编 程 学 习 笔 记

Python

作 者 姓 名： 颜佳

目 录

[1 第一章 1](#_Toc76209995)

[1.1 第一节 1](#_Toc76209996)

[1.1.1 第一段 1](#_Toc76209997)

[1.2 1](#_Toc76209998)

[2 1](#_Toc76209999)

# Python历史

# Python标准与PEP

PEP（Python Enhancement Proposals，Python增强建议）

# 内置数据类型

## 对象

每个对象都有各自的编号、类型和值。一个对象被创建后，它的编号和类型是不会改变的。有些对象的值是可以改变的，**值可以改变的对象被称为可变对象；值不可以改变的对象被称为不可变对象**。

能够包含对其他对象的引用的对象被称为容器。这些引用是容器对象值的组成部分。在多数情况下，当谈论一个容器的值时，我们是指所包含对象的值而不是其编号；但是，当我们谈论一个容器的可变性时，则仅**指其直接包含的对象的编号**。因此，如果一个不可变容器 (例如元组) 包含对一个可变对象的引用，则当该可变对象被改变时容器的值也会改变。但是该容器仍属于不可变对象，因为它所包含的对象集是不会改变的。因此，**不可变并不严格等同于值不能改变，实际含义要更微妙。**

对于不可变类型，**会得出新值的运算**实际上会返回对相同类型和取值的任一现有对象的引用，而对于可变类型来说这是不允许的。例如在 a = 1; b = 1 之后，a 和 b 可能会也可能不会指向同一个值为一的对象，这取决于具体实现，但是在 c = []; d = [] 之后，c 和 d 保证会指向两个不同、单独的新建空列表。(请注意 c = d = [] 则是将同一个对象赋值给 c 和 d。)

## 数字类型——numbers.Number

### 整型——numbers.Integral

#### 整型——int

表示任意大小的数字，仅受限于可用的内存 (包括虚拟内存)

#### 布尔型——bool（False 和 True）

表示逻辑值 False 和 True。两个布尔值在各种场合的行为分别类似于数值 0 和 1，例外情况只有**在转换为字符串时分别返回字符串 "False" 或 "True"**。

以下对象被看做False:

* 常量None和False
* 值为0的数值类型：0， 0.0， 0j，decimal.Decimal(0)，fractions.Fraction(0, 1)
* 空的序列和容器类型：[]，()，{}，set()，frozenset()，range(0)
* 定义了\_\_bool\_\_方法且该方法返回False的类
* 没有定义\_\_bool\_\_方法，但是定义了\_\_len\_\_方法且该方法返回0的类

### 浮点型——numbers.Real (float)

表示机器级的双精度浮点数，Python 不支持单精度浮点数

### 复数——numbers.Complex (complex)

以一对机器级的双精度浮点数来表示复数值

## 序列

### 不可变序列

不可变序列类型的对象一旦创建就不能再改变。(如果对象包含对其他对象的引用，其中的可变对象就是可以改变的；但是，**一个不可变对象所直接引用的对象集**是不能改变的。)

#### 元组

一个元组中的条目可以是任意 Python 对象。包含两个或以上条目的元组**由逗号分隔的表达式**构成。只有一个条目的元组 ('单项元组') 可通过在**表达式后加一个逗号**来构成 (一个表达式本身不能创建为元组，因为圆括号要用来设置表达式分组)。一个空元组可通过一对内容为空的圆括号创建。

#### 字符串

字符串是由 Unicode 码位值组成的序列。Python 没有 char 类型；而是将字符串中的每个码位表示为一个长度为 1 的字符串对象。

1. **内置函数 ord() 可将一个码位由字符串形式转换成一个范围在 0 - 10FFFF 之内的整型数；**
2. **内置函数 chr() 可将一个范围在 0 - 10FFFF 之内的整型数转换为长度为 1 的对应字符串对象。**
3. **str.encode() 可以使用指定的文本编码将 str 转换为 bytes，**
4. **bytes.decode() 可以将bytes转换为str。**

#### 字节串

**字节串对象是不可变的数组。**字节串字面值 (例如 b'abc') 和内置的 **bytes()** 构造器可被用来创建字节串对象。其中每个条目都是一个 8 位字节，以取值范围 0 <= x < 256 的整型数表示。字节串对象还可以通过 decode() 方法解码为字符串。

### 可变序列

#### 列表

列表中的条目可以是任意 Python 对象。列表由用方括号括起并由逗号分隔的多个表达式构成。(注意创建长度为 0 或 1 的列表无需使用特殊规则。)

#### 字节数组

**字节数组对象属于可变数组。**可以通过内置的 **bytearray()** 构造器来创建。除了是可变的 (因而也是不可哈希的)，在其他方面字节数组提供的接口和功能都与不可变的 bytes 对象一致。

## 集合类型

此类对象表示**由不重复且不可变对象组成的无序且有限**的集合。因此它们不能通过下标来索引。但是它们可被迭代，也可用内置函数 len() 返回集合中的条目数。集合常见的用处是快速成员检测，去除序列中的重复项，以及进行交、并、差和对称差等数学运算。

对于集合元素所采用的不可变规则与字典的键相同。注意数字类型遵循正常的数字比较规则: 如果两个数字相等 (例如 1 和 1.0)，则同一集合中只能包含其中一个。

### 集合

此类对象表示可变集合。它们可通过内置的 set() 构造器创建，并且创建之后可以通过方法进行修改，例如 add()。

### 冰冻集合

此类对象表示不可变集合。它们可通过内置的 frozenset() 构造器创建。由于 frozenset 对象不可变且 hashable，它可以被用作另一个集合的元素或是字典的键。

## 映射

此类对象表示由任意索引集合所索引的对象的集合。通过下标 a[k] 可在映射 a 中选择索引为 k 的条目；这可以在表达式中使用，也可作为赋值或 del 语句的目标。内置函数 len() 可返回一个映射中的条目数。

### 字典

此类对象表示由几乎任意值作为索引的有限个对象的集合。不可作为键的值类型只有包含列表或字典或其他可变类型，通过值而非对象编号进行比较的值，其原因在于高效的字典实现需要使用键的哈希值以保持一致性。**用作键的数字类型遵循正常的数字比较规则: 如果两个数字相等 (例如 1 和 1.0) 则它们均可来用来索引同一个字典条目。**

字典会保留插入顺序，这意味着键将以它们被添加的顺序在字典中依次产生。 替换某个现有的键不会改变其顺序，但是移除某个键再重新插入则会将其添加到末尾而不会保留其原有位置。

扩展模块 dbm.ndbm 和 dbm.gnu 提供了额外的映射类型示例，collections 模块

## 可调用类型

### 用户定义函数

| **属性** | **含意** | **权限** |
| --- | --- | --- |
| \_\_doc\_\_ | 该函数的文档字符串，没有则为 None；不会被子类继承。 | 可写 |
| [\_\_name\_\_](https://docs.python.org/zh-cn/3/library/stdtypes.html#definition.__name__) | 该函数的名称。 | 可写 |
| [\_\_qualname\_\_](https://docs.python.org/zh-cn/3/library/stdtypes.html#definition.__qualname__) | 该函数的 [qualified name](https://docs.python.org/zh-cn/3/glossary.html#term-qualified-name)。 | 可写 |
| \_\_module\_\_ | 该函数所属模块的名称，没有则为 None。 | 可写 |
| \_\_defaults\_\_ | 由具有默认值的参数的默认参数值组成的元组，如无任何参数具有默认值则为 None。 | 可写 |
| \_\_code\_\_ | 表示编译后的函数体的代码对象。 | 可写 |
| \_\_globals\_\_ | 对存放该函数中全局变量的字典的引用 --- 函数所属模块的全局命名空间。 | 只读 |
| [\_\_dict\_\_](https://docs.python.org/zh-cn/3/library/stdtypes.html#object.__dict__) | 命名空间支持的函数属性。 | 可写 |
| \_\_closure\_\_ | None 或包含该函数可用变量的绑定的单元的元组。有关 cell\_contents 属性的详情见下。 | 只读 |
| \_\_annotations\_\_ | 包含形参标注的字典。 字典的键是形参名，而如果提供了 'return' 则是用于返回值标注。 有关如何使用此属性的更多信息，请参阅 [对象注解属性的最佳实践](https://docs.python.org/zh-cn/3/howto/annotations.html#annotations-howto)。 | 可写 |
| \_\_kwdefaults\_\_ | 仅包含关键字参数默认值的字典。 | 可写 |

### 实例方法

### 生成器函数

### 协程函数

### 异步生成器函数

### 内置函数

### 内置方法

### 类

### 类实例

## 自定义类

每个类都有通过**一个字典对象**实现的**独立命名空间**。类属性引用会被转化为在此字典中查找，例如 C.x 会被转化为 C.\_\_dict\_\_["x"]。当未在其中发现某个属性名称时，会继续在基类中查找。

| **属性** | **含意** | **权限** |
| --- | --- | --- |
| \_\_doc\_\_ | 该类的文档字符串，没有则为 None；不会被子类继承。 | 可写 |
| [\_\_name\_\_](https://docs.python.org/zh-cn/3/library/stdtypes.html#definition.__name__) | 该类的名称。 | 可写 |
| \_\_annotations\_\_ | 包含在类体执行期间收集的 变量标注 的字典。 | 可写 |
| \_\_module\_\_ | 该类所属模块的名称，没有则为 None。 | 可写 |
| [\_\_dict\_\_](https://docs.python.org/zh-cn/3/library/stdtypes.html#object.__dict__) | 包含类命名空间的字典 | 可写 |

## 模块

## 可变数据类型与不可变数据类型

### 可变数据类型：列表、集合、字典

当该数据类型的对应变量的值发生了改变，但是它对应的内存地址不发生改变，对于这种数据类型，就称可变数据类型。

### 不可变数据类型：整型、布尔类型、浮点数类型、复数、元组、字符串、冰冻集合

当该数据类型的对应变量的值发生了改变，且它对应的内存地址也会发生改变，对于这种数据类型，就称不可变数据类型。

## x += 1与x = x + 1的区别

### 对于可变类型，x = x + 1相当于x的重新赋值，x的id会改变

L = [1, 2]

M = L

L += L

print(L, M)

L = [1, 2]

M = L

L = L + L

print(L, M)

[1, 2, 1, 2] [1, 2, 1, 2]

[1, 2, 1, 2] [1, 2]

### 对于不可变类型，结果一样，都会导致x的id会改变

s = "123"

t = s

s += s

print(s, t)

s = "123"

t = s

s = s + s

print(s, t)

123123 123

123123 123

## is比较的是id是否相同，==比较的是值是否相同

### 同一类型的空不可变对象的id是相同的。

### 同一类型的空可变对象的id是不同的。

### 不同类型的空对象的id是不同的。

### 同一类型不可变对象的内容相同，则id也相同

a = ""

b = ""

print("a == b:", a == b)

print("a is b:", a is b)

a = []

b = []

print("a == b:", a == b)

print("a is b:", a is b)

a = "123"

b = "123"

print("a == b:", a == b)

print("a is b:", a is b)

a = [1, 2, 3]

b = [1, 2, 3]

print("a == b:", a == b)

print("a is b:", a is b)

a == b: True

a is b: True

a == b: True

a is b: False

a == b: True

a is b: True

a == b: True

a is b: False

# 变量

## 变量不是盒子，而是对象的标注

## 命名规则

* 变量名只能包含字母、数字和下划线。变量名只能以字母或下划线开头。
* 不要将Python关键字和函数名用作变量名

## 变量的作用域

变量的查找顺序（LEGB）：**当前作用域的局部变量>>外层作用域变量>>当前模块的全局变量>>Python的内置变量。**

在函数中，可以直接引用外部变量，但是不能改写外部变量。

### 局部变量

对数字、字符串、元组等不可变类型来说，只能读取，不能更新。如果尝试重新绑定，例如 count = count + 1，其实会隐式创建局部变量 count。这样，count 就不是自由变量了，因此不会保存在闭包中。

### nonlocal变量

nonlocal 声明的作用是**把变量标记为自由变量**，即使在函数中为变量赋予新值了，也会变成自由变量。如果为 nonlocal 声明的变量赋予新值，闭包中保存的绑定会更新。用来在函数或其他作用域中使用**外层（非全局）变量**。

nonlocal语句回去搜索本地变量与全局变量之间的变量，其会优先寻找层级关系与闭包作用域最近的外部变量。

### global变量

#### 在函数中如果使用了global声明，如果直接给全局变量重新赋值或者修改可变类型的变量，都会影响全局变量的取值

g1 = 1

g2 = []

g3 = []

def f2():

    global g1

    global g2

    global g3

    g1 = 2

    g2.append(1)

    g3 = [1, 2]

f2()

print(g1, g2, g3)

2 [1] [1, 2]

