_\_init\_\_(self, name, age):  
       self.name = name  
       self.age = age  
​  
   def \_\_repr\_\_(self):  
       return f"Person(name={self.name}, age={self.age})"  
​  
person = Person("Alice", 30)  
print(repr(person))  # 输出: Person(name=Alice, age=30)

#### 高级用法

\_\_repr\_\_ 方法可以用于提供对象的详细和明确的表示，特别是在调试和开发时。一个好的 \_\_repr\_\_ 方法应该能够重新创建对象，即 eval(repr(obj)) 应该能够生成一个等价的对象。

### 示例代码

class Person:  
   def \_\_init\_\_(self, name, age):  
       self.name = name  
       self.age = age  
​  
   def \_\_str\_\_(self):  
       return f"Person(name={self.name}, age={self.age})"  
​  
   def \_\_repr\_\_(self):  
       return f"Person(name={self.name!r}, age={self.age!r})"  
​  
person = Person("Alice", 30)  
print(str(person))  # 输出: Person(name=Alice, age=30)  
print(repr(person))  # 输出: Person(name='Alice', age=30)

### 总结

* \_\_str\_\_ 方法用于提供对象的用户友好字符串表示，通常在打印和格式化字符串时使用。
* \_\_repr\_\_ 方法用于提供对象的详细和明确的字符串表示，通常在调试和开发时使用。
* 如果没有定义 \_\_str\_\_ 方法，print(obj) 会调用 \_\_repr\_\_ 方法。
* \_\_str\_\_ 和 \_\_repr\_\_ 方法都必须返回一个字符串。

## 9. \_\_eq\_\_ 和 \_\_hash\_\_ 方法

在Python中，\_\_eq\_\_ 和 \_\_hash\_\_ 方法是用于定义对象的相等性和哈希值的特殊方法。它们在不同的场景中被调用，具有不同的用途和行为。下面是对这两个方法的详细解释，包括它们的语法格式、作用、基本用法和高级用法。

### \_\_eq\_\_ 方法

#### 语法格式

def \_\_eq\_\_(self, other):  
   # 方法体  
   return boolean

#### 作用

\_\_eq\_\_ 方法用于定义对象之间的相等性比较。它决定了两个对象是否被认为是相等的。

#### 参数要求

* self: 当前对象。
* other: 要比较的另一个对象。

#### 返回值

返回一个布尔值，表示两个对象是否相等。

#### 基本用法

class Person:  
   def \_\_init\_\_(self, name, age):  
       self.name = name  
       self.age = age  
​  
   def \_\_eq\_\_(self, other):  
       if isinstance(other, Person):  
           return self.name == other.name and self.age == other.age  
       return False  
​  
person1 = Person("Alice", 30)  
person2 = Person("Alice", 30)  
person3 = Person("Bob", 30)  
​  
print(person1 == person2)  # 输出: True  
print(person1 == person3)  # 输出: False

#### 高级用法

\_\_eq\_\_ 方法可以用于自定义对象在集合（如字典、集合）中的相等性比较。

### \_\_hash\_\_ 方法

#### 语法格式

def \_\_hash\_\_(self):  
   # 方法体  
   return integer

#### 作用

\_\_hash\_\_ 方法用于定义对象的哈希值。哈希值在集合（如字典、集合）中用于快速查找和比较对象。

#### 参数要求

* self: 当前对象。

#### 返回值

返回一个整数，表示对象的哈希值。

#### 基本用法

class Person:  
   def \_\_init\_\_(self, name, age):  
       self.name = name  
       self.age = age  
​  
   def \_\_eq\_\_(self, other):  
       if isinstance(other, Person):  
           return self.name == other.name and self.age == other.age  
       return False  
​  
   def \_\_hash\_\_(self):  
       return hash((self.name, self.age))  
​  
person1 = Person("Alice", 30)  
person2 = Person("Alice", 30)  
person3 = Person("Bob", 30)  
​  
print(hash(person1))  # 输出: 某个整数  
print(person1 == person2)  # 输出: True  
print(person1 == person3)  # 输出: False  
​  
# 使用集合  
people = {person1, person2, person3}  
print(len(people))  # 输出: 2

#### 高级用法

\_\_hash\_\_ 方法可以用于自定义对象在集合中的哈希值计算，确保对象在集合中的唯一性。

### 注意事项

* 如果一个类定义了 \_\_eq\_\_ 方法但没有定义 \_\_hash\_\_ 方法，那么该类的实例将不可哈希（即不能用作字典的键或集合的元素）。
* 如果一个类定义了 \_\_hash\_\_ 方法，那么它也应该定义 \_\_eq\_\_ 方法，以确保一致性。
* 如果一个类的实例是可变的，通常不应该定义 \_\_hash\_\_ 方法，因为哈希值应该是不变的。

### 示例代码

class Person:  
   def \_\_init\_\_(self, name, age):  
       self.name = name  
       self.age = age  
​  
   def \_\_eq\_\_(self, other):  
       if isinstance(other, Person):  
           return self.name == other.name and self.age == other.age  
       return False  
​  
   def \_\_hash\_\_(self):  
       return hash((self.name, self.age))  
​  
person1 = Person("Alice", 30)  
person2 = Person("Alice", 30)  
person3 = Person("Bob", 30)  
​  
print(person1 == person2)  # 输出: True  
print(person1 == person3)  # 输出: False  
print(hash(person1))  # 输出: 某个整数  
​  
# 使用集合  
people = {person1, person2, person3}  
print(len(people))  # 输出: 2

### 总结

* \_\_eq\_\_ 方法用于定义对象之间的相等性比较。
* \_\_hash\_\_ 方法用于定义对象的哈希值，确保对象在集合中的唯一性。
* 如果定义了 \_\_eq\_\_ 方法，通常也应该定义 \_\_hash\_\_ 方法。
* 如果对象是可变的，通常不应该定义 \_\_hash\_\_ 方法。

## 10. \_\_del\_\_ 方法

在Python中，\_\_del\_\_ 方法是一个特殊的方法，用于定义对象在被垃圾回收时的行为。它通常被称为析构方法（destructor）。下面是对 \_\_del\_\_ 方法的详细解释，包括它的语法格式、作用、基本用法和高级用法。

### \_\_del\_\_ 方法

#### 语法格式

def \_\_del\_\_(self):  
   # 方法体

#### 作用

\_\_del\_\_ 方法在对象被垃圾回收时调用。它可以用于执行一些清理工作，例如关闭文件、释放资源等。

#### 参数要求

* self: 当前对象。

#### 返回值

\_\_del\_\_ 方法没有返回值。

#### 基本用法

class MyClass:  
   def \_\_init\_\_(self, name):  
       self.name = name  
       print(f"{self.name} 对象已创建")  
​  
   def \_\_del\_\_(self):  
       print(f"{self.name} 对象将被销毁")  
​  
obj1 = MyClass("Object1")  
obj2 = MyClass("Object2")  
​  
# 删除对象引用  
del obj1  
del obj2

#### 高级用法

\_\_del\_\_ 方法可以用于释放资源，例如关闭文件或网络连接。然而，由于Python的垃圾回收机制是不确定的，\_\_del\_\_ 方法的调用时间也是不确定的。因此，不建议依赖 \_\_del\_\_ 方法进行关键的资源管理。

### 注意事项

* \_\_del\_\_ 方法的调用时间是不确定的，因为它依赖于Python的垃圾回收机制。
* 不要在 \_\_del\_\_ 方法中进行复杂的操作，因为这可能会导致未定义的行为。
* 如果对象之间存在循环引用，\_\_del\_\_ 方法可能不会被调用，除非显式地打破循环引用。

### 示例代码

class FileHandler:  
   def \_\_init\_\_(self, filename):  
       self.filename = filename  
       self.file = open(filename, 'w')  
       print(f"文件 {self.filename} 已打开")  
​  
   def write(self, data):  
       self.file.write(data)  
​  
   def \_\_del\_\_(self):  
       self.file.close()  
       print(f"文件 {self.filename} 已关闭")  
​  
file1 = FileHandler("test1.txt")  
file1.write("Hello, World!")  
​  
file2 = FileHandler("test2.txt")  
file2.write("This is a test.")  
​  
# 删除对象引用  
del file1  
del file2

### 总结

* \_\_del\_\_ 方法在对象被垃圾回收时调用，用于执行一些清理工作。
* \_\_del\_\_ 方法没有返回值。
* \_\_del\_\_ 方法的调用时间是不确定的，因此不建议依赖它进行关键的资源管理。
* 如果对象之间存在循环引用，\_\_del\_\_ 方法可能不会被调用，除非显式地打破循环引用。

## 11. \_\_call\_\_ 方法

### \_\_call\_\_ 方法

#### 语法格式

class MyClass:  
   def \_\_call\_\_(self, \*args, \*\*kwargs):  
       # 方法体  
       pass

class MyClass: 定义一个类。

def \_\_call\_\_(self, \*args, \*\*kwargs):定义 \_\_call\_\_方法。

* self: 类的实例本身。
* \*args: 可变位置参数，可以接收任意数量的位置参数。
* \*\*kwargs: 可变关键字参数，可以接收任意数量的关键字参数。

#### 作用

\_\_call\_\_ 方法的主要作用是让类的实例具有函数的行为。这意味着你可以像调用函数一样调用类的实例，并传递参数给它。

#### 参数要求

* self: 类的实例本身。
* \*args: 可变位置参数，可以接收任意数量的位置参数。
* \*\*kwargs: 可变关键字参数，可以接收任意数量的关键字参数。

#### 返回值

\_\_call\_\_ 方法可以返回任何类型的值，具体取决于方法内部的实现。

### 基本用法

class Adder:  
   def \_\_init\_\_(self, base):  
       self.base = base  
​  
   def \_\_call\_\_(self, x):  
       return self.base + x  
​  
# 创建实例  
adder = Adder(10)  
​  
# 像函数一样调用实例  
result = adder(5)  # 结果为 15  
print(result)