#### 在函数中如果没有使用global声明，如果直接给全局变量重新赋值，则不会影响全局变量的取值。如果修改可变类型的变量，则会影响全局变量的取值

*# 全局变量*

g1 = 1

g2 = []

g3 = []

def f1():

    g1 = 2

    g2.append(1)

    g3 = [1]

f1()

print(g1, g2, g3)

1 [1] []

## 类变量

在类中定义在函数外面的变量是类变量，不属于类的实例。类变量可以通过【类名.变量名】或者【对象.变量名】引用。类变量可以在定义类的代码段中定义，也可以在类定义完成之后定义。

### 类变量是可变数据类型（list，set）

* 在执行【对象1.变量名】 = x后，会修改【类名.变量名】和【其他对象.变量名】。
* 在执行【类名.变量名】 = x后，会修改所有的对象.变量名。

### 类变量是不可变数据类型（int float string tuple）

* 在执行【对象1.变量名】 = x后，不会修改【类名.变量名】和【其他对象.变量名】
* 在执行【类名.变量名】 = x后，不会修改之前通过【对象.变量名】修改的值，但是对于没有通过【对象.变量名】修改过类变量，则会修改为【对象.变量名】的值。

### 子类中显示定义了与父类同名的类变量

不管类变量是什么类型，修改父类的类变量时，不会影响子类的类变量，修改子列的类变量时，不会影响父类的类变量的值。

### 子类中没有显示定义与父类同名的类变量

* 在没有修改过【子类名.变量名】，则完全继承【父类.变量名】修改的值，否则【子类名.变量名】与【父类.变量名】完全独立。

# 运算符与表达式

## 运算符

## 表达式

### 条件表达式（三元运算符）

ans = 1 if 5 > 6 else 0

print(ans)

# 简单语句

<https://docs.python.org/zh-cn/3/reference/simple_stmts.html#the-assert-statement>

## 表达式语句

## 赋值语句

### 序列解包

#### 可同时给多个变量赋值

x, y, z = 1, 2, 3

#### 可交换多个变量的值

x, y = y, x

#### 解包的序列包含的元素个数必须与等号左边的目标个数相同，否则将引发异常。

#### 可使用星号运算符（\*）来收集多余的值，带星号的变量放在其他位置，带星号的变量最终包含的总是一个列表。

a, b, \*rest = [1, 2, 3, 4]

#### 如果不关心变量值，可用\_，\*\_接收函数

### 链式赋值

用于将多个变量关联到同一个值

## assert语句

## pass语句

## del语句

### del 语句按索引，而不是值从列表中移除元素。

### del 语句可以从列表中移除切片，或清空整个列表

### del 也可以用来删除整个变量，再引用 a 就会报错（直到为它赋与另一个值）

## return语句

## yield语句

## raise语句

## break语句

用于跳出最近的 for 或 while 循环。

## continue语句

## print语句

### 自定义分隔符

print("I", "wish", "to", "register", "a", "complaint", **sep="\_"**)

### 自定义结束字符串

print("Hello,", end="")

print("world! ")

## import语句

### import somemodule

### from somemodule import somefunction

### from somemodule import somefunction, anotherfunction, yetanotherfunction

### from somemodule import \*

### import somemodule as modulebais

### from somemodule import somefunction as functionbais

## global语句

## nonlocal语句

# 复合语句

## 条件判断——if语句

## 循环迭代

### while语句

### for语句

### 循环的else 子句

循环语句支持 else 子句；for 循环中，可迭代对象中的元素全部循环完毕时，或 while 循环的条件为假时，执行该子句；**break 语句终止循环时，不执行该子句。**

for x in range(3):

    if x > 3:

        print("x =", x)

        break

else:

    print("循环结束，不是break退出。")

循环结束，不是break退出。

## with语句

## try语句

try 的 else 子句在未触发异常时执行。

# 容器

Python支持一种数据结构的基本概念，名为**容器（container）**。容器基本上就是可包含其他对象的对象。容器包括序列（如列表和元组）、映射（如字典）和集合（set）

## 序列

在Python中有这样一种类型，他们的成员有序排列，并且可以通过下标偏移量来访问他们的一个或几个成员，这类Python类型统称为序列。

序列的两个特点：

* 有序排列；
* **索引操作符和切片操作符**，获取序列一个元素或者一部分序列。

### 通用序列操作

| **运算** | **结果：** |
| --- | --- |
| x in s | 如果 *s* 中的某项等于 *x* 则结果为 True，否则为 False |
| x not in s | 如果 *s* 中的某项等于 *x* 则结果为 False，否则为 True |
| s + t | *s* 与 *t* 相拼接 |
| s \* n 或 n \* s | 相当于 *s* 与自身进行 *n* 次拼接 |
| s[i] | *s* 的第 *i* 项，起始为 0 |
| s[i:j] | *s* 从 *i* 到 *j* 的切片 |
| s[i:j:k] | *s* 从 *i* 到 *j* 步长为 *k* 的切片 |
| len(s) | *s* 的长度 |
| min(s) | *s* 的最小项 |
| max(s) | *s* 的最大项 |
| s.index(x[, i[, j]]) | *x* 在 *s* 中首次出现项的索引号（索引号在 *i* 或其后且在 *j* 之前） |
| s.count(x) | *x* 在 *s* 中出现的总次数 |

### 列表——list，用[ ]表示

像字符串一样列表也是序列式的类型，可以通过下标或者切片的方式来访问元素，但**字符串只能由字符组成，而且是不可变的（不能单独改变它的某个值）**，但列表是能够保留任意数量Python对象的灵活的容器，可以修改，可以嵌套字符串，元组等其他类型。通常用于存放同类项目的集合（其中精确的相似程度将根据应用而变化）。

#### 列表特点

1. 元素可以是任意类型，一般为同质类型，可迭代访问。
2. 是一种可变的序列，所有的操作都是在原处进行修改

#### 列表构造

1. 使用一对方括号来表示空列表: []
2. 使用方括号，其中的项以逗号分隔: [a], [a, b, c]
3. 使用列表推导式: [x for x in iterable]，列表推导式的方括号内包含以下内容：一个表达式，后面为一个 for 子句，然后，是零个或多个 for 或 if 子句。结果是由表达式依据 for 和 if 子句求值计算而得出一个新列表。
4. 使用类型的构造器: list() 或 list(iterable)

构造器将构造一个列表，其中的项与 iterable 中的项具有相同的的值与顺序。 iterable 可以是**序列、支持迭代的容器或其它可迭代对象**。 如果 iterable 已经是一个列表，将创建并返回其副本，类似于 iterable[:]。 例如，list('abc') 返回 ['a', 'b', 'c'] 而 list( (1, 2, 3) ) 返回 [1, 2, 3]。 如果没有给出参数，构造器将创建一个空列表 []。

#### 列表特有操作

表格中的 s 是可变序列类型的实例，t 是任意可迭代对象，而 x 是符合对 s 所规定类型与值限制的任何对象

| **运算** | **结果** |
| --- | --- |
| s[i] = x | 将 *s* 的第 *i* 项替换为 *x* |
| s[i:j] = t | 将 *s* 从 *i* 到 *j* 的切片替换为可迭代对象 *t* 的内容 |
| del s[i:j] | 等同于 s[i:j] = [] |
| s[i:j:k] = t | 将 s[i:j:k] 的元素替换为 *t* 的元素 |
| del s[i:j:k] | 从列表中移除 s[i:j:k] 的元素 |
| s.append(x) | 将 *x* 添加到序列的末尾 (等同于 s[len(s):len(s)] = [x]) |
| s.clear() | 从 *s* 中移除所有项 (等同于 del s[:]) |
| s.copy() | 创建 *s* 的浅拷贝 (等同于 s[:]) |
| s.extend(t) 或 s += t | 用 *t* 的内容扩展 *s* (基本上等同于 s[len(s):len(s)] = t) |
| s \*= n | 使用 *s* 的内容重复 *n* 次来对其进行更新 |
| s.insert(i, x) | 在由 *i* 给出的索引位置将 *x* 插入 *s* (等同于 s[i:i] = [x]) |
| s.pop() or s.pop(i) | 提取在 *i* 位置上的项，并将其从 *s* 中移除 |
| s.remove(x) | 删除 *s* 中第一个 s[i] 等于 *x* 的项目。 |
| s.reverse() | 就地将列表中的元素逆序。 |

### 元组——tuple，用( )表示

实际上元组是与列表非常相近的一种容器类型，支持下标索引和切片操作，看上去的区别只有元组是用圆括号而列表用的是方括号，但元组和列表相比有一个非常重要的区别，**元组是一种不可变类型（这一点与字符串类似）**，元组中也可以嵌套字符串，列表等其他类型。通常用于储存异构数据的多项集。

#### 元组特点

1. 元素可以是任意类型，一般可包含异质元素序列；
2. 是一种不可变的序列

#### 元组构造

1. 使用一对圆括号来表示空元组: ()
2. 使用一个后缀的逗号来表示单元组: a, 或 (a,)
3. 使用以逗号分隔的多个项: a, b, c or (a, b, c)
4. 使用内置的 tuple(): tuple() 或 tuple(iterable)

构造器将构造一个元组，其中的项与 iterable 中的项具有相同的值与顺序。 iterable 可以是序列、支持迭代的容器或其他可迭代对象。 如果 iterable 已经是一个元组，会不加改变地将其返回。 例如，tuple('abc') 返回 ('a', 'b', 'c') 而 tuple( [1, 2, 3] ) 返回 (1, 2, 3)。 如果没有给出参数，构造器将创建一个空元组 ()。

请注意**决定生成元组的其实是逗号而不是圆括号**。 圆括号只是可选的，生成空元组或需要避免语法歧义的情况除外。 例如，f(a, b, c) 是在调用函数时附带三个参数，而 f((a, b, c)) 则是在调用函数时附带一个三元组。

元组与多数 Python 集合（列表、字典、集，等等）一样，**保存的是对象的引用**。 如果引用的元素是可变的，即便元组本身不可变，元素依然可变。也就是说，元组的不可变性其实是指 **tuple 数据结构的物理内容（即保存的引用）不可变，与引用的对象无关**。

### 字符串——str，用“”表示

Python没有专门用于表示字符的类型，因此一个字符就是只包含一个元素的字符串。Python中实际上有三种字串，str，unicode，basestring，其中str和unicode字符串都是抽象类basestring的子类，而抽象类是不能被实例化的，Python中通过单引号，双引号，以及三引号来定义str字符串，如(’ hello world’)，其中单引号和双引号的作用是相同的，在str字符串前加上一个字母u即为unicode字符串，如(u’hello world’)。

#### 字符串特点

1. 元素是字符；
2. 是一种不可变的序列

#### 字符串构造

str()

单引号: '允许包含有 "双" 引号'

双引号: "允许包含有 '单' 引号"。

三重引号: '''三重单引号''', """三重双引号"""

使用三重引号的字符串可以跨越多行 —— 其中所有的空白字符都将包含在该字符串字面值中。

单引号（'）与双引号（"）的用法比较 ：

使用单引号（'）表示的字符串中可以直接使用双引号而不必进行转义，使用双引号表示的字符串同理。

二者通常用于单行字符串的表示，也可通过使用\n换行后表示多行字符串

单引号、双引号与三单引号、三双引号的用法比较

单引号、双引号表示多行时需要添加换行符\n。

三单引号、三双引号表示多行时无需使用任何多余字符

三单引号、三双引号中可直接使用单引号、双引号而无需使用反斜杠\进行转义

#### 字符串特有方法

s.casefold

s.endswith

s.find

s.format

s.format\_map

s.isalnum

s.isalpha

s.isascii

s.isdecimal

s.isdigit

s.isidentifier

s.islower

s.isnumeric

s.isprintable

s.isspace

s.istitle

s.isupper

s.maketrans

s.partition

s.rfind

s.rindex

s.rpartition

s.startswith

| **运算** | **结果** |
| --- | --- |
| str.capitalize() | 返回原字符串的副本，其首个字符大写，其余为小写。 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

#### 字符串的格式化输出

1. **%[(name)] [flags] [width][.precision] typecode**

(name)输出字典的value使用，这里的name是字典的key（实际指定时，必须有外面的圆括号）

flags取值为：-（左对齐）、+（正负号）、0（补零），不使用任何flag数字默认是右对齐

[width]和[.precision]，分别为总宽度（包括小数点）和精度（小数点后的位数），可指定为\*，则须在后面的列表中明确指出具体数值（一般情况下很少指定width和precision为\*）

1. **format方法**
2. **f-string格式化**

## 映射

字典是Python中唯一的映射类型，它包含哈希值（键，key）和指向的对象（值，value），通过键”映射“到值，可以把本来毫无关系的两个对象建立起联系，它是一个可变的容器类型，能存储任意个Python对象，与序列不同，字典是无序的，它通过键对值进行访问，而且具有很好的性能，用键查询非常快。

### 字典——dict，用{}表示

#### 字典特点

1. 无序的可变对象；
2. 字典的键几乎可以是任何值。 非 hashable 的值，即包含**列表、字典或其他可变类型的值**（此类对象基于值而非对象标识进行比较）不可用作键。键可能是数、字符串或元组

#### 字典的构造

1. 使用一对花括号来表示空字典: {}
2. 使用花括号内以逗号分隔键: 值对的方式: {'jack': 4098, 'sjoerd': 4127} or {4098: 'jack', 4127: 'sjoerd'}
3. 使用字典推导式: {}, {x: x \*\* 2 for x in range(10)}
4. 使用类型构造器: dict(), dict([('foo', 100), ('bar', 200)]), dict(foo=100, bar=200)

如果没有给出位置参数，将创建一个空字典。 如果给出一个位置参数并且其属于映射对象，将创建一个具有与映射对象相同键值对的字典。 否则的话，位置参数必须为一个 iterable 对象。 该可迭代对象中的每一项本身必须为一个刚好包含两个元素的可迭代对象。 每一项中的第一个对象将成为新字典的一个键，第二个对象将成为其对应的值。 如果一个键出现一次以上，该键的最后一个值将成为其在新字典中对应的值。

如果给出了关键字参数，则关键字参数及其值会被加入到基于位置参数创建的字典。 如果要加入的键已存在，来自关键字参数的值将替代来自位置参数的值。

**>>>** a = dict(one=1, two=2, three=3)

**>>>** b = {'one': 1, 'two': 2, 'three': 3}

**>>>** c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))

**>>>** d = dict([('two', 2), ('one', 1), ('three', 3)])

**>>>** e = dict({'three': 3, 'one': 1, 'two': 2})

**>>>** f = dict({'one': 1, 'three': 3}, two=2)

**>>>** a == b == c == d == e == f

True

#### 字典支持的操作

| 运算 | 结果 |
| --- | --- |
| list(d) | 返回字典 d 中使用的所有键的列表。 |
| len(d) | 返回字典 d 中的项数。 |
| d[key] | 返回 d 中以 key 为键的项。 如果映射中不存在 key 则会引发 [KeyError](https://docs.python.org/zh-cn/3/library/exceptions.html#KeyError)。 |
| d[key] = value | 将 d[key] 设为 value。 |
| del d[key] | 将 d[key] 从 d 中移除。 如果映射中不存在 key 则会引发 [KeyError](https://docs.python.org/zh-cn/3/library/exceptions.html#KeyError)。 |
| key in d | 如果 d 中存在键 key 则返回 True，否则返回 False。 |
| key not in d | 等价于 not key in d。 |
| iter(d) | 返回以字典的键为元素的迭代器。 这是 iter(d.keys()) 的快捷方式。 |
| d.clear() | 移除字典中的所有元素。 |
| d.copy() | 返回原字典的浅拷贝。 |
| get(key[, default]) | 如果 key 存在于字典中则返回 key 的值，否则返回 default。 如果 default 未给出则默认为 None，因而此方法绝不会引发 [KeyError](https://docs.python.org/zh-cn/3/library/exceptions.html#KeyError)。 |
| d.items() | 返回由字典项 ((键, 值) 对) 组成的一个新视图。 |
| d.keys() | 返回由字典键组成的一个新视图。 |
| pop(key[, default]) | 如果 key 存在于字典中则将其移除并返回其值，否则返回 default。 如果 default 未给出且 key 不存在于字典中，则会引发 [KeyError](https://docs.python.org/zh-cn/3/library/exceptions.html#KeyError)。 |
| popitem() | 从字典中移除并返回一个 (键, 值) 对。 键值对会按 LIFO 的顺序被返回。 |
| reversed(d) | 返回一个逆序获取字典键的迭代器。 这是 reversed(d.keys()) 的快捷方式。 |
| setdefault(key[, default]) | 如果字典存在键 key ，返回它的值。如果不存在，插入值为 default 的键 key ，并返回 default 。 default 默认为 None。 |
| update([other]) | 使用来自 other 的键/值对更新字典，覆盖原有的键。 返回 None。 |
| d.values() | 返回由字典值组成的一个新视图。 |
| d | other | 合并 d 和 other 中的键和值来创建一个新的字典，两者必须都是字典。当 d 和 other 有相同键时， other 的值优先。 |
| d |= other | 用 other 的键和值更新字典 d ，other 可以是 [mapping](https://docs.python.org/zh-cn/3/glossary.html#term-mapping) 或 [iterable](https://docs.python.org/zh-cn/3/glossary.html#term-iterable) 的键值对。当 d 和 other 有相同键时， other 的值优先。 |

## 集合

没错，这里的集合就是数学中所谓的集合，它满足无序性、确定性、互异性，支持求交集，并集，差集等运算，十分方便。集合有两种不同类型，可变集合（set）和不可变集合（frozenset），见名知意，可变集合，即可以添加和删除元素，而不可变的则不能进行此类操作。

集合是由不重复元素组成的无序容器。基本用法包括成员检测、消除重复元素。集合对象支持合集、交集、差集、对称差分等数学运算。

### 可变集合——set

创建集合用花括号或 set() 函数。注意，创建空集合只能用set()，不能用 {}，

#### 集合特点

1. 元素可以是任意类型，但是**不能包含重复元**素；
2. 是一种**可变的无序容器**

#### 集合构造

1. 创建空集合只能用set()，不能用 {}

#### 集合的特有操作

| **运算** | **结果** |
| --- | --- |
| 交集 & | x & y，返回一个新的集合，包括同时在集合 x 和y中的共同元素。 |
| 并集 | | x | y，返回一个新的集合，包括集合 x 和 y 中所有元素。 |
| 差集 - | x - y，返回一个新的集合,包括在集合 x 中但不在集合 y 中的元素。 |
| 补集 ^ | x ^ y，返回一个新的集合，包括集合 x 和 y 的非共同元素。 |

x = set([1, 2, 3])

y = set([2, 3, 4])

print(x & y)

print(x.intersection(y))

print(x | y)

print(x.union(y))

print(x - y)

print(x.difference(y))

print(x ^ y)

print(x.symmetric\_difference(y))

print(x and y)

print(x or y)

{2, 3}

{2, 3}

{1, 2, 3, 4}

{1, 2, 3, 4}

{1}

{1}

{1, 4}

{1, 4}

{2, 3, 4}

{1, 2, 3}

当 a and b为真的时候，返回的是表达式的最后一个判断条件的值。这里是b的值

当a and b为假的时候，返回的是第一个为False的值，如果a为False，返回a的值，否则返回b的值。

当 a or b 为真的时候，返回的是第一个为True的值，如果a为True，返回 a的值，否则返回b的值。

若 a or b 为假的时候，返回的是表达式的最后一个判断条件的值，这里是b的值

### 不可变集合——frozenset

# 赋值语句、浅复制和深复制

src = [1, [2, 3], (4, 5, 6), "789"]

dst1 = src

dst2 = src[:]

dst3 = list(src)

print(dst1 is src)

print(dst1[0] is src[0])

print(dst1[1] is src[1])

print(dst1[2] is src[2])

print(dst1[3] is src[3])

print("==========================")

print(dst2 is src)

print(dst2[0] is src[0])

print(dst2[1] is src[1])

print(dst2[2] is src[2])

print(dst2[3] is src[3])

print("==========================")

print(dst3 is src)

print(dst3[0] is src[0])

print(dst3[1] is src[1])

print(dst3[2] is src[2])

print(dst3[3] is src[3])

True

True

True

True

True

==========================

False

True

True

True

True

==========================

False

True

True

True

True

## 赋值语句

赋值语句只是给同一个对象起了一个新的标识符，其id是不变的，它拷贝了对象的引用，python并没有拷贝这个对象。

## 浅复制

拷贝一个对象，但是对象的属性还是引用原来的。对于可变类型，只是复制了其引用。

### 构造方法或 [:] 做的是浅复制（即复制了最外层容器，副本中的元素是源容器中元素的引用）

### 对于可变的对象， += 运算符就地修改列表。

### 对于不可变对象来说，+= 运算符创建一个新对象，然后重新绑定给变量。

## 深复制

副本不共享内部对象的引用。

a = [1, 2, ["x", "y"]]

b = a

c = copy.copy(a)

d = copy.deepcopy(a)

a.append(3)

a[2].append("z")

a.append(["x", "y"])

print("a = ", a)

print("b = ", b)

print("c = ", c)

print("d = ", d)

a =  [1, 2, ['x', 'y', 'z'], 3, ['x', 'y']]

b =  [1, 2, ['x', 'y', 'z'], 3, ['x', 'y']]

c =  [1, 2, ['x', 'y', 'z']]

d =  [1, 2, ['x', 'y']]

# 函数

编程语言理论家把“一等对象”定义为满足下述条件的程序实体：

* 在运行时创建
* 能赋值给变量或数据结构中的元素
* 能作为参数传给函数
* 能作为函数的返回结果

## 文档字符串

## 局部变量符号表

引用变量时，首先，在局部符号表里查找变量，然后，是外层函数局部符号表，再是全局符号表，最后是内置名称符号表。因此，尽管可以引用全局变量和外层函数的变量，但最好不要在函数内直接赋值（除非是 global 语句定义的全局变量，或 nonlocal 语句定义的外层函数变量）。

## 函数参数

### 默认值参数­——在函数定义时

#### 默认值在定义作用域里的函数定义中求值

#### 默认值只计算一次。默认值为列表、字典或类实例等可变对象时，会产生与该规则不同的结果。

#### 只有参数个数相同时，才是同一个声明

### \*arguments, \*\*keywords参数­——在函数定义时

#### \*arguments形参接收一个元组，该元组包含形参列表之外的位置参数。

#### \*\*keywords形参接收一个字典，该字典包含与函数中已定义形参对应之外的所有关键字参数。

#### \*arguments必须定义在\*\*keywords的前面

### 位置参数——在函数调用时

### 关键字参数——在函数调用时

#### 关键字参数必须跟在位置参数后面

#### 不能对同一个参数多次赋值

#### 关键字参数的顺序并不重要

## 返回值

return 语句不带表达式参数时，返回 None。函数执行完毕退出也返回 None。

## 特殊参数——限制参数的传递方式

确定参数项是仅按位置、按位置或关键字，还是仅按关键字传递。

def f(pos1, pos2, /, pos\_or\_kwd, \*, kwd1, kwd2):

----------- ---------- ----------

| | |

| Positional or keyword |

| - Keyword only

-- Positional only

/ 和 \* 是可选的。这些符号表明形参如何把参数值传递给函数：位置、位置或关键字、关键字。关键字形参也叫作命名形参。

### 仅位置参数

仅限位置形参应放在 / （正斜杠）前。/ 用于在逻辑上分割仅限位置形参与其它形参。如果函数定义中没有 /，则表示没有仅限位置形参。/ 后可以是 位置或关键字 或 仅限关键字 形参。

### 仅限关键字参数

把形参标记为 仅限关键字，表明必须以关键字参数形式传递该形参，应在参数列表中第一个仅限关键字形参前添加 \*。

## lambda表达式

lambda 关键字用于创建小巧的匿名函数。

## 内置函数

### str()

### repr()

str() 和 repr() 最主要的差别在于“目标用户”。 repr() 的作用是产生机器可读的输出（大部分情况下，其输出可以作为有效的Python代码），而 str() 则产生人类可读的输出。

# 闭包、高阶函数与装饰器

## 闭包

程序被加载到内存执行时，函数定义的代码被存放在代码段中。函数被调用时，会在栈上创建其执行环境，也就是初始化其中定义的变量和外部传入的形参以便函数进行下一步的执行操作。当函数执行完成并返回函数结果后，函数栈帧便会被销毁掉。函数中的临时变量以及存储的中间计算结果都不会保留。下次调用时唯一发生变化的就是函数传入的形参可能会不一样。函数栈帧会重新初始化函数的执行环境。

在计算机科学中，闭包（Closure），又称词法闭包（Lexical Closure）或函数闭包（Function Closures），**是引用了自由变量的函数**。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。同一闭包的不同实例中引用的自由变量互相没有影响的。

只有涉及**嵌套函数**时才有闭包问题。闭包指延伸了作用域的函数，其中包含函数定义体中引用、但是不在定义体中定义的非全局变量。闭包是一种函数，它会保留定义函数时存在的自由变量的绑定，这样调用函数时，虽然定义作用域不可用了，但是仍能使用那些绑定。