在这个例子中，Adder 类的实例 adder 可以像函数一样被调用，并返回 base 和 x 的和。

### 高级用法

#### 实现装饰器

class Timer:  
   def \_\_init\_\_(self, func):  
       self.func = func  
​  
   def \_\_call\_\_(self, \*args, \*\*kwargs):  
       import time  
       start\_time = time.time()  
       result = self.func(\*args, \*\*kwargs)  
       end\_time = time.time()  
       print(f"Function {self.func.\_\_name\_\_} took {end\_time - start\_time} seconds to run.")  
       return result  
​  
@Timer  
def slow\_function():  
   import time  
   time.sleep(2)  
​  
slow\_function()

在这个例子中，Timer 类用作装饰器，\_\_call\_\_ 方法用于在调用被装饰函数时记录运行时间。

* \_\_init\_\_(self, func): 初始化方法，接收一个函数 func 作为参数。
* \_\_call\_\_(self, \*args, \*\*kwargs): 在调用被装饰函数时执行，记录运行时间并返回结果。

#### 实现回调函数

class Callback:  
   def \_\_init\_\_(self, callback\_func):  
       self.callback\_func = callback\_func  
​  
   def \_\_call\_\_(self, \*args, \*\*kwargs):  
       return self.callback\_func(\*args, \*\*kwargs)  
​  
def my\_callback(x):  
   return x \* 2  
​  
callback\_instance = Callback(my\_callback)  
result = callback\_instance(5)  # 结果为 10  
print(result)

在这个例子中，Callback 类的实例 callback\_instance 可以像函数一样被调用，并执行回调函数 my\_callback。

* \_\_init\_\_(self, callback\_func): 初始化方法，接收一个回调函数 callback\_func 作为参数。
* \_\_call\_\_(self, \*args, \*\*kwargs): 在调用实例时执行回调函数，并返回结果。

### 其他高级用法

#### 实现可调用的状态机

class StateMachine:  
   def \_\_init\_\_(self):  
       self.state = 'INIT'  
​  
   def \_\_call\_\_(self, event):  
       if self.state == 'INIT' and event == 'start':  
           self.state = 'RUNNING'  
           return "Machine started"  
       elif self.state == 'RUNNING' and event == 'stop':  
           self.state = 'STOPPED'  
           return "Machine stopped"  
       else:  
           return "Invalid event for current state"  
​  
machine = StateMachine()  
print(machine('start'))  # 输出: Machine started  
print(machine('stop'))   # 输出: Machine stopped

在这个例子中，StateMachine 类的实例 machine 可以根据当前状态和事件进行状态转换。

* \_\_init\_\_(self): 初始化方法，设置初始状态。
* \_\_call\_\_(self, event): 根据当前状态和事件进行状态转换，并返回相应消息。

### 总结

\_\_call\_\_ 方法使得类的实例可以像函数一样被调用，这在许多高级编程场景中非常有用，例如实现装饰器、回调函数、状态机等。通过定义 \_\_call\_\_ 方法，你可以灵活地控制实例在被调用时的行为。

## 12. \_\_getitem\_\_ 和 \_\_setitem\_\_ 方法

在Python中，\_\_getitem\_\_ 和 \_\_setitem\_\_ 是两个特殊的方法，用于实现对象的索引操作，即通过索引访问和修改对象的元素。下面是对这两个方法的详细解释，包括它们的语法格式、作用、基本用法和高级用法。

### \_\_getitem\_\_ 方法

#### 语法格式

def \_\_getitem\_\_(self, key):  
   # 方法体  
   return value

#### 作用

\_\_getitem\_\_ 方法用于定义对象的索引访问操作。它通常在以下情况下被调用：

* 使用 obj[key] 时。

#### 参数要求

* self: 当前对象。
* key: 索引键，可以是任何可哈希的对象（如整数、字符串等）。

#### 返回值

\_\_getitem\_\_ 方法必须返回一个值，该值是索引键对应的元素。

#### 基本用法

class MyList:  
   def \_\_init\_\_(self, data):  
       self.data = data  
​  
   def \_\_getitem\_\_(self, key):  
       return self.data[key]  
​  
my\_list = MyList([1, 2, 3, 4, 5])  
print(my\_list[2])  # 输出: 3

#### 高级用法

\_\_getitem\_\_ 方法可以用于实现复杂的索引操作，例如支持切片、多维索引等。

class MyList:  
   def \_\_init\_\_(self, data):  
       self.data = data  
​  
   def \_\_getitem\_\_(self, key):  
       if isinstance(key, slice):  
           return self.data[key.start:key.stop:key.step]  
       elif isinstance(key, tuple):  
           return [self.data[k] for k in key]  
       else:  
           return self.data[key]  
​  
my\_list = MyList([1, 2, 3, 4, 5])  
print(my\_list[1:3])  # 输出: [2, 3]  
print(my\_list[(0, 2, 4)])  # 输出: [1, 3, 5]

### \_\_setitem\_\_ 方法

#### 语法格式

def \_\_setitem\_\_(self, key, value):  
   # 方法体

#### 作用

\_\_setitem\_\_ 方法用于定义对象的索引赋值操作。它通常在以下情况下被调用：

* 使用 obj[key] = value 时。

#### 参数要求

* self: 当前对象。
* key: 索引键，可以是任何可哈希的对象（如整数、字符串等）。
* value: 要赋给索引键的值。

#### 返回值

\_\_setitem\_\_ 方法不返回任何值，它只负责修改对象的状态。

#### 基本用法

class MyList:  
   def \_\_init\_\_(self, data):  
       self.data = data  
​  
   def \_\_setitem\_\_(self, key, value):  
       self.data[key] = value  
​  
my\_list = MyList([1, 2, 3, 4, 5])  
my\_list[2] = 10  
print(my\_list.data)  # 输出: [1, 2, 10, 4, 5]

#### 高级用法

\_\_setitem\_\_ 方法可以用于实现复杂的索引赋值操作，例如支持切片赋值、多维索引赋值等。

class MyList:  
   def \_\_init\_\_(self, data):  
       self.data = data  
​  
   def \_\_setitem\_\_(self, key, value):  
       if isinstance(key, slice):  
           self.data[key.start:key.stop:key.step] = value  
       elif isinstance(key, tuple):  
           for k, v in zip(key, value):  
               self.data[k] = v  
       else:  
           self.data[key] = value  
​  
my\_list = MyList([1, 2, 3, 4, 5])  
my\_list[1:3] = [20, 30]  
print(my\_list.data)  # 输出: [1, 20, 30, 4, 5]  
my\_list[(0, 2, 4)] = [10, 30, 50]  
print(my\_list.data)  # 输出: [10, 20, 30, 4, 50]

### 示例代码

class MyList:  
   def \_\_init\_\_(self, data):  
       self.data = data  
​  
   def \_\_getitem\_\_(self, key):  
       if isinstance(key, slice):  
           return self.data[key.start:key.stop:key.step]  
       elif isinstance(key, tuple):  
           return [self.data[k] for k in key]  
       else:  
           return self.data[key]  
​  
   def \_\_setitem\_\_(self, key, value):  
       if isinstance(key, slice):  
           self.data[key.start:key.stop:key.step] = value  
       elif isinstance(key, tuple):  
           for k, v in zip(key, value):  
               self.data[k] = v  
       else:  
           self.data[key] = value  
​  
my\_list = MyList([1, 2, 3, 4, 5])  
print(my\_list[2])  # 输出: 3  
my\_list[2] = 10  
print(my\_list.data)  # 输出: [1, 2, 10, 4, 5]  
print(my\_list[1:3])  # 输出: [2, 10]  
my\_list[1:3] = [20, 30]  
print(my\_list.data)  # 输出: [1, 20, 30, 4, 5]  
print(my\_list[(0, 2, 4)])  # 输出: [1, 30, 5]  
my\_list[(0, 2, 4)] = [10, 30, 50]  
print(my\_list.data)  # 输出: [10, 20, 30, 4, 50]

### 总结

* \_\_getitem\_\_ 方法用于定义对象的索引访问操作，返回索引键对应的元素。
* \_\_setitem\_\_ 方法用于定义对象的索引赋值操作，修改索引键对应的元素。
* 这两个方法可以支持复杂的索引操作，例如切片、多维索引等。

## 13. \_\_iter\_\_ 和 \_\_next\_\_ 方法

在Python中，\_\_iter\_\_ 和 \_\_next\_\_ 是两个特殊的方法，用于实现对象的迭代器协议。通过这两个方法，可以使得自定义的对象能够被用于 for 循环和其他迭代上下文中。下面是对这两个方法的详细解释，包括它们的语法格式、作用、基本用法和高级用法。

### \_\_iter\_\_ 方法

#### 语法格式

def \_\_iter\_\_(self):  
   # 方法体  
   return iterator

#### 作用

\_\_iter\_\_ 方法用于返回一个迭代器对象。这个迭代器对象必须实现 \_\_next\_\_ 方法，以便能够进行迭代操作。通常在以下情况下被调用：

* 使用 for 循环迭代对象时。
* 使用内置函数 iter() 时。

#### 参数要求

* self: 当前对象。

#### 返回值

\_\_iter\_\_ 方法必须返回一个迭代器对象，该对象实现了 \_\_next\_\_ 方法。

#### 基本用法

class MyIterator:  
   def \_\_init\_\_(self, data):  
       self.data = data  
       self.index = 0  
​  
   def \_\_iter\_\_(self):  
       return self  
​  
   def \_\_next\_\_(self):  
       if self.index >= len(self.data):  
           raise StopIteration  
       value = self.data[self.index]  
       self.index += 1  
       return value  
​  
my\_iter = MyIterator([1, 2, 3, 4, 5])  
for item in my\_iter:  
   print(item)  # 输出: 1 2 3 4 5

#### 高级用法

\_\_iter\_\_ 方法可以返回不同的迭代器对象，以实现复杂的迭代逻辑。

class MyIterable:  
   def \_\_init\_\_(self, data):  
       self.data = data  
​  
   def \_\_iter\_\_(self):  
       return MyIterator(self.data)  
​  
class MyIterator:  
   def \_\_init\_\_(self, data):  
       self.data = data  
       self.index = 0  
​  
   def \_\_iter\_\_(self):  
       return self  
​  
   def \_\_next\_\_(self):  
       if self.index >= len(self.data):  
           raise StopIteration  
       value = self.data[self.index]  
       self.index += 1  
       return value  
​  
my\_iterable = MyIterable([1, 2, 3, 4, 5])  
for item in my\_iterable:  
   print(item)  # 输出: 1 2 3 4 5