## 高阶函数

接受函数为参数，或者把函数作为结果返回的函数是高阶函数。

### map(function, iterable, ...)

返回一个将function应用于iterable中每一项并输出其结果的迭代器。如果传入了额外的 iterable 参数，function 必须接受相同个数的实参并被应用于从所有可迭代对象中并行获取的项。当有多个可迭代对象时，最短的可迭代对象耗尽则整个迭代就将结束。

### filter(function, iterable)

用 iterable 中函数 function 返回真的那些元素，构建一个新的迭代器。iterable 可以是一个序列，一个支持迭代的容器，或一个迭代器。如果 function 是 None ，则会假设它是一个身份函数，即 iterable 中所有返回假的元素会被移除。请注意， filter(function, iterable) 相当于一个生成器表达式，当 function 不是 None 的时候为 **(item for item in iterable if function(item))**；function 是 None 的时候为 **(item for item in iterable if item)** 。

### reduce

1. reduce()函数返回值为一个值而不是迭代器对象，故其常用与叠加、叠乘等；
2. reduce()函数传入的函数必须有两个参数，依次从序列中取出一个元素，和上一次调用function的结果作为参数再次调用参数；
3. reduce()函数不在全局名字空间，需要通过引入 functools 模块来调用

from functools import reduce

print(map(lambda x: x \* 2, [0, 1, 2, 3]))

print(list(map(lambda x: x \* 2, [0, 1, 2, 3])))

print(filter(lambda x: x \* 2, [0, 1, 2, 3]))

print(list(filter(lambda x: x \* 2, [0, 1, 2, 3])))

print(reduce((lambda x, y: x \* y), [1, 2, 3, 5]))

<map object at 0x0000028E69DCFDC0>

[0, 2, 4, 6]

<filter object at 0x0000028E69DCFCA0>

[1, 2, 3]

30

### sorted

可选的key 参数用于提供一个函数，它会应用到各个元素上进行排序。任何单参数函数都能作为 key 参数的值。

#### 根据单词长度给一个列表排序

fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']

fruits = sorted(fruits, key=len)

print(fruits)

['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']

#### 根据反向拼写给一个单词列表排序

fruits = sorted(fruits, key=lambda word: word[::-1])

print(fruits)

['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']

## 装饰器

# 类

任何 Python 对象都可以像函数一样被调用，只需实现实例方法 \_\_call\_\_。

## 类对象与实例对象

### 类对象支持两种操作：属性引用和实例化。

#### 属性引用

类名.属性名

#### 实例化

类的实例化使用函数表示法。可以把类对象视为是返回该类的一个新实例的不带参数的函数。实例化操作会创建一个空对象。当一个类定义了 \_\_init\_\_() 方法时，类的实例化操作会自动为新创建的类实例发起调用\_\_init\_\_()。

调用类时会运行类的 \_\_new\_\_ 方法创建一个实例，然后运行 \_\_init\_\_ 方法，初始化实例，最后把实例返回给调用方。只有在\_\_new\_\_返回一个cls的实例时，才会调用—\_\_init\_\_函数。

### 实例对象支持唯一操作：属性引用

有两种有效的属性名称：**数据属性和方法**。

#### 数据属性

数据属性不需要声明；像局部变量一样，它们将在第一次被赋值时产生。

#### 方法

方法是“从属于”对象的函数。方法的特殊之处就在于实例对象会作为函数的第一个参数被传入。调用 x.f() 其实就相当于 MyClass.f(x)。当一个实例的非数据属性被引用时，将搜索实例所属的类。 如果被引用的属性名称表示一个有效的类属性中的函数对象，会通过打包（指向）查找到的实例对象和函数对象到一个抽象对象的方式来创建方法对象：这个抽象对象就是方法对象。 当附带参数列表调用方法对象时，将基于实例对象和参数列表构建一个新的参数列表，并使用这个新参数列表调用相应的函数对象。

方法的第一个参数常常被命名为 self。 这也不过就是一个约定: self 这一名称在 Python 中绝对没有特殊含义。**方法可以通过使用 self 参数的方法属性调用其他方法**:

**class** **Bag**:

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

self.data = []

**def** add(self, x):

self.data.append(x)

**def** addtwice(self, x):

self.add(x)

self.add(x)

## 类变量和实例变量

### 类变量

在类中定义在函数外面的变量是类变量，不属于类的实例。类变量可以通过【类名.变量名】或者【对象.变量名】引用。类变量可以在定义类的代码段中定义，也可以在类定义完成之后定义。

#### 类变量是可变数据类型（list，set）

* 在执行【对象1.变量名】 = x后，会修改【类名.变量名】和【其他对象.变量名】。
* 在执行【类名.变量名】 = x后，会修改所有的对象.变量名。

#### 类变量是不可变数据类型（int float string tuple）

* 在执行【对象1.变量名】 = x后，不会修改【类名.变量名】和【其他对象.变量名】
* 在执行【类名.变量名】 = x后，不会修改之前通过【对象.变量名】修改的值，但是对于没有通过【对象.变量名】修改过类变量，则会修改为【对象.变量名】的值。

#### 子类中显示定义了与父类同名的类变量

不管类变量是什么类型，修改父类的类变量时，不会影响子类的类变量，修改子列的类变量时，不会影响父类的类变量的值。

#### 子类中没有显示定义与父类同名的类变量

* 在没有修改过【子类名.变量名】，则完全继承【父类.变量名】修改的值，否则【子类名.变量名】与【父类.变量名】完全独立。

### 实例变量

实例变量用于每个实例的唯一数据，而类变量用于类的所有实例共享的属性和方法，如果同样的属性名称同时出现在实例和类中，则属性查找会优先选择实例。

## 类的成员

### 成员的命名

#### 类的私有成员属性（Private）建议使用双下划线\_\_作为名称前缀

仅允许在本类中使用此属性，不希望被外界访问。

#### 类的保护成员属性（Protected）建议使用单下划线\_作为名称前缀

仅允许本类或者子类使用。

### 静态方法与类方法

类的成员方法不需要访问实例时，需要定义为staticmethod或者classmethod

#### 静态方法

在不需要访问实例也不需要访问类属性时，通过@staticmethod定义为静态方法

#### 类方法

在不需要访问实例但是需要访问类属性时，通过@classmethod定义为类方法

### 魔法方法

魔法方法是在特定的情况下自动调用的方法。

* **重写类的魔法函数时，必须返回其原型指定的类型。**
* 未实现的数值运算类型魔法函数必须返回NotImplememted而不是抛出NotImplementedError。

#### 构造方法：\_\_new\_\_(), \_\_init\_\_(), \_\_del\_\_()

* \_\_new\_\_(cls,[...])

**对象实例化时**第一个调用的方法。

1. \_\_new\_\_至少有一个参数cls，代表要实例化的类，其他其他参数传给 \_\_init\_\_ 。
2. \_\_new\_\_必须要有返回值，返回实例化出来的实例，若\_\_new\_\_没有正确返回当前类cls的实例，那\_\_init\_\_是不会被调用的

* \_\_init\_\_(self,[...])

**实例初始化时**调用的方法。在实例 (通过 \_\_new\_\_()) 被创建之后，返回调用者之前调用。

1. 其参数与传递给类构造器表达式的参数相同。
2. 返回的值只能是 None。
3. 一个基类如果有 \_\_init\_\_() 方法，则其所派生的类如果也有 \_\_init\_\_() 方法，就必须显式地调用它以确保实例基类部分的正确初始化；例如: **super().\_\_init\_\_([args...]).**

* \_\_del\_\_(self)

实例被销毁时调用的方法，定义了当对象被垃圾回收时的行为

#### 类的表示

使用字符串来表示类是一个相当有用的特性。在Python中有一些内建方法可以返回类的表示，相对应的，也有一系列魔法方法可以用来自定义在使用这些内建函数时类的行为。

* \_\_str\_\_(self)

由内置函数 str() 以及format() 和 print() 调用以生成一个对象的“非正式”或格式良好的字符串表示。返回值必须为一个 字符串 对象。

此方法与 \_\_repr\_\_(self) 的不同点在于 \_\_str\_\_() 并不预期返回一个有效的 Python 表达式：可以使用更方便或更准确的描述信息。

* \_\_repr\_\_(self)

由内置函数 repr() 调用以输出一个对象的“官方”字符串表示。如果可能，这应类似一个有效的 Python 表达式，能被用来重建具有相同取值的对象（只要有适当的环境）。如果这不可能，则应返回形式如 <...some useful description...> 的字符串。返回值必须是一个字符串对象。

* \_\_unicode\_\_(self)

定义对类的实例调用 unicode() 时的行为。 unicode() 和 str() 很像，只是它返回unicode字符串。注意，如果调用者试图调用 str() 而你的类只实现了 \_\_unicode\_\_() ，那么类将不能正常工作。所有你应该总是定义 \_\_str\_\_() ，以防有些人没有闲情雅致来使用unicode。

* \_\_bytes\_\_(self)

通过 bytes 调用以生成一个对象的字节串表示。这应该**返回一个 bytes 对象**。

* \_\_format\_\_(self)

定义当类的实例用于新式字符串格式化时的行为，例如， “Hello, 0:abc!”.format(a) 会导致调用 a.\_\_format\_\_(“abc”) 。当定义你自己的数值类型或字符串类型时，你可能想提供某些特殊的格式化选项，这种情况下这个魔法方法会非常有用。

* \_\_hash\_\_(self)

**定义对类的实例调用 hash() 时的行为。它必须返回一个整数**，其结果会被用于字典中键的快速比较。

由内置函数 hash() 调用，用于对散列集合成员的操作，包括 set、frozenset 和 dict。 \_\_hash\_\_() 方法应该返回一个整数。 唯一需要的属性是比较相等的对象具有相同的哈希值； 建议将对象组件的哈希值混合在一起，通过将它们打包成一个元组并对元组进行哈希处理，这些值也参与了对象的比较。

1. 如果一个类没有定义 \_\_eq\_\_() 方法，那么也不应该定义 \_\_hash\_\_() 操作；
2. 如果一个类定义了 \_\_eq\_\_() ，但没有定义 \_\_hash\_\_()，则其实例将不可被用作可哈希集的项。
3. 如果一个类定义了可变对象并实现了 \_\_eq\_\_() 方法，则不应该实现 \_\_hash\_\_()，因为可哈希集的实现要求键的哈希集是不可变的（如果对象的哈希值发生改变，它将处于错误的哈希桶中）。

用户定义的类默认带有 \_\_eq\_\_() 和 \_\_hash\_\_() 方法；使用它们与任何对象（自己除外）比较必定不相等，并且 x.\_\_hash\_\_() 会返回一个恰当的值以确保 x == y 同时意味着 x is y 且 hash(x) == hash(y)。

一个类如果重载了 \_\_eq\_\_() 且没有定义 \_\_hash\_\_() 则会将其 \_\_hash\_\_() 隐式地设为 None。当一个类的 \_\_hash\_\_() 方法为 None 时，该类的实例将在一个程序尝试获取其哈希值时正确地引发 TypeError，并会在检测 isinstance(obj, collections.abc.Hashable) 时被正确地识别为不可哈希对象。

如果一个重载了 \_\_eq\_\_() 的类需要保留来自父类的 \_\_hash\_\_() 实现，则必须通过设置 \_\_hash\_\_ = <ParentClass>.\_\_hash\_\_ 来显式地告知解释器。

如果一个没有重载 \_\_eq\_\_() 的类需要去掉哈希支持，则应该在类定义中包含 \_\_hash\_\_ = None。一个自定义了 \_\_hash\_\_() 以显式地引发 TypeError 的类会被 isinstance(obj, collections.abc.Hashable) 调用错误地识别为可哈希对象。

* \_\_nonzero\_\_(self)

定义对类的实例调用 bool() 时的行为，根据你自己对类的设计，针对不同的实例，这个魔法方法应该相应地返回True或False。

* \_\_dir\_\_(self)

定义对类的实例调用 dir() 时的行为，这个方法应该**向调用者返回一个属性列表**。一般来说，没必要自己实现 \_\_dir\_\_ 。但是如果你重定义了 \_\_getattr\_\_ 或者 \_\_getattribute\_\_ （下个部分会介绍），乃至使用动态生成的属性，以实现类的交互式使用，那么这个魔法方法是必不可少的。

#### 比较操作符

* \_\_cmp\_\_(self, other)

\_\_cmp\_\_ 是所有比较魔法方法中最基础的一个，它实际上定义了所有比较操作符的行为（<，==，!=，等等）

1. **当 self < other时，返回负整数。**
2. **当 self == other 时，返回0。**
3. **当 self > other时，返回正整数。**

最好只定义你所需要的比较形式，而不是一次定义全部。如果你需要实现所有的比较形式，而且它们的判断标准类似，那么 \_\_cmp\_\_ 是一个很好的方法，可以减少代码重复，让代码更简洁。

* \_\_eq\_\_ (self, other)

定义等于操作符(==)的行为。

* \_\_ne\_\_(self, other)

定义不等于操作符(!=)的行为。

* \_\_lt\_\_(self, other)

定义小于操作符(<)的行为。

* \_\_gt\_\_(self, other)

定义大于操作符(>)的行为。

* \_\_le\_\_(self, other)

定义小于等于操作符(<)的行为。

* \_\_ge\_\_(self, other)

定义大于等于操作符(>)的行为。

### 方法成员排列顺序规则

#### \_\_new\_\_()静态方法

#### \_\_int\_\_()方法

类的实例属性可以定义在其他方法中，但是建议定义在\_\_init\_\_()方法中

#### \_\_post\_init\_\_()方法

#### 其他魔法方法

#### @property修饰的对象属性

#### @staticmethod修饰的静态方法

#### @classmethod修饰的类方法

#### 普通方法

#### 保护方法

#### 私有方法

## 继承（允许多重继承）

**class** **DerivedClassName**(BaseClassName):

<statement-1>

.