### \_\_next\_\_ 方法

#### 语法格式

def \_\_next\_\_(self):  
   # 方法体  
   return value

#### 作用

\_\_next\_\_ 方法用于返回迭代器的下一个元素。如果没有更多的元素可供返回，则必须引发 StopIteration 异常。通常在以下情况下被调用：

* 使用 for 循环迭代对象时。
* 使用内置函数 next() 时。

#### 参数要求

* self: 当前迭代器对象。

#### 返回值

\_\_next\_\_ 方法必须返回迭代器的下一个元素。如果没有更多的元素可供返回，则必须引发 StopIteration 异常。

#### 基本用法

class MyIterator:  
   def \_\_init\_\_(self, data):  
       self.data = data  
       self.index = 0  
​  
   def \_\_iter\_\_(self):  
       return self  
​  
   def \_\_next\_\_(self):  
       if self.index >= len(self.data):  
           raise StopIteration  
       value = self.data[self.index]  
       self.index += 1  
       return value  
​  
my\_iter = MyIterator([1, 2, 3, 4, 5])  
print(next(my\_iter))  # 输出: 1  
print(next(my\_iter))  # 输出: 2

#### 高级用法

\_\_next\_\_ 方法可以实现复杂的迭代逻辑，例如条件过滤、元素转换等。

class MyIterator:  
   def \_\_init\_\_(self, data):  
       self.data = data  
       self.index = 0  
​  
   def \_\_iter\_\_(self):  
       return self  
​  
   def \_\_next\_\_(self):  
       while self.index < len(self.data):  
           value = self.data[self.index]  
           self.index += 1  
           if value % 2 == 0:  # 只返回偶数  
               return value  
       raise StopIteration  
​  
my\_iter = MyIterator([1, 2, 3, 4, 5])  
for item in my\_iter:  
   print(item)  # 输出: 2 4

### 示例代码

class MyIterable:  
   def \_\_init\_\_(self, data):  
       self.data = data  
​  
   def \_\_iter\_\_(self):  
       return MyIterator(self.data)  
​  
class MyIterator:  
   def \_\_init\_\_(self, data):  
       self.data = data  
       self.index = 0  
​  
   def \_\_iter\_\_(self):  
       return self  
​  
   def \_\_next\_\_(self):  
       if self.index >= len(self.data):  
           raise StopIteration  
       value = self.data[self.index]  
       self.index += 1  
       return value  
​  
my\_iterable = MyIterable([1, 2, 3, 4, 5])  
for item in my\_iterable:  
   print(item)  # 输出: 1 2 3 4 5  
​  
my\_iter = MyIterator([1, 2, 3, 4, 5])  
print(next(my\_iter))  # 输出: 1  
print(next(my\_iter))  # 输出: 2

### 总结

* \_\_iter\_\_ 方法用于返回一个迭代器对象，该对象必须实现 \_\_next\_\_ 方法。
* \_\_next\_\_ 方法用于返回迭代器的下一个元素，如果没有更多的元素可供返回，则必须引发 StopIteration 异常。
* 这两个方法使得自定义的对象能够被用于 for 循环和其他迭代上下文中。
* 可以通过实现复杂的逻辑来定制迭代行为，例如条件过滤、元素转换等。

## 14. \_\_enter\_\_ 和 \_\_exit\_\_ 方法

在Python中，\_\_enter\_\_ 和 \_\_exit\_\_ 是两个特殊的方法，用于实现上下文管理协议（Context Management Protocol）。通过这两个方法，可以使得自定义的对象能够被用于 with 语句中，从而实现资源的自动管理。下面是对这两个方法的详细解释，包括它们的语法格式、作用、基本用法和高级用法。

### \_\_enter\_\_ 方法

#### 语法格式

def \_\_enter\_\_(self):  
   # 方法体  
   return value

#### 作用

\_\_enter\_\_ 方法在 with 语句开始时被调用，用于设置上下文环境。通常用于资源的初始化或准备工作。

#### 参数要求

* self: 当前对象。

#### 返回值

\_\_enter\_\_ 方法可以返回一个对象，该对象将被绑定到 with 语句中的 as 子句（如果有的话）。

#### 基本用法

class MyContextManager:  
   def \_\_enter\_\_(self):  
       print("Entering the context")  
       return self  
​  
   def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
       print("Exiting the context")  
​  
with MyContextManager() as cm:  
   print("Inside the context")

输出：

Entering the context  
Inside the context  
Exiting the context

#### 高级用法

\_\_enter\_\_ 方法可以返回任意对象，这些对象可以在 with 语句块中使用。

class FileManager:  
   def \_\_init\_\_(self, filename, mode):  
       self.filename = filename  
       self.mode = mode  
       self.file = None  
​  
   def \_\_enter\_\_(self):  
       self.file = open(self.filename, self.mode)  
       return self.file  
​  
   def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
       if self.file:  
           self.file.close()  
​  
with FileManager('test.txt', 'w') as f:  
   f.write('Hello, World!')

### \_\_exit\_\_ 方法

#### 语法格式

def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
   # 方法体  
   return boolean

#### 作用

\_\_exit\_\_ 方法在 with 语句结束时被调用，用于清理上下文环境。通常用于资源的释放或清理工作。

#### 参数要求

* self: 当前对象。
* exc\_type: 异常类型（如果 with 语句块中发生异常）。
* exc\_value: 异常实例（如果 with 语句块中发生异常）。
* traceback: 异常的追踪信息（如果 with 语句块中发生异常）。

#### 返回值

\_\_exit\_\_ 方法可以返回一个布尔值，用于指示是否要抑制异常。如果返回 True，则异常被抑制；如果返回 False，则异常继续传播。

#### 基本用法

class MyContextManager:  
   def \_\_enter\_\_(self):  
       print("Entering the context")  
       return self  
​  
   def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
       print("Exiting the context")  
       if exc\_type is not None:  
           print(f"Exception occurred: {exc\_type}, {exc\_value}")  
       return False  
​  
with MyContextManager() as cm:  
   print("Inside the context")  
   raise ValueError("An error occurred")

输出：

Entering the context  
Inside the context  
Exiting the context  
Exception occurred: <class 'ValueError'>, An error occurred  
Traceback (most recent call last):  
...  
ValueError: An error occurred

#### 高级用法

\_\_exit\_\_ 方法可以处理异常并决定是否抑制异常。

class FileManager:  
   def \_\_init\_\_(self, filename, mode):  
       self.filename = filename  
       self.mode = mode  
       self.file = None  
​  
   def \_\_enter\_\_(self):  
       self.file = open(self.filename, self.mode)  
       return self.file  
​  
   def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
       if self.file:  
           self.file.close()  
       if exc\_type is not None:  
           print(f"Exception occurred: {exc\_type}, {exc\_value}")  
       return True  # 抑制异常  
​  
with FileManager('test.txt', 'w') as f:  
   f.write('Hello, World!')  
   raise ValueError("An error occurred")

输出：

Exception occurred: <class 'ValueError'>, An error occurred

### 示例代码

class MyContextManager:

def \_\_enter\_\_(self):

print("Entering the context")

return self

def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):

print("Exiting the context")

if exc\_type is not None:

print(f"Exception occurred: {exc\_type}, {exc\_value}")

return False

with MyContextManager() as cm:

print("Inside the context")

raise ValueError("An error occurred")

输出：

Entering the context

Inside the context

Exiting the context

Exception occurred: <class 'ValueError'>, An error occurred

Traceback (most recent call last):

...

ValueError: An error occurred

### 总结

* \_\_enter\_\_ 方法在 with 语句开始时被调用，用于设置上下文环境，可以返回一个对象绑定到 as 子句。
* \_\_exit\_\_ 方法在 with 语句结束时被调用，用于清理上下文环境，可以处理异常并决定是否抑制异常。
* 这两个方法使得自定义的对象能够被用于 with 语句中，从而实现资源的自动管理。
* 可以通过处理异常和返回布尔值来定制上下文管理的行为。

## 15. super 函数

在Python中，super 函数是一个非常重要的工具，用于调用父类或兄弟类的方法。它主要用于多重继承和类的层次结构中，以确保方法调用的正确性和灵活性。下面是对 super 函数的详细解释，包括其语法格式、作用、基本用法和高级用法。

### 语法格式

#### 基本语法

super()

#### 带参数的语法

super(type, obj)

#### 带参数的语法（用于类方法）

super(type, type2)

### 作用

super 函数的主要作用是在子类中调用父类的方法，特别是在多重继承的情况下，确保方法解析顺序（MRO）的正确性。它可以帮助避免直接使用父类名称带来的硬编码问题，使得代码更加灵活和可维护。

### 参数要求

* type: 类类型，通常是当前类。
* obj: 实例对象，通常是当前实例。
* type2: 另一个类类型，通常是当前类。

### 返回值

super 函数返回一个临时对象，该对象可以调用父类或兄弟类的方法。

### 基本用法

#### 单继承中的基本用法

在单继承中，super 函数通常用于调用父类的构造函数或其他方法。

class Parent:  
   def \_\_init\_\_(self, name):  
       self.name = name  
​  
class Child(Parent):  
   def \_\_init\_\_(self, name, age):  
       super().\_\_init\_\_(name)  # 调用父类的构造函数  
       self.age = age  
​  
child = Child("Alice", 10)  
print(child.name)  # 输出: Alice  
print(child.age)   # 输出: 10

#### 多重继承中的基本用法

在多重继承中，super 函数用于按照方法解析顺序（MRO）调用适当的方法。

class A:  
   def method(self):  
       print("A.method")  
​  
class B(A):  
   def method(self):  
       print("B.method")  
       super().method()  
​  
class C(A):  
   def method(self):  
       print("C.method")  
       super().method()  
​  
class D(B, C):  
   def method(self):  
       print("D.method")  
       super().method()  
​  
d = D()  
d.method()

输出：

D.method  
B.method  
C.method  
A.method

### 高级用法

#### 带参数的 super 调用

在某些情况下，需要显式地传递类和实例参数。

class Parent:  
   def \_\_init\_\_(self, name):  
       self.name = name  
​  
class Child(Parent):  
   def \_\_init\_\_(self, name, age):  
       super(Child, self).\_\_init\_\_(name)  # 显式传递类和实例参数  
       self.age = age  
​  
child = Child("Alice", 10)  
print(child.name)  # 输出: Alice  
print(child.age)   # 输出: 10