.

.

<statement-N>

**class** **DerivedClassName**(Base1, Base2, Base3):

<statement-1>

.

.

.

<statement-N>

### \_\_init\_\_()方法

#### 子类的\_\_init\_\_()方法和父类的\_\_init\_\_()方法

1. 如果子类没有定义自己的\_\_init\_\_方法，会默认执行父类的初始化函数；
2. 如果子类定义了自己的\_\_init\_\_方法：
3. 如果在子类中没有显式调用父类的初始化函数，那么父类的属性不会被初始化，即对于只定义在父类中的属性，子类无法访问。
4. 通过super()\_\_init\_\_()来显式调用父类的初始化函数。

#### 调用父类的\_\_init\_\_()方法

父类类名.\_\_init\_\_()

super.\_\_init\_\_()

### 重写方法

#### 子类重写父类方法时，建议子类的方法签名和父类保持一致。

## 抽象类

### 抽象类不能被实例化

### 继承抽象类的子列必须实现其中定义的抽象方法

# 魔法方法

## 比较方法

### 哈希值相等

两个对象的哈希值不相等，则一定内容不相等，也一定不是同一个队形。

### 比较结果相等

比较运算符==

### ID相等

# 元类与元编程

# 迭代器（Iterator）

## 定义

实现了**\_\_iter\_\_()和\_\_next\_\_()方法**的对象在Python 中被称之为迭代器。Python 的迭代器协议需要iter () 方法返回一个实现了next () 方法的迭代器对象。

## 迭代器（Iterator）和可迭代对象（Iterable）的区别

* 可以通过**for循环**获取数据的的对象都是可迭代对象（Iterable），可以通过**next()方法**获取数据的对象都是迭代器（Iterator）
* 迭代器只是在通过使用next()方法获取数据的时候才会计算，因此**迭代器比可迭代对象效率更高并在内存使用率更低**，因为可迭代对象是一次生成。
* 迭代器执行过程中无法后退到上一个或返回到指定的元素，其只能通过next()方法继续往前执行

### 使用isinstance(obj, Iterable)方法查看指定对象是否是可迭代对象

from collections import Iterable

print("10是否是可迭代对象:", isinstance((10), Iterable))

print("[]是否是可迭代对象:", isinstance([], Iterable))

print("()是否是可迭代对象:", isinstance((), Iterable))

print("{}是否是可迭代对象:", isinstance({}, Iterable))

print("字符串是否是可迭代对象:", isinstance("abcd", Iterable))

print("生成器表达式是否是可迭代对象:", isinstance((x for x in range(10)), Iterable))

10是否是可迭代对象: False

[]是否是可迭代对象: True

()是否是可迭代对象: True

{}是否是可迭代对象: True

字符串是否是可迭代对象: True

生成器表达式是否是可迭代对象: True

### 使用isinstance(obj, Iterator)方法查看指定对象是否是迭代器

from collections import Iterator

print("10是否是迭代器:", isinstance((10), Iterator))

print("[]是否是迭代器:", isinstance([], Iterator))

print("()是否是迭代器:", isinstance((), Iterator))

print("{}是否是迭代器:", isinstance({}, Iterator))

print("字符串是否是迭代器:", isinstance("abcd", Iterator))

print("生成器表达式是否是迭代器:", isinstance((x for x in range(10)), Iterator))

10是否是迭代器: False

[]是否是迭代器: False

()是否是迭代器: False

{}是否是迭代器: False

字符串是否是迭代器: False

生成器表达式是否是迭代器: True

for 语句会在容器对象上调用 iter()。 该函数返回一个定义了 \_\_next\_\_() 方法的迭代器对象，此方法将逐一访问容器中的元素。 当元素用尽时，\_\_next\_\_() 将引发 StopIteration 异常来通知终止 for 循环。 你可以使用 next() 内置函数来调用 \_\_next\_\_() 方法

# 生成器（Generator）

## 定义

带有 **yield 关键字的函数**在 Python 中被称之为生成器（Generator）。

生成器是一个用于创建迭代器的简单而强大的工具。 它们的写法类似于标准的函数，但当它们要返回数据时会使用 yield 语句（）。 每次在生成器上调用 next() 时，它会从上次离开的位置恢复执行（它会记住上次执行语句时的所有数据值）。**可以用生成器来完成的操作同样可以用基于类的迭代器来完成。 但生成器的写法更为紧凑，因为它会自动创建 \_\_iter\_\_() 和 \_\_next\_\_() 方法。**

Python 解释器会将带有 yield 关键字的函数视为一个生成器来处理。一个函数或者子程序都只能 return 一次，但是一个生成器能暂停执行并返回一个中间的结果，这就是 yield 语句的功能——返回一个中间值给调用者并暂停执行。

例子

生成器fab()的执行过程



因为fab()是一个生成器，在执行语句 f = fab(10) 时，并不会像通常的函数那样马上执行 fab() 函数的代码块，而是首先返回一个 iterable 对象！当执行到第22行时，调用了\_\_next\_\_()属性，此时才会执行 fab() 函数的代码块。 执行到第11行的语句 yield b 时，fab() 函数会返回一个迭代值，并跳出函数，执行第23行。看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。

由此可以看出，生成器通过关键字 yield 不断的将迭代器返回到内存进行处理，而不会一次性的将对象全部放入内存，从而节省内存空间。从这点看来生成器和迭代器非常相似，但如果更深入的了解的话，其实两者仍存在区别。

除了第一次循环外，在主函数中，碰到\_\_next\_\_()，就会跳到生成器的yield后面的语句执行；当碰到yield语句时，就会跳到主函数的\_\_next\_\_()语句的下一行执行。

优点

 适合用于遍历一些巨大的或是无限的类序列对象

## 生成器（Generator）与生成器表达式（Generator Expressions）

某些简单的生成器可以写成简洁的表达式代码，所用语法类似列表推导式，但外层为圆括号而非方括号。 这种表达式被设计用于生成器将立即被外层函数所使用的情况。 **生成器表达式相比完整的生成器更紧凑但较不灵活，相比等效的列表推导式则更为节省内存。**

生成器表达式是列表解析的扩展，就如上文所述：生成器是一个特定的函数，允许返回一个中间值，然后挂起代码的执行，稍后再恢复执行。列表解析的不足在于，它必须一次性生成所有的数据，用以创建列表对象，所以不适用于迭代大量的数据。

生成器表达式通过结合列表解析和生成器来解决这个问题。

· 列表解析 [expr for iter\_var in iterable if cond\_expr]

· 生成器表达式 (expr for iter\_var in iterable if cond\_expr)

两者的语法非常相似，但生成器表达式返回的不是一个列表类型对象，而是一个生成器对象，生成器是一个内存使用友好的结构。

生成器和迭代器的区别

生成器的另一个优点就是它不要求你事先准备好整个迭代过程中所有的元素，即无须将对象的所有元素都存入内存之后，才开始进行操作。生成器仅在迭代至某个元素时才会将该元素放入内存，而在这之前或之后，元素可以不存在或者被销毁。这个特点使得它特别适合用于遍历一些巨大的或是无限的类序列对象，EG. 大文件/大集合/大字典/斐波那契数列等。这个特点被称为延迟计算或惰性求值(Lazy evaluation)，可以有效的节省内存。惰性求值实际上是现实了协同程序的思想。

协同程序：是一个可以独立运行的函数调用，该调用可以被暂停或者挂起，之后还能够从程序流挂起的地方继续或重新开始。当协同程序被挂起时，Python 就能够从该协同程序中获取一个处于中间状态的属性的返回值(由 yield 返回)，当调用 next() 方法使得程序流回到协同程序中时，能够为其传入额外的或者是被改变了的参数，并且从上次挂起的下一条语句继续执行。这是一种类似于进程中断的函数调用方式。这种挂起函数调用并在返回属性中间值后，仍然能够多次继续执行的协同程序被称之为生成器。

加强的生成器特性

除了可以使用next()方法来获取下一个生成的值，用户还可以使用send()方法将一个新的或者是被修改的值返回给生成器。除此之外，还可以使用close()方法来随时退出生成器。

# 异常

# 文件操作

读写模式

1. w 写模式，不能读，打开一个已经存在的文件，清空以前的文件内容，重新写；若文件不存在则建立该文件

w+ 是读写模式，只要沾上w，肯定清空原来的文件

2. r 读模式，只能读，不能写，文件必须存在

r+ 是读写模式，只要沾上r，文件必须存在

3. a 追加模式，也能写，在文件的末尾添加内容

4. rb+、wb+、ab+，这种是二进制模式打开或者读取，一些音乐文件

w+与r+区别：

r+：可读可写，若文件不存在，报错；w+: 可读可写，若文件不存在，创建

r+与a+区别：

r+，可读写，文件必须存在，从文件开头开始写；a+，可读写，文件存在则在文件末尾追加写；文件不存在则创建文件

模式 描述

r 打开一个文件为只读。文件指针置于该文件的开头。这是默认模式。

rb 打开一个文件只能以二进制格式读取。文件指针置于该文件的开头。这是默认模式。

r+ 打开用于读取和写入文件。文件指针将会在文件的开头，写的时候文件从文件开头写入并且会覆盖相同长度的内容。

rb+ 打开用于读取和写入二进制格式的文件。文件指针将会在文件的开头。

w 打开一个文件只写。如果该文件存在，覆盖该文件。如果该文件不存在，则创建用于写入一个新的文件。

wb 打开一个文件只能以二进制格式写入。如果该文件存在，覆盖该文件。如果该文件不存在，则创建用于写入一个新的文件。

w+ 打开用于写入和读取的文件。如果文件存在，覆盖现有的文件。如果该文件不存在，则创建读取和写入新的文件。

wb+ 打开用于写入和读取的二进制格式的文件。覆盖现有的文件，如果文件存在。如果该文件不存在，则创建读取和写入新的文件。

a 将打开追加文件。文件指针是在文件的结尾。也就是说，该文件是在附加模式。如果该文件不存在，它创造了写入一个新的文件。

ab 将打开追加的二进制格式的文件。文件指针在该文件的结束。也就是说，该文件为追加模式。如果该文件不存在，它创建并写入一个新的文件。

a+ 打开为追加和读取文件。文件指针在该文件的结束。该文件将为追加模式。如果该文件不存在，它创建并读取和写入的新文件。

ab+ 打开两个追加和读取的二进制格式的文件。文件指针在该文件的结束。该文件将在追加模式。如果该文件不存在，它创建并读取和写入的新文件。

for循环的readlines方法是一次读取所有行并将所有行放在一个列表当中，read()方法是读取所有内容到内存并保存为字符串格式

seek():指定指针的位置

tell()：获取指针当前位置

read([size]):读取文件所有内容

readline([size])：每次读一行,

readlines([size]):读取所有行并返回列表，若给定sizeint>0，返回总和大约为sizeint字节的行, 实际读取值可能比 sizeint 较大, 因为需要填充缓冲区。

truncate([size])：截取文件，使文件大小为size

flush():刷新输出缓存

close():关闭打开的文件，将缓存写入磁盘文件

# 模块与包

**模块是包含 Python 定义和语句的文件**。**其文件名是模块名加后缀名 .py** 。在模块内部，通过全局变量 **\_\_name\_\_** 可以获取模块名（即字符串）。模块中的定义可以导入 到其他模块或主模块（在顶层和计算器模式下，执行脚本中可访问的变量集）

1. 对于没有定义\_\_all\_\_的模块，使用from xxx import \*时，不会导入单下划线和双下划线开头的类，函数和属性。
2. 对于定义了\_\_all\_\_的模块，仅在from xxx import \*的场景下起作用；
3. main中导入两个模块A，B中有同名的类，函数或者属性时，后导入的生效。