#### 在类方法中使用 super

在类方法中，可以使用 super 调用父类的类方法。

class Parent:  
   @classmethod  
   def class\_method(cls):  
       print(f"Parent class method called with {cls}")  
​  
class Child(Parent):  
   @classmethod  
   def class\_method(cls):  
       print(f"Child class method called with {cls}")  
       super(Child, cls).class\_method()  
​  
Child.class\_method()

输出：

Child class method called with <class '\_\_main\_\_.Child'>

Parent class method called with <class '\_\_main\_\_.Child'>

### 总结

* super 函数用于在子类中调用父类的方法，特别是在多重继承的情况下，确保方法解析顺序（MRO）的正确性。
* 基本语法为 super()，可以带参数 super(type, obj) 或 super(type, type2)。
* super 返回一个临时对象，该对象可以调用父类或兄弟类的方法。
* 基本用法包括在单继承和多重继承中调用父类的方法。
* 高级用法包括显式传递类和实例参数，以及在类方法中使用 super。

通过使用 super 函数，可以编写更加灵活和可维护的代码，特别是在处理复杂的类层次结构时。

## 16. property 函数

在Python中，property 是一个内置的装饰器，用于将类的方法转换为只读属性。它提供了一种简洁且灵活的方式来定义类的属性，使得属性的访问、设置和删除可以通过方法来控制，从而实现更好的封装和数据验证。下面是对 property 函数的详细解释，包括其语法格式、作用、基本用法和高级用法。

### 语法格式

#### 基本语法

@property  
def name(self):  
   return self.\_name

#### 完整语法（包括 setter 和 deleter）

class MyClass:  
   @property  
   def name(self):  
       return self.\_name  
​  
   @name.setter  
   def name(self, value):  
       self.\_name = value  
​  
   @name.deleter  
   def name(self):  
       del self.\_name

### 作用

property 装饰器的主要作用是将类的方法转换为只读属性，并且可以通过定义 setter 和 deleter 方法来控制属性的设置和删除操作。这样可以实现对属性的细粒度控制，包括数据验证、计算属性值等。

### 参数要求

* @property 装饰器不需要显式传递参数。
* @name.setter 装饰器需要一个参数，即要设置的值。
* @name.deleter 装饰器不需要参数。

### 返回值

* @property 方法返回属性的值。
* @name.setter 方法没有返回值，通常是 None。
* @name.deleter 方法没有返回值，通常是 None。

### 基本用法

#### 只读属性

使用 @property 装饰器定义一个只读属性。

class Person:  
   def \_\_init\_\_(self, name):  
       self.\_name = name  
​  
   @property  
   def name(self):  
       return self.\_name  
​  
person = Person("Alice")  
print(person.name)  # 输出: Alice  
# person.name = "Bob" # 这行代码会引发 AttributeError

#### 可写属性

使用 @name.setter 装饰器定义一个可写属性。

class Person:  
   def \_\_init\_\_(self, name):  
       self.\_name = name  
​  
   @property  
   def name(self):  
       return self.\_name  
​  
   @name.setter  
   def name(self, value):  
       self.\_name = value  
​  
person = Person("Alice")  
print(person.name)  # 输出: Alice  
person.name = "Bob"  
print(person.name)  # 输出: Bob

#### 可删除属性

使用 @name.deleter 装饰器定义一个可删除属性。

class Person:  
   def \_\_init\_\_(self, name):  
       self.\_name = name  
​  
   @property  
   def name(self):  
       return self.\_name  
​  
   @name.setter  
   def name(self, value):  
       self.\_name = value  
​  
   @name.deleter  
   def name(self):  
       del self.\_name  
​  
person = Person("Alice")  
print(person.name)  # 输出: Alice  
del person.name  
# print(person.name) # 这行代码会引发 AttributeError

### 高级用法

#### 数据验证

在 setter 方法中进行数据验证，确保属性值的有效性。

class Person:  
   def \_\_init\_\_(self, name):  
       self.\_name = name  
​  
   @property  
   def name(self):  
       return self.\_name  
​  
   @name.setter  
   def name(self, value):  
       if not isinstance(value, str):  
           raise ValueError("Name must be a string")  
       self.\_name = value  
​  
person = Person("Alice")  
print(person.name)  # 输出: Alice  
person.name = "Bob"  
print(person.name)  # 输出: Bob  
# person.name = 123 # 这行代码会引发 ValueError

#### 计算属性

在 property 方法中进行计算，返回计算后的属性值。

class Circle:  
   def \_\_init\_\_(self, radius):  
       self.\_radius = radius  
​  
   @property  
   def radius(self):  
       return self.\_radius  
​  
   @radius.setter  
   def radius(self, value):  
       self.\_radius = value  
​  
   @property  
   def area(self):  
       return 3.14 \* self.\_radius \*\* 2  
​  
circle = Circle(5)  
print(circle.radius)  # 输出: 5  
print(circle.area)    # 输出: 78.5

### 总结

* property 装饰器用于将类的方法转换为只读属性，并且可以通过定义 setter 和 deleter 方法来控制属性的设置和删除操作。
* 基本语法为 @property，完整语法包括 @name.setter 和 @name.deleter。
* @property 方法返回属性的值，@name.setter 方法设置属性的值，@name.deleter 方法删除属性。
* 基本用法包括定义只读属性、可写属性和可删除属性。
* 高级用法包括数据验证和计算属性。

通过使用 property 装饰器，可以实现对类属性的细粒度控制，提高代码的封装性和可维护性。

## 高级应用示例

### 1. 使用 super 函数实现多重继承

class A:  
   def method(self):  
       return "A method"  
​  
class B(A):  
   def method(self):  
       return super().method() + " in B"  
​  
class C(A):  
   def method(self):  
       return super().method() + " in C"  
​  
class D(B, C):  
   def method(self):  
       return super().method() + " in D"  
​  
d = D()  
print(d.method())  # 输出: A method in C in B in D

### 2. 使用 property 函数实现只读属性

class Circle:  
   def \_\_init\_\_(self, radius):  
       self.\_radius = radius  
​  
   @property  
   def area(self):  
       return 3.14 \* self.\_radius \* self.\_radius  
​  
circle = Circle(5)  
print(circle.area)  # 输出: 78.5  
# circle.area = 100 # 这行代码会报错，因为 area 是只读属性

### 3. 使用 \_\_call\_\_ 方法实现可调用对象

class Adder:  
   def \_\_init\_\_(self, base):  
       self.base = base  
​  
   def \_\_call\_\_(self, x):  
       return self.base + x  
​  
add5 = Adder(5)  
print(add5(10))  # 输出: 15

### 4. 使用 \_\_getitem\_\_ 和 \_\_setitem\_\_ 方法实现自定义字典

class CaseInsensitiveDict:  
   def \_\_init\_\_(self):  
       self.\_data = {}  
​  
   def \_\_getitem\_\_(self, key):  
       return self.\_data[key.lower()]  
​  
   def \_\_setitem\_\_(self, key, value):  
       self.\_data[key.lower()] = value  
​  
cid = CaseInsensitiveDict()  
cid['Name'] = 'Alice'  
print(cid['name'])  # 输出: Alice

### 5. 使用 \_\_iter\_\_ 和 \_\_next\_\_ 方法实现自定义迭代器

class Fibonacci:  
   def \_\_init\_\_(self, limit):  
       self.a, self.b = 0, 1  
       self.limit = limit  
​  
   def \_\_iter\_\_(self):  
       return self  
​  
   def \_\_next\_\_(self):  
       self.a, self.b = self.b, self.a + self.b  
       if self.a > self.limit:  
           raise StopIteration  
       return self.a  
​  
fib = Fibonacci(100)  
for num in fib:  
   print(num)  # 输出: 1 1 2 3 5 8 13 21 34 55 89

### 6. 使用 \_\_enter\_\_ 和 \_\_exit\_\_ 方法实现上下文管理器

class FileManager:  
   def \_\_init\_\_(self, filename, mode):  
       self.filename = filename  
       self.mode = mode  
       self.file = None  
​  
   def \_\_enter\_\_(self):  
       self.file = open(self.filename, self.mode)  
       return self.file  
​  
   def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
       self.file.close()  
​  
with FileManager('test.txt', 'w') as f:  
   f.write('Hello, World!')

## 总结

通过这个完整的教程，你应该对 Python 的类和对象有了全面的了解，包括类的定义和基本用法、类变量和实例变量、类方法和静态方法、私有属性和方法、多重继承、抽象基类、元类以及一些高级用法如 \_\_call\_\_、\_\_getitem\_\_、\_\_setitem\_\_、\_\_iter\_\_、\_\_next\_\_、\_\_enter\_\_ 和 \_\_exit\_\_ 方法。此外，还补充了 super 函数和 property 函数的使用，帮助你更好地理解和应用这些概念。这些内容涵盖了 Python 面向对象编程的各个方面，帮助你更好地理解和应用类和对象。

# Python异常处理

## 语法格式

### 基本语法

try:  
   # 可能引发异常的代码  
   pass  
except SomeException as e:  
   # 处理特定异常的代码  
   pass

### 完整语法

try:  
   # 可能引发异常的代码  
   pass  
except SomeException as e:  
   # 处理特定异常的代码  
   pass  
except AnotherException as e:  
   # 处理另一个特定异常的代码  
   pass  
else:  
   # 如果没有异常发生时执行的代码  
   pass  
finally:  
   # 无论是否发生异常都会执行的代码  
   pass

## 作用

try 语句的主要作用是捕获和处理程序执行过程中可能发生的异常。通过使用 try 语句，可以提高程序的健壮性，避免因异常导致程序崩溃。

## 参数要求

* try 块：包含可能引发异常的代码。
* except 块：处理特定异常的代码，可以有多个 except 块来处理不同类型的异常。
* else 块：如果没有异常发生时执行的代码。
* finally 块：无论是否发生异常都会执行的代码。

## 返回值

* try 块没有返回值。
* except 块没有返回值。
* else 块没有返回值。
* finally 块没有返回值。

## 基本用法

### 捕获和处理异常

使用 try 和 except 块捕获和处理异常。

try:  
   result = 10 / 0  
except ZeroDivisionError as e:  
   print(f"Error: {e}")  # 输出: Error: division by zero

### 捕获多种异常

使用多个 except 块捕获多种异常。

try:  
   result = 10 / 0  
except ZeroDivisionError as e:  
   print(f"ZeroDivisionError: {e}")  
except TypeError as e:  
   print(f"TypeError: {e}")

### 使用 else 块

如果没有异常发生时执行 else 块。

try:  
   result = 10 / 2  
except ZeroDivisionError as e:  
   print(f"Error: {e}")  
else:  
   print(f"Result: {result}")  # 输出: Result: 5.0

### 使用 finally 块

无论是否发生异常都会执行 finally 块。

try:  
   result = 10 / 0  
except ZeroDivisionError as e:  
   print(f"Error: {e}")  
finally:  
   print("Finally block executed")  # 输出: Finally block executed

## 高级用法

### 自定义异常

定义和引发自定义异常。

class MyCustomError(Exception):  
   def \_\_init\_\_(self, message):  
       self.message = message  
       super().\_\_init\_\_(self.message)  
​  
try:  
   raise MyCustomError("This is a custom error")  
except MyCustomError as e:  
   print(f"Custom Error: {e}")  # 输出: Custom Error: This is a custom error

### 嵌套 try 语句

在 try 块中嵌套另一个 try 语句。

try:  
   try:  
       result = 10 / 0  
   except ZeroDivisionError as e:  
       print(f"Inner Error: {e}")  # 输出: Inner Error: division by zero  
except Exception as e:  
   print(f"Outer Error: {e}")

### 使用 raise 重新引发异常

#### 语法格式

##### 基本语法

raise ExceptionType("error message")

* ExceptionType：异常类型，可以是内置的异常类型（如 ValueError、TypeError）或自定义的异常类型。
* "error message"：可选参数，当异常被引发时，作为异常的错误信息。

#### 作用

raise 语句用于在代码中手动引发异常。通过 raise 语句，开发者可以在特定条件下主动抛出异常，以便在程序的其他部分进行处理。

#### 基本用法

##### 引发内置异常

使用 raise 语句引发内置的异常类型。

raise ValueError("Invalid value")  # 引发 ValueError 异常，并输出 "Invalid value"

##### 引发自定义异常

定义一个自定义异常类，并使用 raise 语句引发该异常。

class MyCustomError(Exception):  
   pass  
​  
raise MyCustomError("This is a custom error")  # 引发自定义异常 MyCustomError，并输出 "This is a custom error"

#### 高级用法

##### 结合 try 和 except 使用

在 try 块中使用 raise 语句引发异常，并在 except 块中处理该异常。

try:  
   raise ValueError("An error occurred")  
except ValueError as e:  
   print(f"Caught an exception: {e}")  # 输出: Caught an exception: An error occurred

##### 引发异常并传递信息

在引发异常时，传递额外的信息以便在异常处理时使用。

def divide(a, b):  
   if b == 0:  
       raise ValueError(f"Denominator cannot be zero: {a} / {b}")  
   return a / b  
​  
try:  
   result = divide(10, 0)  
except ValueError as e:  
   print(f"Error: {e}")  # 输出: Error: Denominator cannot be zero: 10 / 0

##### 引发异常并重新引发

在 except 块中处理异常后，可以选择重新引发该异常。

try:  
   raise ValueError("Original error")  
except ValueError as e:  
   print(f"Caught an exception: {e}")  # 输出: Caught an exception: Original error  
   raise  # 重新引发原始异常

##### 引发异常并包装

在 except 块中处理异常后，可以选择引发一个新的异常，并将原始异常作为其原因。

try:  
   raise ValueError("Original error")  
except ValueError as e:  
   raise TypeError("Wrapped error") from e  # 引发新的 TypeError 异常，并将原始 ValueError 异常作为其原因

##### 引发异常并记录日志

在引发异常时，记录日志以便后续分析。

import logging  
​  
logging.basicConfig(level=logging.ERROR)  
​  
def log\_error():  
   try:  
       raise ValueError("An error occurred")  
   except ValueError as e:  
       logging.error(f"Caught an exception: {e}")  # 记录错误日志  
       raise  # 重新引发原始异常  
​  
log\_error()

##### 引发异常并返回特定值

在函数中引发异常，并在异常处理后返回特定值。

def safe\_divide(a, b):  
   try:  
       return a / b  
   except ZeroDivisionError:  
       raise ValueError("Denominator cannot be zero")  
   except TypeError:  
       raise TypeError("Both arguments must be numbers")  
​  
try:  
   result = safe\_divide(10, 0)  
except ValueError as e:  
   print(f"Error: {e}")  # 输出: Error: Denominator cannot be zero  
except TypeError as e:  
   print(f"Error: {e}")  # 输出: Error: Both arguments must be numbers

##### 引发异常并处理多个异常

在 try 块中引发多个异常，并在 except 块中分别处理。

try:  
   value = int("abc")  
except ValueError:  
   raise ValueError("Invalid integer conversion")  
except TypeError:  
   raise TypeError("Invalid type conversion")

#### 总结

* raise 语句用于在代码中手动引发异常，可以是内置的异常类型或自定义的异常类型。
* 基本语法为 raise ExceptionType("error message")，其中 ExceptionType 是异常类型，"error message" 是可选的错误信息。
* 基本用法包括引发内置异常和自定义异常。
* 高级用法包括结合 try 和 except 使用、引发异常并传递信息、引发异常并重新引发、引发异常并包装、引发异常并记录日志、引发异常并返回特定值、引发异常并处理多个异常。

通过使用 raise 语句，开发者可以在特定条件下主动抛出异常，以便在程序的其他部分进行处理，从而提高代码的健壮性和可维护性

### 使用 assert 语句

#### 语法格式

#### 基本语法

assert condition, message

* condition：一个布尔表达式，如果为 False，将引发 AssertionError。
* message：可选参数，当 condition 为 False 时，作为 AssertionError 的错误信息。

#### 作用

assert 语句用于在代码中进行断言检查。如果断言条件为 True，程序继续执行；如果断言条件为 False，将引发 AssertionError 异常，并输出可选的错误信息。

#### 基本用法

#### 简单断言

检查一个条件是否为 True，如果不为 True，则引发 AssertionError。

assert 1 == 1  # 条件为 True，程序继续执行  
assert 1 == 2  # 条件为 False，引发 AssertionError

#### 带错误信息的断言

在断言失败时，输出自定义的错误信息。

assert 1 == 2, "1 is not equal to 2"  # 引发 AssertionError，并输出 "1 is not equal to 2"

#### 高级用法

##### 检查函数返回值

在函数调用后使用 assert 检查返回值是否符合预期。

def add(a, b):  
   return a + b  
​  
result = add(2, 3)  
assert result == 5, f"Expected 5, but got {result}"  # 条件为 True，程序继续执行  
​  
result = add(2, 3)  
assert result == 6, f"Expected 6, but got {result}"  # 条件为 False，引发 AssertionError

##### 检查列表长度

使用 assert 检查列表的长度是否符合预期。

my\_list = [1, 2, 3, 4, 5]  
assert len(my\_list) == 5, f"Expected length 5, but got {len(my\_list)}"  # 条件为 True，程序继续执行  
​  
my\_list = [1, 2, 3]  
assert len(my\_list) == 5, f"Expected length 5, but got {len(my\_list)}"  # 条件为 False，引发 AssertionError

##### 检查字典键值

使用 assert 检查字典中是否包含特定键值对。

my\_dict = {'name': 'Alice', 'age': 30}  
assert my\_dict['name'] == 'Alice', f"Expected 'Alice', but got {my\_dict['name']}"  # 条件为 True，程序继续执行  
​  
assert my\_dict['age'] == 30, f"Expected 30, but got {my\_dict['age']}"  # 条件为 True，程序继续执行  
​  
assert my\_dict['age'] == 25, f"Expected 25, but got {my\_dict['age']}"  # 条件为 False，引发 AssertionError

##### 结合 try 和 except 使用

在 try 块中使用 assert 进行断言检查，并在 except 块中处理 AssertionError。

try:  
   assert 1 == 2, "Assertion failed"  
except AssertionError as e:  
   print(f"Assertion Error: {e}")  # 输出: Assertion Error: Assertion failed

##### 检查文件是否存在

使用 assert 检查文件是否存在。

import os  
​  
file\_path = 'file.txt'  
assert os.path.exists(file\_path), f"File {file\_path} does not exist"  # 如果文件不存在，引发 AssertionError

##### 检查变量类型

使用 assert 检查变量是否为特定类型。

value = 42  
assert isinstance(value, int), f"Expected an integer, but got {type(value)}"  # 条件为 True，程序继续执行  
​  
value = "42"  
assert isinstance(value, int), f"Expected an integer, but got {type(value)}"  # 条件为 False，引发 AssertionError

##### 检查函数参数

在函数内部使用 assert 检查参数是否符合预期。

def divide(a, b):  
   assert b != 0, "Denominator cannot be zero"  
   return a / b  
​  
result = divide(10, 2)  # 条件为 True，程序继续执行  
print(result)  # 输出: 5.0  
​  
result = divide(10, 0)  # 条件为 False，引发 AssertionError

#### 总结

* assert 语句用于在代码中进行断言检查，如果条件为 False，将引发 AssertionError 异常。
* 基本语法为 assert condition, message，其中 condition 是一个布尔表达式，message 是可选的错误信息。
* 基本用法包括简单断言和带错误信息的断言。
* 高级用法包括检查函数返回值、检查列表长度、检查字典键值、结合 try 和 except 使用、检查文件是否存在、检查变量类型和检查函数参数。

通过使用 assert 语句，可以在代码中进行断言检查，确保程序的正确性和健壮性。使用 with 语句

with 语句可以与 try 语句结合使用，用于管理资源。

try:  
   with open('file.txt', 'r') as file:  
       content = file.read()  
except FileNotFoundError as e:  
   print(f"FileNotFoundError: {e}")