## 模块导入

### import somemodule

只导入模块名到当前符号表，不会将模块中定义的函数定义的名称导入，

### from somemodule import somefunction

将模块中定义的函数定义的名称导入到当前符号表，不会把模块名导入到局部符号表里

### from somemodule import somefunction, anotherfunction, yetanotherfunction

### from somemodule import \*

会导入模块中所有不以下划线（\_）开头的名称。

### import somemodule as modulebais

### from somemodule import somefunction as functionbais

## 以脚本方式执行模块

将把下列代码添加到模块末尾，既可以把这个文件当脚本使用，也可以用作导入的模块， 因为，解析命令行的代码只有在模块以 “main” 文件执行时才会运行，导入模块时，不运行这些代码。

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

**import** **sys**

fib(int(sys.argv[1]))

python fibo.py <arguments>

这项操作将执行模块里的代码，和导入模块一样，但会把 \_\_name\_\_ 赋值为 "\_\_main\_\_"。

## 模块搜索路径

导入 spam模块时，解释器首先查找名为 spam 的内置模块。如果没找到，解释器再从 sys.path 变量中的目录列表里查找 spam.py 文件。sys.path 初始化时包含以下位置：

* 输入脚本的目录（或未指定文件时的当前目录）。
* PYTHONPATH （目录列表，与 shell 变量 PATH 的语法一样）。
* 依赖于安装的默认值（按照惯例，包括一个 site-packages 目录，由 site模块处理）。

## dir() 函数

实例的\_\_dict\_\_仅存储与该实例相关的实例属性；

类的\_\_dict\_\_存储所有实例共享的属性，不包含其父类的属性。

### 内置函数 dir() 用于查找模块定义的名称。返回结果是经过排序的字符串列表

**>>> import** **fibo**

**>>>** dir(fibo)

['\_\_name\_\_', 'fib', 'fib2']

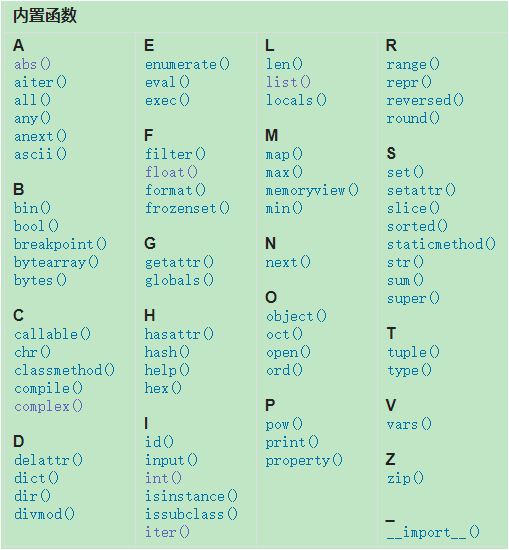
### 没有参数时，dir() 列出当列出所有类型的名称：变量、模块、函数等。

# 并行与并发设计

# 内置函数

<https://docs.python.org/zh-cn/3/library/functions.html>

Python 解释器内置了很多函数和类型，任何时候都能使用。以下按字母顺序给出列表。



# 标准库

## 函数式编程模块

### functools 模块

functools 模块应用于**高阶函数，即参数或（和）返回值为其他函数的函数**。 通常来说，此模块的功能适用于所有可调用对象。

### itertools 模块

为高效循环而创建迭代器的函数

#### 创建无穷迭代器

| **迭代器** | **实参** | **结果** |
| --- | --- | --- |
| [count()](https://docs.python.org/zh-cn/3/library/itertools.html#itertools.count) | start, [step] | start, start+step, start+2\*step, ... |
| [cycle()](https://docs.python.org/zh-cn/3/library/itertools.html#itertools.cycle) | p | p0, p1, ... plast, p0, p1, ... |
| [repeat()](https://docs.python.org/zh-cn/3/library/itertools.html#itertools.repeat) | elem [,n] | elem, elem, elem, ... 重复无限次或n次 |

| **迭代器** | **示例** |
| --- | --- |
| count(10) | 10 11 12 13 14 ... |
| cycle('ABCD') | A B C D A B C D ... |
| repeat(10, 3) | 10 10 10 |

#### 创建排列组合迭代器

| **迭代器** | **实参** | **结果** |
| --- | --- | --- |
| [product()](https://docs.python.org/zh-cn/3/library/itertools.html#itertools.product) | p, q, ... [repeat=1] | 笛卡尔积，相当于嵌套的for循环 |
| [permutations()](https://docs.python.org/zh-cn/3/library/itertools.html#itertools.permutations) | p[, r] | 长度r元组，所有可能的排列，无重复元素 |
| [combinations()](https://docs.python.org/zh-cn/3/library/itertools.html#itertools.combinations) | p, r | 长度r元组，有序，无重复元素 |
| [combinations\_with\_replacement()](https://docs.python.org/zh-cn/3/library/itertools.html#itertools.combinations_with_replacement) | p, r | 长度r元组，有序，元素可重复 |

| **例子** | **结果** |
| --- | --- |
| product('ABCD', repeat=2) | AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD |
| permutations('ABCD', 2) | AB AC AD BA BC BD CA CB CD DA DB DC |
| combinations('ABCD', 2) | AB AC AD BC BD CD |
| combinations\_with\_replacement('ABCD', 2) | AA AB AC AD BB BC BD CC CD DD |

### operator 模块

<https://docs.python.org/zh-cn/3/library/operator.html>

operator 模块提供了一套与Python的**内置运算符**对应的**高效率函数**。函数包含的种类有：**比较运算函数、逻辑运算函数、数学运算函数以及序列运算函数**。

#### 比较运算函数

对象比较函数适用于所有的对象，函数名根据它们对应的比较运算符命名。

| **运算** | **语法** | **函数** |
| --- | --- | --- |
| 比较 | a < b | lt(a, b) |
| 比较 | a <= b | le(a, b) |
| 相等 | a == b | eq(a, b) |
| 不等 | a != b | ne(a, b) |
| 比较 | a >= b | ge(a, b) |
| 比较 | a > b | gt(a, b) |

#### 逻辑运算函数

| **运算** | **语法** | **函数** |
| --- | --- | --- |
| 加法 | a + b | add(a, b) |
| 字符串拼接 | seq1 + seq2 | concat(seq1, seq2) |
| 包含测试 | obj in seq | contains(seq, obj) |
| 除法 | a / b | truediv(a, b) |
| 除法 | a // b | floordiv(a, b) |
| 按位与 | a & b | and\_(a, b) |
| 按位异或 | a ^ b | xor(a, b) |
| 按位取反 | ~ a | invert(a) |
| 按位或 | a | b | or\_(a, b) |
| 取幂 | a \*\* b | pow(a, b) |
| 标识 | a is b | is\_(a, b) |
| 标识 | a is not b | is\_not(a, b) |
| 索引赋值 | obj[k] = v | setitem(obj, k, v) |
| 索引删除 | del obj[k] | delitem(obj, k) |
| 索引取值 | obj[k] | getitem(obj, k) |
| 左移 | a << b | lshift(a, b) |
| 取模 | a % b | mod(a, b) |
| 乘法 | a \* b | mul(a, b) |
| 矩阵乘法 | a @ b | matmul(a, b) |
| 取反（算术） | - a | neg(a) |
| 取反（逻辑） | not a | not\_(a) |
| 正数 | + a | pos(a) |
| 右移 | a >> b | rshift(a, b) |
| 切片赋值 | seq[i:j] = values | setitem(seq, slice(i, j), values) |
| 切片删除 | del seq[i:j] | delitem(seq, slice(i, j)) |
| 切片取值 | seq[i:j] | getitem(seq, slice(i, j)) |
| 字符串格式化 | s % obj | mod(s, obj) |
| 减法 | a - b | sub(a, b) |
| 真值测试 | obj | truth(obj) |
| 比较 | a < b | lt(a, b) |
| 比较 | a <= b | le(a, b) |
| 相等 | a == b | eq(a, b) |
| 不等 | a != b | ne(a, b) |
| 比较 | a >= b | ge(a, b) |
| 比较 | a > b | gt(a, b) |

#### 数学运算函数

| **运算** | **语法** | **函数** |
| --- | --- | --- |
| 加法 | a + b | add(a, b) |
| 除法 | a / b | truediv(a, b) |
| 除法 | a // b | floordiv(a, b) |
| 取幂 | a \*\* b | pow(a, b) |
| 取模 | a % b | mod(a, b) |
| 乘法 | a \* b | mul(a, b) |
| 矩阵乘法 | a @ b | matmul(a, b) |
| 取反（算术） | - a | neg(a) |
| 正数 | + a | pos(a) |
| 减法 | a - b | sub(a, b) |

#### 序列运算函数

| **运算** | **语法** | **函数** |
| --- | --- | --- |
| 加法 | a + b | add(a, b) |
| 字符串拼接 | seq1 + seq2 | concat(seq1, seq2) |
| 包含测试 | obj in seq | contains(seq, obj) |
| 除法 | a / b | truediv(a, b) |
| 除法 | a // b | floordiv(a, b) |
| 按位与 | a & b | and\_(a, b) |
| 按位异或 | a ^ b | xor(a, b) |
| 按位取反 | ~ a | invert(a) |
| 按位或 | a | b | or\_(a, b) |
| 取幂 | a \*\* b | pow(a, b) |
| 标识 | a is b | is\_(a, b) |
| 标识 | a is not b | is\_not(a, b) |
| 索引赋值 | obj[k] = v | setitem(obj, k, v) |
| 索引删除 | del obj[k] | delitem(obj, k) |
| 索引取值 | obj[k] | getitem(obj, k) |
| 左移 | a << b | lshift(a, b) |
| 取模 | a % b | mod(a, b) |
| 乘法 | a \* b | mul(a, b) |
| 矩阵乘法 | a @ b | matmul(a, b) |
| 取反（算术） | - a | neg(a) |
| 取反（逻辑） | not a | not\_(a) |
| 正数 | + a | pos(a) |
| 右移 | a >> b | rshift(a, b) |
| 切片赋值 | seq[i:j] = values | setitem(seq, slice(i, j), values) |
| 切片删除 | del seq[i:j] | delitem(seq, slice(i, j)) |
| 切片取值 | seq[i:j] | getitem(seq, slice(i, j)) |
| 字符串格式化 | s % obj | mod(s, obj) |
| 减法 | a - b | sub(a, b) |
| 真值测试 | obj | truth(obj) |
| 比较 | a < b | lt(a, b) |
| 比较 | a <= b | le(a, b) |
| 相等 | a == b | eq(a, b) |
| 不等 | a != b | ne(a, b) |
| 比较 | a >= b | ge(a, b) |
| 比较 | a > b | gt(a, b) |

## 操作系统接口

### os模块——提供了许多与操作系统交互的函数

一定要使用 import os 而不是 from os import \* 。这将避免内建的 open() 函数被 os.open() 隐式替换掉，

### shutil模块

对于日常文件和目录管理任务， shutil模块提供了更易于使用的更高级别的接口

## 文件通配符

### glob模块

glob模块提供了一个在目录中使用通配符搜索创建文件列表的函数:

**>>> import** **glob**

**>>>** glob.glob('\*.py')

['primes.py', 'random.py', 'quote.py']

## 命令行参数

### sys模块

#### sys.argv

通用实用程序脚本通常需要处理命令行参数。这些参数作为列表存储在 sys模块的 argv 属性中。

**>>> import** **sys**

**>>>** print(sys.argv)

['demo.py', 'one', 'two', 'three']

#### sys.ps1 和 sys.ps2

用作主提示符和辅助提示符

**>>> import** **sys**

**>>>** sys.ps1

'>>> '

**>>>** sys.ps2

'... '

**>>>** sys.ps1 = 'C> '

C> print('Yuck!')

Yuck!

C>

#### sys.path

sys.path 是字符串列表，用于确定解释器的模块搜索路径。该变量以环境变量 PYTHONPATH 提取的默认路径进行初始化，如未设置 PYTHONPATH，则使用内置的默认路径。可以用标准列表操作修改该变量

#### sys.stdin，sys.stdout和sys.stderr

#### sys.exit()——程序终止

### argparse模块

## 字符串模式匹配

### re模块——为高级字符串处理提供正则表达式工具

#### 基本概念

元字符，模式字符串和被搜索的字符串

1. 特殊字符

| 运算 | 结果 |
| --- | --- |
| \d | 匹配所有的十进制数字，0-9 |
| \D | 匹配所有的非数字，包含下划线 |
| \s | 匹配所有空白字符（空格、TAB等） |
| \S | 匹配所有非空白字符，包含下划线 |
| \w | 匹配所有字母、汉字、数字，a-z A-Z 0-9 |
| \W | 匹配所有非字母、汉字、数字，包含下划线 |

1. 元字符

| 运算 | 结果 |
| --- | --- |
| . | 匹配除换行符外的任意的1个字符 |
| ^ | 匹配字符串的开始 |
| $ | 匹配字符串的结尾 |
| \* | 前面的字符可以出现0次或多次（0~无限） |
| + | 前面的字符可以出现1次或多次（1~无限） |
| ? | 前面的字符可以出现0次或1次 |
| | | 匹配前面一项或者而后面一项 |
| [] | 匹配其中一个字符 |
| {} | 用于标记前面的字符出现的频率 |

#### 模块函数

重要的应用程序大多会在使用前先编译正则表达式。

| **运算** | **结果** |
| --- | --- |
| re.compile(pattern, flags=0) | 返回正则表达式对象，re.Pattern |
| re.search(pattern, string, flags=0) | 扫描**整个字符串**找到匹配样式的第一个位置，并返回一个相应的匹配对象，re.Match。如果没有匹配，就返回一个None |
| re.match(pattern, string, flags=0) | 如果 string 开始的0或者多个字符匹配到了正则表达式样式，就返回一个相应的匹配对象，re.Match。如果没有匹配，就返回None。**只匹配字符串的开始位置** |
| re.fullmatch(pattern, string, flags=0) | 如果**整个string 匹配到正则表达式样式**，就返回一个相应的 匹配对象 。 否则就返回一个None |
| re.split(pattern, string, maxsplit=0, flags=0) | 用 pattern 分开 string 。 如果在 pattern 中捕获到括号，那么所有的组里的文字也会包含在列表里。如果 maxsplit 非零， 最多进行 maxsplit 次分隔， 剩下的字符全部返回到列表的最后一个元素。 |
| re.findall(pattern, string, flags=0) | 找到正则匹配的所有子字符串，并将它们作为列表返回。 |
| re.finditer(pattern, string, flags=0) | 找到正则匹配的所有子字符串，并将它们返回为一个 iterator。 |

#### 正则表达式对象——re.Pattern

#### 匹配对象——re.Match

匹配对象总是有一个布尔值 True。如果没有匹配的话 [match()](https://docs.python.org/zh-cn/3/library/re.html#re.Pattern.match) 和 [search()](https://docs.python.org/zh-cn/3/library/re.html#re.Pattern.search) 返回 None 所以你可以简单的用 if 语句来判断是否匹配

match = re.search(pattern, string)

**if** match:

process(match)

| **运算** | **结果** |
| --- | --- |
| Match.group([group1, ...]) | 返回正则匹配的字符串。如果只有一个参数，结果就是一个字符串，如果有多个参数，结果就是一个元组（每个参数对应一个项） |
| Match.groups(default=None) | 返回一个元组，包含所有匹配的子组，在样式中出现的从1到任意多的组合。 default 参数用于不参与匹配的情况，默认为 None。 |
| Match.start([group]) | 返回 group 匹配到的字串的开始标号。 |
| Match.end([group]) | 返回 group 匹配到的字串的结束标号。 |
| Match.span([group]) | 对于一个匹配 m ， 返回一个二元组 (m.start(group), m.end(group)) 。 注意如果 group 没有在这个匹配中，就返回 (-1, -1) 。group 默认为0，就是整个匹配。 |

## 数学

### math模块——提供对浮点数学的底层C库函数的访问

### random模块——提供了进行随机选择的工具

### statistics模块——计算数值数据的基本统计属性（均值，中位数，方差等）

### decimal模块——提供了一种 Decimal 数据类型用于十进制浮点运算。

相比内置的 float 二进制浮点实现，该类特别适用于

* 财务应用和其他需要精确十进制表示的用途，
* 控制精度，
* 控制四舍五入以满足法律或监管要求，
* 跟踪有效小数位
* 用户期望结果与手工完成的计算相匹配的应用程序。

## 互联网访问

### urllib.request——用于从URL检索数据

### smtplib——用于发送邮件

## 日期和时间

### datetime模块——提供了以简单和复杂的方式操作日期和时间的类。

虽然支持日期和时间算法，但实现的重点是有效的成员提取以进行输出格式化和操作。该模块还支持可感知时区的对象。

## 多线程

### threading模块——提供了多个同步操作原语，包括线程锁、事件、条件变量

实现多任务协作的首选方法是将所有对资源的请求集中到一个线程中，然后使用 queue模块向该线程供应来自其他线程的请求。 应用程序使用 Queue 对象进行线程间通信和协调，更易于设计，更易读，更可靠。

## 日志记录

### logging模块——提供功能齐全且灵活的日志记录系统

**import** **logging**

logging.debug('Debugging information')

logging.info('Informational message')

logging.warning('Warning:config file **%s** not found','server.conf')

logging.error('Error occurred')

logging.critical('Critical error -- shutting down')

## 用于操作列表的工具

### array模块——array()

array模块提供了一种 array() 对象，它类似于列表，但只能**存储类型一致的数据**且**存储密集更高**。此模块定义了一种对象类型，可以紧凑地表示基本类型值的数组：字符、整数、浮点数等。 数组属于序列类型，其行为与列表非常相似，不同之处在于其中存储的对象类型是受限的。 类型在对象创建时使用单个字符的 类型码 来指定。

### collections模块

该模块实现了特定目标的容器，以提供Python标准内建容器dict, list, set和tuple 的替代选择。

#### deque对象——队列，保留最后N 个元素

deque（double-ended queue，双向队列）是由栈或者队列生成的，类似于列表，但从左端添加和弹出的速度较快，而在中间查找的速度较慢，索引访问在两端的复杂度均为 O(1) 但在中间则会低至 O(n)。 如需快速随机访问，请改用列表。。此种对象适用于**实现队列和广度优先树搜索**。

| s.append(x) | 添加x到右端。 |
| --- | --- |
| s.appendleft(x) | 添加x到左端。 |
| s.clear() | 移除所有元素，使其长度为0. |
| s.copy() | 创建一份浅拷贝。 |
| s.count(x) | 计算deque中元素等于x的个数。 |
| s.extend(iterable) | 扩展deque的右侧，通过添加iterable参数中的元素。 |
| s.extendleft(iterable) | 扩展deque的左侧，通过添加iterable参数中的元素。注意，左添加时，在结果中iterable参数中的顺序将被反过来添加。 |
| s.index(x[,start[,stop]]) | 返回x在deque中的位置（在索引start之后，索引stop之前）。返回第一个匹配项，如果未找到则引发ValueError。 |
| s.insert(i,x) | 在位置i插入x。如果插入会导致一个限长deque超出长度maxlen的话，就引发一个IndexError。 |
| s.pop() | 移去并且返回一个元素，deque最右侧的那一个。如果没有元素的话，就引发一个[IndexError](https://docs.python.org/zh-cn/3/library/exceptions.html#IndexError)。 |
| s.popleft() | 移去并且返回一个元素，deque最左侧的那一个。如果没有元素的话，就引发[IndexError](https://docs.python.org/zh-cn/3/library/exceptions.html#IndexError)。 |
| s.remove(value) | 移除找到的第一个value。如果没有的话就引发[ValueError](https://docs.python.org/zh-cn/3/library/exceptions.html#ValueError)。 |
| s.reverse() | 将deque逆序排列。返回None。 |
| s.rotate(n=1) | 向右循环移动n步。如果n是负数，就向左循环。如果deque不是空的，向右循环移动一步就等价于d.appendleft(d.pop())，向左循环一步就等价于d.append(d.popleft())。 |
| s.maxlen | 只读，Deque的最大尺寸，如果没有限定的话就是None 。 |

#### defaultdict 对象

defaultdict 是内置 dict 类的子类，返回一个新的类似字典的对象。本对象包含一个名为 **default\_factory** 的属性，构造时，第一个参数用于为该属性提供初始值，默认为 None。所有其他参数（包括关键字参数）都相当于传递给 dict 的构造函数。

1. 使用 list 作为 default\_factory，很轻松地将（键-值对组成的）序列转换为（**键-列表组成的**）字典

**>>>** s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

**>>>** d = defaultdict(list)

**>>> for** k, v **in** s:

**...**  d[k].append(v)

**...**

**>>>** sorted(d.items())

[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

当每个键第一次遇见时，它还没有在字典里面，所以自动创建该条目，即调用 default\_factory 方法，返回一个空的 list。 list.append() 操作添加值到这个新的列表里。当再次存取该键时，就正常操作。

1. 设置 default\_factory 为 int，使 defaultdict 用于计数

**>>>** s = 'mississippi'

**>>>** d = defaultdict(int)

**>>> for** k **in** s:

**...**  d[k] += 1

**...**

**>>>** sorted(d.items())

[('i', 4), ('m', 1), ('p', 2), ('s', 4)]

当一个字母首次遇到时，它会查询失败，则 default\_factory 会调用 int() 来提供一个整数 0 作为默认值。后续的自增操作建立起对每个字母的计数。

1. 设置 default\_factory 为 set 使 defaultdict 用于构建 set 集合

**>>>** s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]

**>>>** d = defaultdict(set)

**>>> for** k, v **in** s:

**...**  d[k].add(v)

**...**

**>>>** sorted(d.items())

[('blue', {2, 4}), ('red', {1, 3})]

#### namedtuple()对象

返回一个新的元组子类，名为 typename 。这个新的子类用于创建类元组的对象，可以通过字段名来获取属性值，同样也可以通过索引和迭代获取值。子类实例同样有文档字符串（类名和字段名）另外一个有用的 \_\_repr\_\_() 方法，以 name=value 格式列明了元组内容。

### heapq 模块——实现了堆队列算法，也称为优先队列算法。

堆是一个二叉树，它的每个父节点的值都只会小于或等于所有孩子节点（的值）。 它使用了数组来实现：从零开始计数，对于所有的 k ，都有 heap[k] <= heap[2\*k+1] 和 heap[k] <= heap[2\*k+2]。 为了便于比较，不存在的元素被认为是无限大。 堆最有趣的特性在于最小的元素总是在根结点：heap[0]。要创建一个堆，可以使用list来初始化为 [] ，或者你可以通过一个函数 heapify() ，来把一个list转换成堆。

| heapq.heapify(x) | 将list x 转换成堆，原地，线性时间内。 |
| --- | --- |
| heapq.heappush(heap, item) | 将 item 的值加入 heap 中，保持堆的不变性。 |
| heapq.heappop(heap) | 弹出并返回 heap 的最小的元素，保持堆的不变性。如果堆为空，抛出 IndexError 。使用 heap[0] ，可以只访问最小的元素而不弹出它。 |
| heapq.heappushpop(heap, item) | 将 item 放入堆中，然后弹出并返回 heap 的最小元素。该组合操作比先调用 heappush() 再调用 heappop() 运行起来更有效率。 |
| heapq.heapreplace(heap, item) | 弹出并返回 heap 中最小的一项，同时推入新的 item。 堆的大小不变。 如果堆为空则引发 IndexError。  这个单步骤操作比 heappop() 加 heappush() 更高效，并且在使用固定大小的堆时更为适宜。 pop/push 组合总是会从堆中返回一个元素并将其替换为 item。  返回的值可能会比添加的 item 更大。 如果不希望如此，可考虑改用 heappushpop()。 它的 push/pop 组合会返回两个值中较小的一个，将较大的值留在堆中。 |
| heapq.merge(\*iterables, key=None, reverse=False) | 将多个已排序的输入合并为一个已排序的输出（例如，合并来自多个日志文件的带时间戳的条目）。 返回已排序值的 iterator。 |
| heapq.nlargest(n, iterable, key=None) | 从 iterable 所定义的数据集中返回前 n 个最大元素组成的列表。 如果提供了 key 则其应指定一个单参数的函数，用于从 iterable 的每个元素中提取比较键 (例如 key=str.lower)。 等价于: sorted(iterable, key=key, reverse=True)[:n]。 |
| heapq.nsmallest(n, iterable, key=None) | 返回x在deque中的位置（在索引start之后，索引stop之前）。返回第一个匹配项，如果未找到则引发ValueError。 |

heapq.nlargest(n, iterable, key=None)和heapq.nsmallest(n, iterable, key=None)函数

* 当 n==1 时，使用内置的min() 和max() 函数会更有效率。
* 当 n==len(iterable) 时，使用内置的sorted() 函数会更有效率。。
* 在 n 值较小时性能最好。

### bisect模块——具有用于操作有序列表的函数

## 数据压缩

zlib

gzip

bz2

lzma

zipfile

tarfile

## 性能测量

### timeit模块——测量同一问题的不同方法的相对性能

## 质量控制

### doctest质量控制

### unittest模块

## 格式化输出

### reprlib模块——用于缩略显示大型或深层嵌套的容器对象

### pprint模块——提供了更加复杂的打印控制

### textwrap模块——格式化文本段落，以适应给定的屏幕宽度

### locale模块——处理与特定地域文化相关的数据格式

## 类型检查——typing模块

### typing模块的作用：

1. 类型检查，防止运行时出现参数和返回值类型不符合。
2. 作为开发文档附加说明，方便使用者调用时传入和返回参数类型。
3. 该模块加入后并不会影响程序的运行，不会报正式的错误，只有提醒。