### 捕获所有异常

使用 except 块捕获所有类型的异常。

try:  
   result = 10 / 0  
except Exception as e:  
   print(f"An error occurred: {e}")  # 输出: An error occurred: division by zero

### 使用 traceback 模块

使用 traceback 模块获取更详细的异常信息。

import traceback  
​  
try:  
   result = 10 / 0  
except ZeroDivisionError as e:  
   print(f"Error: {e}")  
   traceback.print\_exc()  # 输出详细的异常信息

## 总结

* try 语句用于捕获和处理程序执行过程中可能发生的异常。
* 基本语法包括 try 和 except 块，完整语法还包括 else 和 finally 块。
* 基本用法包括捕获和处理异常、捕获多种异常、使用 else 块和 finally 块。
* 高级用法包括自定义异常、嵌套 try 语句、使用 raise 重新引发异常、使用 assert 语句进行断言检查、使用 with 语句管理资源、捕获所有异常和使用 traceback 模块获取详细异常信息。

通过使用 try 语句，可以有效地捕获和处理异常，提高程序的健壮性和可靠性。

## Python常见错误表

| **错误类型** | **含义** | **解决方式** |
| --- | --- | --- |
| SyntaxError | 代码不符合Python语法规则 | 检查代码中的拼写错误、缺少括号、引号等语法问题 |
| IndentationError | 代码缩进不一致或不正确 | 确保同一层级的代码使用相同的缩进（通常是4个空格或一个Tab键） |
| NameError | 尝试使用一个未定义的变量或函数 | 确保所有使用的变量和函数都已正确定义 |
| TypeError | 对不适当类型的对象执行操作 | 检查操作的对象类型是否正确，例如字符串和整数不能相加 |
| ValueError | 操作或函数接收到类型正确但值不合适的参数 | 检查传递给函数的参数值是否在允许范围内 |
| IndexError | 尝试访问序列中不存在的索引 | 确保使用的索引在序列的有效范围内 |
| KeyError | 尝试访问字典中不存在的键 | 确保使用的键在字典中存在，或者使用dict.get()方法避免错误 |
| AttributeError | 尝试访问对象不存在的属性或方法 | 检查对象的类型，确保访问的属性或方法存在 |
| ZeroDivisionError | 尝试除以零 | 在进行除法操作前检查除数是否为零 |
| FileNotFoundError | 尝试打开不存在的文件 | 确保文件路径正确，或者使用try-except块捕获异常 |
| IOError | 输入输出操作失败 | 检查文件路径、权限等问题，确保操作的文件存在且可访问 |
| ImportError | 尝试导入不存在的模块或模块中的不存在的属性 | 确保导入的模块和属性存在，或者安装缺失的模块 |
| RecursionError | 递归调用超过Python允许的最大深度 | 优化递归算法，使用循环代替递归，或者增加递归深度限制 |
| MemoryError | 程序运行时内存不足 | 优化代码，减少内存使用，或者增加系统内存 |
| OverflowError | 数值运算结果超出数据类型允许的范围 | 使用更大范围的数据类型，或者优化算法避免溢出 |
| StopIteration | 迭代器没有更多的值可供迭代 | 确保迭代器的使用方式正确，或者在适当的位置捕获该异常 |
| AssertionError | assert语句失败 | 检查assert语句的条件是否正确，确保程序逻辑符合预期 |
| TabError | 代码中混用了Tab和空格进行缩进 | 统一使用Tab或空格进行缩进，避免混用 |
| UnboundLocalError | 尝试访问未初始化的局部变量 | 确保所有局部变量在使用前已正确初始化 |
| UnicodeError | 处理Unicode字符串时发生错误 | 确保字符串编码和解码方式正确，使用合适的编码处理字符串 |
| SystemError | Python解释器内部错误 | 报告给Python社区，或者检查代码中是否有不规范的操作 |
| SystemExit | 调用sys.exit()函数或exit()函数 | 确保在适当的位置调用退出函数，或者捕获该异常进行处理 |
| KeyboardInterrupt | 用户中断程序执行（通常是按Ctrl+C） | 在适当的位置捕获该异常，进行清理操作或提示用户 |

# Python正则

## 正则表达式

正则表达式（Regular Expression，简称 regex 或 regexp）是一种用于匹配字符串的模式。它广泛应用于文本搜索和文本替换等操作。以下是正则表达式的详细语法及其作用。

### 基本语法

#### 字符匹配

* **普通字符**: 直接匹配自身。
  + 示例: a 匹配字符 "a"。

#### 元字符

* **.**: 匹配任意单个字符（除换行符外）。
  + 示例: a.b 匹配 "aab", "axb", "a1b" 等。
* **^**: 匹配字符串的开始位置。
  + 示例: ^abc 匹配以 "abc" 开头的字符串。
* **$**: 匹配字符串的结束位置。
  + 示例: xyz$ 匹配以 "xyz" 结尾的字符串。
* **\***: 匹配前面的子表达式零次或多次。
  + 示例: ab\*c 匹配 "ac", "abc", "abbc" 等。
* **+**: 匹配前面的子表达式一次或多次。
  + 示例: ab+c 匹配 "abc", "abbc", "abbbc" 等。
* **?**: 匹配前面的子表达式零次或一次。
  + 示例: ab?c 匹配 "ac" 或 "abc"。
* **{n}**: 匹配前面的子表达式恰好 n 次。
  + 示例: a{3} 匹配 "aaa"。
* **{n,}**: 匹配前面的子表达式至少 n 次。
  + 示例: a{2,} 匹配 "aa", "aaa", "aaaa" 等。
* **{n,m}**: 匹配前面的子表达式至少 n 次，至多 m 次。
  + 示例: a{2,4} 匹配 "aa", "aaa", "aaaa"。
* **[]**: 字符集合，匹配方括号内的任意一个字符。
  + 示例: [abc] 匹配 "a", "b", "c"。
* **[^]**: 否定字符集合，匹配不在方括号内的任意一个字符。
  + 示例: [^abc] 匹配除 "a", "b", "c" 之外的任意字符。
* **|**: 或运算符，匹配两个或多个表达式中的一个。
  + 示例: a|b 匹配 "a" 或 "b"。
* **()**: 分组，将多个字符组合成一个子表达式。
  + 示例: (abc)+ 匹配 "abc", "abcabc" 等。

#### 字符类

* **\\d**: 匹配任意一个数字字符，等价于 [0-9]。
  + 示例: \\d{3} 匹配 "123", "456" 等。
* **\\D**: 匹配任意一个非数字字符，等价于 [^0-9]。
  + 示例: \\D+ 匹配 "abc", "xyz" 等。
* **\\s**: 匹配任意一个空白字符，包括空格、制表符、换页符等。
  + 示例: \\s+ 匹配一个或多个空白字符。
* **\\S**: 匹配任意一个非空白字符。
  + 示例: \\S+ 匹配一个或多个非空白字符。
* **\\w**: 匹配任意一个字母、数字或下划线字符，等价于 [A-Za-z0-9\_]。
  + 示例: \\w+ 匹配 "hello", "world123" 等。
* **\\W**: 匹配任意一个非字母、数字或下划线字符，等价于 [^A-Za-z0-9\_]。
  + 示例: \\W+ 匹配 "!@#", " " 等。

#### 常用搭配

* **\\b**: 匹配单词边界。
  + 示例: \\bword\\b 匹配独立的单词 "word"。
* **\\B**: 匹配非单词边界。
  + 示例: \\Bword\\B 匹配非独立的单词 "word"。
* **(?P<name>...)**: 命名分组，给分组命名以便后续引用。
  + 示例: (?P<year>\\d{4}) 匹配四位数字并命名为 "year"。
* **(?=...)**: 正向先行断言，匹配后面跟着特定模式的字符串。
  + 示例: \\w+(?=\\s) 匹配后面跟着空格的单词。
* **(?!...)**: 负向先行断言，匹配后面不跟着特定模式的字符串。
  + 示例: \\d{3}(?!\\d) 匹配后面不跟着数字的三位数字。
* **(?<=...)**: 正向后行断言，匹配前面跟着特定模式的字符串。
  + 示例: (?<=\\s)\\w+ 匹配前面跟着空格的单词。
* **(?<!...)**: 负向后行断言，匹配前面不跟着特定模式的字符串。
  + 示例: (?<!\\d)\\d{3} 匹配前面不跟着数字的三位数字。

### 示例

#### 匹配邮箱地址

^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$

解释: 匹配一个标准的邮箱地址。

#### 匹配手机号码

^1[3-9]\\d{9}$

解释: 匹配中国的手机号码，以 1 开头，第二位是 3-9 之间的数字，后面跟着 9 位数字。

#### 匹配日期格式 (YYYY-MM-DD)

^\\d{4}-\\d{2}-\\d{2}$

解释: 匹配格式为 YYYY-MM-DD 的日期。

#### 匹配 HTML 标签

<[^>]+>

解释: 匹配任意 HTML 标签。

通过以上语法和示例，可以灵活运用正则表达式进行复杂的文本匹配和处理。a

## 模块常用方法

### 1. re.compile(pattern, flags=0)

* **语法格式**: re.compile(pattern, flags=0)
* 参数:
  + pattern: 要编译的正则表达式。
  + flags: 可选参数，用于控制正则表达式的匹配方式。
* **作用**: 将正则表达式编译成一个正则表达式对象，提高匹配效率。

**示例**:

import re  
​  
pattern = r'\\d+'  
regex = re.compile(pattern)  
string = 'abc123def456ghi'  
match = regex.search(string)  
if match:  
   print("Match found:", match.group())  
else:  
   print("No match")

输出:

Match found: 123

### 2. re.match(pattern, string, flags=0)

* **语法格式**: re.match(pattern, string, flags=0)
* 参数:
  + pattern: 要匹配的正则表达式。
  + string: 要匹配的字符串。
  + flags: 可选参数，用于控制正则表达式的匹配方式。
* **作用**: 从字符串的起始位置匹配正则表达式，如果匹配成功，返回一个匹配对象；否则返回 None。

**示例**:

import re  
​  
pattern = r'\\d+'  
string = '123abc'  
match = re.match(pattern, string)  
if match:  
   print("Match found:", match.group())  
else:  
   print("No match")

输出:

Match found: 123

### 3. re.search(pattern, string, flags=0)

* **语法格式**: re.search(pattern, string, flags=0)
* 参数:
  + pattern: 要匹配的正则表达式。
  + string: 要匹配的字符串。
  + flags: 可选参数，用于控制正则表达式的匹配方式。
* **作用**: 扫描整个字符串并返回第一个成功的匹配对象，如果匹配失败，返回 None。

**示例**:

import re  
​  
pattern = r'\\d+'  
string = 'abc123def456ghi'  
search = re.search(pattern, string)  
if search:  
   print("Search found:", search.group())  
else:  
   print("No search")

输出:

Search found: 123

### 4. re.fullmatch(pattern, string, flags=0)

* **语法格式**: re.fullmatch(pattern, string, flags=0)
* 参数:
  + pattern: 要匹配的正则表达式。
  + string: 要匹配的字符串。
  + flags: 可选参数，用于控制正则表达式的匹配方式。
* **作用**: 完全匹配整个字符串，如果匹配成功，返回一个匹配对象；否则返回 None。

**示例**:

import re  
​  
pattern = r'\\d+'  
string = '123'  
fullmatch = re.fullmatch(pattern, string)  
if fullmatch:  
   print("Full match found:", fullmatch.group())  
else:  
   print("No full match")

输出:

Full match found: 123

### 5. re.findall(pattern, string, flags=0)

* **语法格式**: re.findall(pattern, string, flags=0)
* 参数:
  + pattern: 要匹配的正则表达式。
  + string: 要匹配的字符串。
  + flags: 可选参数，用于控制正则表达式的匹配方式。
* **作用**: 返回字符串中所有非重叠匹配的子串列表。

**示例**:

import re  
​  
pattern = r'\\d+'  
string = 'abc123def456ghi'  
findall = re.findall(pattern, string)  
print("Findall results:", findall)

输出:

Findall results: ['123', '456']

### 6. re.finditer(pattern, string, flags=0)

* **语法格式**: re.finditer(pattern, string, flags=0)
* 参数:
  + pattern: 要匹配的正则表达式。
  + string: 要匹配的字符串。
  + flags: 可选参数，用于控制正则表达式的匹配方式。
* **作用**: 返回一个迭代器，包含字符串中所有非重叠匹配的匹配对象。

**示例**:

import re  
​  
pattern = r'\\d+'  
string = 'abc123def456ghi'  
finditer = re.finditer(pattern, string)  
for match in finditer:  
   print("Finditer match:", match.group())

输出:

Finditer match: 123  
Finditer match: 456

### 7. re.sub(pattern, repl, string, count=0, flags=0)

* **语法格式**: re.sub(pattern, repl, string, count=0, flags=0)
* 参数:
  + pattern: 要匹配的正则表达式。
  + repl: 替换的字符串或替换函数。
  + string: 要匹配的字符串。
  + `count项。
  + flags: 可选参数，用于控制正则表达式的匹配方式。
* **作用**: 返回一个新字符串，其中所有匹配的子串都被替换为指定的字符串或函数返回值。

**示例**:

import re  
​  
pattern = r'\\d+'  
repl = 'X'  
string = 'abc123def456ghi'  
result = re.sub(pattern, repl, string)  
print("Sub result:", result)

输出:

Sub result: abcXdefXghi

### 8. re.subn(pattern, repl, string, count=0, flags=0)

* **语法格式**: re.subn(pattern, repl, string, count=0, flags=0)
* 参数:
  + pattern: 要匹配的正则表达式。
  + repl: 替换的字符串或替换函数。
  + string: 要匹配的字符串。
  + count: 可选参数，替换的最大次数，默认为 0，表示替换所有匹配项。
  + flags: 可选参数，用于控制正则表达式的匹配方式。
* **作用**: 返回一个元组，包含新字符串和替换次数。

**示例**:

import re  
​  
pattern = r'\\d+'  
repl = 'X'  
string = 'abc123def456ghi'  
result = re.subn(pattern, repl, string)  
print("Subn result:", result)

输出:

Subn result: ('abcXdefXghi', 2)

### 9. re.split(pattern, string, maxsplit=0, flags=0)

* **语法格式**: re.split(pattern, string, maxsplit=0, flags=0)
* 参数:
  + pattern: 要匹配的正则表达式。
  + string: 要匹配的字符串。
  + maxsplit: 可选参数，最大分割次数，默认为 0，表示分割所有匹配项。
  + flags: 可选参数，用于控制正则表达式的匹配方式。
* **作用**: 返回一个列表，包含字符串被分割后的子串。

**示例**:

import re  
​  
pattern = r'\\d+'  
string = 'abc123def456ghi'  
result = re.split(pattern, string)  
print("Split result:", result)

输出:

Split result: ['abc', 'def', 'ghi']

### 10. re.escape(string)

* **语法格式**: re.escape(string)
* 参数:
  + string: 要转义的字符串。
* **作用**: 返回一个字符串，其中所有非字母数字字符都被转义。

**示例**:

import re  
​  
string = 'Hello, [World]!'  
escaped\_string = re.escape(string)  
print("Escaped string:", escaped\_string)

输出:

Escaped string: Hello\\,\\ \\[World\\]\\!

## 常用搭配

### 1. 匹配单个字符

* .：匹配除换行符以外的任意字符。
* \\d：匹配任意数字。
* \\D：匹配任意非数字字符。
* \\w：匹配任意字母、数字或下划线。
* \\W：匹配任意非字母、数字或下划线字符。
* \\s：匹配任意空白字符。
* \\S：匹配任意非空白字符。

**示例**:

import re  
​  
pattern = r'\\w+'  
string = 'Hello, World!'  
match = re.match(pattern, string)  
if match:  
   print("Match found:", match.group())  
else:  
   print("No match")

输出:

Match found: Hello

### 2. 匹配多个字符

* \*：匹配前面的字符零次或多次。
* +：匹配前面的字符一次或多次。
* ?：匹配前面的字符零次或一次。
* {n}：匹配前面的字符恰好 n 次。
* {n,}：匹配前面的字符至少 n 次。
* {n,m}：匹配前面的字符至少 n 次，但不超过 m 次。

**示例**:

import re  
​  
pattern = r'a{2,3}'  
string = 'a aa aaa aaaa'  
findall = re.findall(pattern, string)  
print("Findall results:", findall)

输出:

Findall results: ['aa', 'aaa']

### 3. 匹配边界

* ^：匹配字符串的开头。
* $：匹配字符串的结尾。
* \\b：匹配单词边界。
* \\B：匹配非单词边界。

**示例**:

import re  
​  
pattern = r'^\\d+'  
string = '123abc'  
match = re.match(pattern, string)  
if match:  
   print("Match found:", match.group())  
else:  
   print("No match")

输出:

Match found: 123

### 4. 分组和捕获

* (...)：分组和捕获匹配的子串。
* (?:...)：仅分组，不捕获匹配的子串。
* (?P<name>...)：命名分组，捕获匹配的子串并赋予名称。

**示例**:

import re  
​  
pattern = r'(?P<year>\\d{4})-(?P<month>\\d{2})-(?P<day>\\d{2})'  
string = '2023-04-30'  
match = re.match(pattern, string)  
if match:  
   print("Year:", match.group('year'))  
   print("Month:", match.group('month'))  
   print("Day:", match.group('day'))

输出:

Year: 2023  
Month: 04  
Day: 30

## 总结

Python 的 re 模块提供了丰富的正则表达式功能，通过上述常用方法和搭配，可以灵活地进行字符串匹配、搜索、替换和分割等操作。希望这些讲解和示例能帮助你更好地理解和使用 re 模块。

# Python文件操作

## open函数

### 1. open() 函数

* **语法格式**：open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

| **参数名** | **描述** |
| --- | --- |
| file | 必需，要打开的文件的路径。 |
| mode | 可选，文件打开模式。默认为 'r'（只读）。常见的模式有：'r'、'w'、'x'、'a'、'b'、't'、'+'。 |
| buffering | 可选，设置缓冲区的大小。默认为 -1，表示使用系统默认的缓冲区大小。 |
| encoding | 可选，指定文件的编码格式。默认为 None，表示使用系统默认编码。 |
| errors | 可选，指定如何处理编码错误。默认为 None，表示使用系统默认处理方式。 |
| newline | 可选，指定如何处理换行符。默认为 None，表示使用系统默认处理方式。 |
| closefd | 可选，如果为 True，关闭文件时同时关闭文件描述符。默认为 True。 |
| opener | 可选，指定一个自定义的打开文件的方法。默认为 None。 |

* **返回值**：返回一个文件对象。

### 2. 文件对象的常用方法

##### 2.1 read([size])

* **语法格式**：file.read([size])
* **参数**：size 为可选参数，指定读取的最大字节数。
* **作用**：读取文件内容。
* **返回值**：读取的内容（字符串或字节串）。

##### 2.2 readline([sizehint])

* **语法格式**：file.readline([sizehint])
* **参数**：sizehint 为可选参数，指定读取的最大字节数。
* **作用**：读取一行。
* **返回值**：读取的行（字符串或字节串）。

##### 2.3 readlines([sizehint])

* **语法格式**：file.readlines([sizehint])
* **参数**：sizehint 为可选参数，指定读取的最大字节数。
* **作用**：读取所有行。
* **返回值**：行的列表（列表元素为字符串或字节串）。

##### 2.4 write(str)

* **语法格式**：file.write(str)
* **参数**：str 为要写入的字符串。
* **作用**：写入字符串到文件。
* **返回值**：写入的字节数。

##### 2.5 writelines(lines)

* **语法格式**：file.writelines(lines)
* **参数**：lines 为要写入的字符串列表。
* **作用**：写入一个字符串列表到文件。
* **返回值**：无。

##### 2.6 close()

* **语法格式**：file.close()
* **参数**：无。
* **作用**：关闭文件。
* **返回值**：无。

##### 2.7 seek(offset[, whence])

* **语法格式**：file.seek(offset[, whence])
* **参数**：offset 为偏移量，whence 为可选参数，默认为 os.SEEK\_SET（0），表示从文件开头开始计算偏移量。其他可选值为 os.SEEK\_CUR（1）和 os.SEEK\_END（2）。
* **作用**：移动文件指针到指定位置。
* **返回值**：新的文件指针位置。

##### 2.8 tell()

* **语法格式**：file.tell()
* **参数**：无。
* **作用**：返回当前文件指针的位置。
* **返回值**：当前文件指针的位置。

##### 2.9 flush()

* **语法格式**：file.flush()
* **参数**：无。
* **作用**：将缓冲区内容写入文件，但不关闭文件。
* **返回值**：无。

##### 2.10 truncate([size])

* **语法格式**：file.truncate([size])
* **参数**：size 为可选参数，指定新的文件大小。
* **作用**：截断文件到指定大小。
* **返回值**：新的文件大小。

### 常用的搭配

* **with open(file, mode) as file:**：使用 with 语句可以确保文件在操作完成后自动关闭。

with 语句在 Python 中用于简化资源管理，确保在代码块执行完毕后自动执行清理操作，如关闭文件、释放锁等。with 语句通常与上下文管理器（context manager）一起使用。上下文管理器是通过实现 \_\_enter\_\_ 和 \_\_exit\_\_ 方法的对象。

#### 语法格式

with context\_expression [as target(s)]:  
   with-body

#### 参数要求和作用

* **context\_expression**：必需，返回一个上下文管理器的表达式。
* **as target(s)**：可选，将上下文管理器的 \_\_enter\_\_ 方法的返回值赋给目标变量。
* **with-body**：必需，包含在 with 语句块中的代码。

#### 返回值和返回值的类型与作用

* **\_\_enter\_\_ 方法**：返回一个对象，该对象可以在 with 语句块中使用。返回值的类型取决于上下文管理器的实现。
* **\_\_exit\_\_ 方法**：在 with 语句块执行完毕后调用，用于执行清理操作。返回值通常为 None，但如果返回 True，则表示异常已被处理，不会抛出异常。

#### 常用的搭配

* **文件操作**：with open(file, mode) as file:
* **锁操作**：with threading.Lock() as lock:
* **数据库连接**：with sqlite3.connect(database) as conn:

#### 具体例子

##### 示例1：文件操作

# 打开文件并读取内容  
with open('example.txt', 'r') as file:  
   content = file.read()  
   print(content)  
# 文件在with语句块结束后自动关闭

##### 示例2：锁操作

import threading  
​  
lock = threading.Lock()  
​  
# 使用锁保护共享资源  
with lock:  
   # 访问共享资源  
   print("Accessing shared resource")  
# 锁在with语句块结束后自动释放

##### 示例3：数据库连接

import sqlite3  
​  
# 连接到数据库  
with sqlite3.connect('example.db') as conn:  
   cursor = conn.cursor()  
   cursor.execute('SELECT \* FROM table\_name')  
   result = cursor.fetchall()  
   print(result)  
# 数据库连接在with语句块结束后自动关闭

##### 示例4：自定义上下文管理器

class MyContextManager:  
   def \_\_enter\_\_(self):  
       print("Entering context")  
       return self  # 返回自身对象  
​  
   def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
       print("Exiting context")  
       if exc\_type is not None:  
           print(f"Exception handled: {exc\_type}, {exc\_value}")  
       return True  # 表示异常已被处理  
​  
# 使用自定义上下文管理器  
with MyContextManager() as cm:  
   print("Inside context")  
   # 故意引发异常  
   raise ValueError("An error occurred")  
# 上下文管理器在with语句块结束后自动执行清理操作

##### 示例5：嵌套上下文管理器

# 嵌套使用多个上下文管理器  
with open('file1.txt', 'r') as file1, open('file2.txt', 'w') as file2:  
   content = file1.read()  
   file2.write(content)  
# 两个文件在with语句块结束后自动关闭

#### 总结

with 语句通过上下文管理器简化了资源管理，确保在代码块执行完毕后自动执行清理操作。常用的搭配包括文件操作、锁操作和数据库连接。通过实现 \_\_enter\_\_ 和 \_\_exit\_\_ 方法，可以创建自定义的上下文管理器。with 语句的语法格式简单明了，能够提高代码的可读性和可靠性。

### 示例

##### 示例1：打开文件并读取内容

# 打开文件  
with open('example.txt', 'r') as file:  
   content = file.read()  
   print(content)

##### 示例2：写入文件

# 打开文件  
with open('example.txt', 'w') as file:  
   file.write('Hello, world!')

##### 示例3：追加内容到文件

# 打开文件  
with open('example.txt', 'a') as file:  
   file.write('\\nThis is a new line.')

##### 示例4：逐行读取文件

# 打开文件  
with open('example.txt', 'r') as file:  
   for line in file:  
       print(line.strip())

##### 示例5：读取所有行并处理

# 打开文件  
with open('example.txt', 'r') as file:  
   lines = file.readlines()  
   for line in lines:  
       print(line.strip())

##### 示例6：写入多行内容

# 打开文件  
with open('example.txt', 'w') as file:  
   lines = ['Line 1\\n', 'Line 2\\n', 'Line 3\\n']  
   file.writelines(lines)

##### 示例7：移动文件指针

# 打开文件  
with open('example.txt', 'r+') as file:  
   file.seek(10)  # 移动文件指针到第10个字节  
   content = file.read()  
   print(content)

##### 示例8：获取当前文件指针位置

# 打开文件  
with open('example.txt', 'r') as file:  
   file.read(5)  # 读取5个字节  
   position = file.tell()  
   print(position)  # 输出当前文件指针位置

##### 示例9：刷新缓冲区

# 打开文件  
with open('example.txt', 'w') as file:  
   file.write('Hello, world!')  
   file.flush()  # 刷新缓冲区

##### 示例10：截断文件

# 打开文件  
with open('example.txt', 'r+') as file:  
   file.truncate(10)  # 截断文件到10个字节

以上是 Python 中文件操作的详细介绍，包括所有的函数和它们的语法格式、参数要求、作用、返回值类型与作用，以及常用的搭配和具体示例。通过这些操作，可以方便地进行文件的读写和管理。

## json模块

json 模块是 Python 标准库中的一个模块，用于处理 JSON（JavaScript Object Notation）数据格式。JSON 是一种轻量级的数据交换格式，易于人阅读和编写，也易于机器解析和生成。json 模块提供了四个主要的方法：dumps、dump、loads 和 load，以及一个用于自定义编码和解码的类 JSONEncoder。

#### 1. json.dumps(obj, \*, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw)

* **参数要求和作用**：
  + obj：要编码的 Python 对象。
  + skipkeys：如果为 True，则跳过非基本类型的键（如 int、str、float 等）。
  + ensure\_ascii：如果为 True，则所有非 ASCII 字符将被转义。
  + check\_circular：如果为 True，则检查循环引用。
  + allow\_nan：如果为 True，则允许 NaN、Infinity 和 -Infinity。
  + cls：自定义的 JSONEncoder 子类。
  + indent：缩进级别，用于美化输出。
  + separators：分隔符元组 (item\_separator, key\_separator)。
  + default：用于处理不可序列化对象的函数。
  + sort\_keys：如果为 True，则按字典键排序。
* **返回值和返回值的类型与作用**：
  + 返回值：JSON 格式的字符串。
  + 返回值类型：str。
* **示例**：

import json  
​  
data = {'name': 'Alice', 'age': 30}  
json\_str = json.dumps(data, indent=4)  
print(json\_str)

输出：

{  
   "name": "Alice",  
   "age": 30  
}

#### 2. json.dump(obj, fp, \*, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw)

* **参数要求和作用**：
  + obj：要编码的 Python 对象。
  + fp：文件对象，用于写入 JSON 数据。
  + 其他参数与 json.dumps 相同。
* **返回值和返回值的类型与作用**：
  + 返回值：无返回值（None）。
  + 返回值类型：None。
* **示例**：

import json  
​  
data = {'name': 'Alice', 'age': 30}  
with open('data.json', 'w') as fp:  
   json.dump(data, fp, indent=4)

生成的 data.json 文件内容：

{  
   "name": "Alice",  
   "age": 30  
}

#### 3. json.loads(s, \*, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw)

* **参数要求和作用**：
  + s：要解码的 JSON 字符串。
  + cls：自定义的 JSONDecoder 子类。
  + object\_hook：用于处理解码后的字典对象的函数。
  + parse\_float：用于处理浮点数的函数。
  + parse\_int：用于处理整数的函数。
  + parse\_constant：用于处理常量的函数。
  + object\_pairs\_hook：用于处理解码后的有序键值对列表的函数。
* **返回值和返回值的类型与作用**：
  + 返回值：解码后的 Python 对象。
  + 返回值类型：取决于 JSON 数据的结构。
* **示例**：

import json  
​  
json\_str = '{"name": "Alice", "age": 30}'  
data = json.loads(json\_str)  
print(data)

输出：

{'name': 'Alice', 'age': 30}

#### 4. json.load(fp, \*, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw)

* **参数要求和作用**：
  + fp：文件对象，用于读取 JSON 数据。
  + 其他参数与 json.loads 相同。
* **返回值和返回值的类型与作用**：
  + 返回值：解码后的 Python 对象。
  + 返回值类型：取决于 JSON 数据的结构。
* **示例**：

import json  
​  
with open('data.json', 'r') as fp:  
   data = json.load(fp)  
   print(data)

输出：

{'name': 'Alice', 'age': 30}

#### 5. json.JSONEncoder 类

* **主要方法**：
  + \_\_init\_\_(self, \*, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, sort\_keys=False, indent=None, separators=None, default=None)
  + encode(self, o)：将 Python 对象编码为 JSON 字符串。
  + iterencode(self, o)：生成编码后的 JSON 字符串的迭代器。
* **示例**：

import json  
​  
class CustomEncoder(json.JSONEncoder):  
   def default(self, obj):  
       if isinstance(obj, set):  
           return list(obj)  
       return super().default(obj)  
​  
data = {'name': 'Alice', 'tags': {'python', 'programming'}}  
json\_str = json.dumps(data, cls=CustomEncoder, indent=4)  
print(json\_str)

输出：

{  
   "name": "Alice",  
   "tags": [  
       "python",  
       "programming"  
  ]  
}

通过以上方法和类，json 模块提供了丰富的功能来处理 JSON 数据的编码和解码，满足各种复杂场景的需求