### typing模块使用方式：

from typing import List, Tuple, Dict

def add(a: int, string: str, f: float, b: bool) -> Tuple[List, Tuple, Dict, bool]:

    list1 = list(range(a))

    tup = (string, string, string)

    d = {"a": f}

    bl = b

    return list1, tup, d, bl

print(add(5, "hhhh", 2.3, False))

#### 基本使用

* 在传入参数时通过“参数名:类型”的形式声明参数的类型；
* 返回结果通过"–> 结果类型"的形式声明结果的类型。

#### 复合类型

from typing import List, Tuple, Dict, Set, FrozenSet, Optional, Union

* List[str]，表示元素是str的列表。
* Tuple[str, int] ，表示第一个元素是str，第二个元素是int的元组。Tuple[int, ...] 表示元素是int的变长元组。
* Dict[str, int]，表示键为str, 值为int的字典。
* Optional[str]，表示类型是str或者None。
* Union[str, int, bool]，表示类型是str或者int或者bool，Optional是一个特例。

#### 函数类型——Callable

from typing import Callable

def add(a: int, b: int) -> int:

    return a + b

def apply(fn: Callable[[int, int], int], \*args: int) -> int:

    return fn(args[0], args[1])

* Callable [[int, int], int]，表示类型是一个有两个参数为int返回值是int的函数。

#### 枚举类型——Literal

from typing import Literal

MODE = Literal['r', 'rb', 'w', 'wb']

def open\_helper(file: str, mode: MODE) -> str:

    pass

open\_helper('/some/path', 'r')  *# 正确*

open\_helper('/other/path', 'typo')  *# pyright 报错*

#### 新类型

from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)

#### 类型别名

## 互联网数据处理

### json——json编码与解码器

JSON（JavaScript Object Notation，JS对象标记）是一种轻量级的数据交换格式。具有数据格式简单，读写方便易懂等很多优点。相对于 XML 来说，更加的轻量级，更方便解析，因此许多开发者都遵循 Json 格式来进行数据的传输和交换。Json 的数据格式其实就是 Python 里面的字典格式，里面可以包含方括号括起来的数组，也就是 Python 里面的列表。

| **运算** | **结果** |
| --- | --- |
| json.dump(obj, fp) | 将dict数据转化成json数据后写入json文件 |
| json.dumps(obj) | 将dict数据转化成json数据（Python里是str类型） |
| json.load(fp) | 读取json文件数据，转成dict数据 |
| json.loads(string) | 将json数据转化成dict数据（Python里是dict类型） |
| json.JSONDecoder() | 简单的JSON解码器 |
| json.JSONEncoder() | 用于Python数据结构的可扩展JSON编码器。 |

### xml.etree.ElementTree模块——解析和创建XML数据

用于处理XML的Python接口分组在 xml 包中。

XML 处理子模块包括:

1. xml.etree.ElementTree： ElementTree API，一个简单而轻量级的XML处理器
2. xml.dom：DOM API 定义
3. xml.dom.minidom：最小的 DOM 实现
4. xml.dom.pulldom：支持构建部分 DOM 树
5. xml.sax：SAX2 基类和便利函数
6. xml.parsers.expat：Expat解析器绑定

XML 是一种继承性的分层数据格式，最自然的表示方法是使用树。 为此， ET 有两个类——**ElementTree** 将整个XML文档表示为一个树， **Element** 表示该树中的单个节点。 与整个文档的交互（读写文件）通常在 ElementTree 级别完成。 与单个 XML 元素及其子元素的交互是在 Element 级别完成的。

#### Element

属性

| **属性** | **结果** |
| --- | --- |
| tag | 将dict数据转化成json数据后写入json文件 |
| text | 存放元素的开始标记及其第一个子元素或结束标记之间的文本，或者为 None |
| tail | 而 tail 属性会存放元素的结束标记及下一个标记之间的文本，或者为 None |
| attrib | 将json数据转化成dict数据（Python里是dict类型） |

#### ElementTree

## 调试和分析

### pdb --- Python 的调试器

#### pdb的进入方法

* 非侵入式方法（不用额外修改源代码，在命令行下直接运行就能调试）

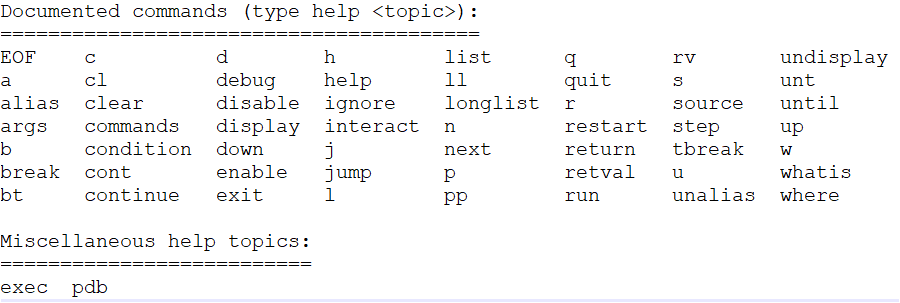
python3 -m pdb filename.py

* 侵入式方法（需要在被调试的代码中添加一行代码，然后再正常运行代码）

import pdb; pdb.set\_trace()

在代码里需要调试的地方放一个pdb.set\_trace()，就设置了一个断点，程序会在pdb.set\_trace()暂停并进入pdb调试环境，可以用pdb 变量名查看变量，或者c继续运行

#### 常用命令行



* h：（help）帮助
* w：（where）打印当前执行堆栈
* d：（down）执行跳转到在当前堆栈的深一层
* u：（up）执行跳转到当前堆栈的上一层
* b：（break）添加断点

b 列出当前所有断点，和断点执行到统计次数

b line\_no：当前脚本的line\_no行添加断点

b filename:line\_no：脚本filename的line\_no行添加断点

b function：在函数function的第一条可执行语句处添加断点

* tbreak：（temporary break）临时断点

在第一次执行到这个断点之后，就自动删除这个断点，用法和b一样

* cl：（clear）清除断点

cl 清除所有断点

cl bpnumber1 bpnumber2... 清除断点号为bpnumber1,bpnumber2...的断点

cl lineno 清除当前脚本lineno行的断点

cl filename:line\_no 清除脚本filename的line\_no行的断点

* disable：停用断点，参数为bpnumber，和cl的区别是，断点依然存在，只是不启用
* enable：激活断点，参数为bpnumber
* s：（step）执行下一条命令

如果本句是函数调用，则s会执行到函数的第一句

* n：（next）执行下一条语句

如果本句是函数调用，则执行函数，接着执行当前执行语句的下一条。

* r：（return）执行当前运行函数到结束
* c：（continue）继续执行，直到遇到下一条断点
* l：（list）列出源码

l 列出当前执行语句周围11条代码

l first 列出first行周围11条代码

l first second 列出first--second范围的代码，如果second < first，second将被解析为行数

* a：（args）列出当前执行函数的函数
* p expression：（print）输出expression的值
* pp expression：好看一点的p expression
* run：重新启动debug，相当于restart
* q：（quit）退出debug
* j lineno：（jump）设置下条执行的语句函数

只能在堆栈的最底层跳转，向后重新执行，向前可直接执行到行号

* unt：（until）执行到下一行（跳出循环），或者当前堆栈结束
* condition bpnumber conditon，给断点设置条件，当参数condition返回True的时候bpnumber断点有效，否则bpnumber断点无效

注意：

1：直接输入Enter，会执行上一条命令；

2：输入PDB不认识的命令，PDB会把他当做Python语句在当前环境下执行；

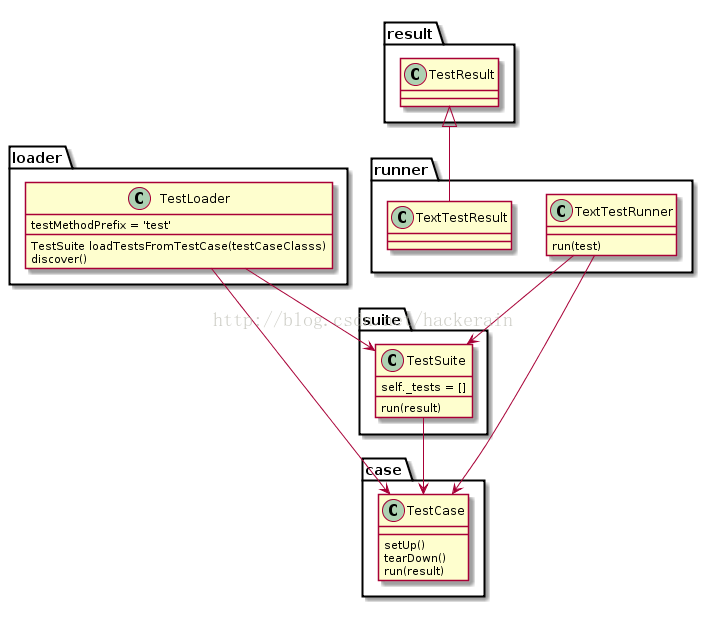
# 单元测试

## unittest

unittest 模块提供了一系列创建和运行测试的工具。

### 基本组件

一个完整的测试流程，包括测试前准备环境的搭建(setUp)，执行测试代码(run)，以及测试后环境的还原(tearDown)。



#### test fixture

test fixture 表示为了开展一项或多项测试所需要进行的**准备工作（setUp）**，以及所有相关的**清理操作（tearDown）**。举个例子，这可能包含创建临时或代理的数据库、目录，再或者启动一个服务器进程。

#### test case

一个test case是一个独立的测试单元。它检查输入特定的数据时的响应。 unittest 提供一个基类：TestCase，用于新建测试用例。

#### test suite

test suite 是一系列的test case，或test suite，或两者皆有。它用于**归档**需要一起执行的测试。

#### test runner

test runner 是一个用于执行和输出测试结果的组件。这个运行器可能使用图形接口、文本接口，或返回一个特定的值表示运行测试的结果。

继承 unittest.TestCase 就创建了一个测试样例。上述三个独立的测试是三个类的方法，这些方法的命名都以 test 开头。 这个命名约定告诉测试运行者类的哪些方法表示测试。

### TestCase类

#### 测试结果的判断方法

| 运算 | 结果 |
| --- | --- |
| assertEqual() |  |
| assertNotEqual |  |
| assertTrue |  |
| assertFalse |  |
| assertIs |  |
| assertIsNot |  |
| assertIsNone |  |
| assertIsNotNone |  |
| assertIn |  |
| assertNotIn |  |
| assertIsInstance |  |
| assertNotIsInstance |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

#### 基本测试用例

import unittest

class MyTest(unittest.TestCase):

*# 类的初始化*

    @classmethod

    def setUpClass(cls) -> None:

        print("set up class")

        return super().setUpClass()

*# 类的释放*

    @classmethod

    def tearDownClass(cls) -> None:

        print("tear down class")

        return super().tearDownClass()

*# 测试用例的初始化*

    def setUp(self) -> None:

        print("set up")

        return super().setUp()

*# 测试用例的释放*

    def tearDown(self) -> None:

        print("tear down")

        return super().tearDown()

*# 自定义函数*

    def add(self, num1, num2):

        return num1 + num2

*# 测试用例1*

    def test\_a(self):

        self.assertTrue('FOO'.isupper())

        self.assertFalse('Foo'.isupper())

*# 测试用例2*

    def test\_b(self):

        num\_sum = self.add(1, 2)

        print("num\_sum =", num\_sum)

        self.assertEqual(num\_sum, 3)

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

    unittest.main(verbosity=2)

* def setUpClass(cls) -> None 用于类的初始化，在所有测试用例执行之前执行，且只会执行一次；
* def tearDownClass(cls) -> None 用于类的释放，在所有测试用例执行之后执行，且只会执行一次；
* def setUp(self) -> None: 用于测试用例的初始化，在每个测试用例执行之前执行，有多少个执行用例就会执行多少次；
* def tearDown(self) -> None: 用于测试用例的释放，在每个测试用例执行之后进行，有多少个执行用例就会执行多少次；
* 上述四个的方法名使固定的，且测试用例要以test\_开头命名，这样的测试用例会自动执行。
* 在main()函数中，调用unittest.main(verbosity=2)运行所有内容。

### TestSuite与TextTestRunner

用于管理测试用例，调整测试顺序等。

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

*# 添加测试用例到测试套件*

    suite = unittest.TestSuite()

*# 方法1：多次调用addTest*

    suite.addTest(MyTest("test\_b"))

    suite.addTest(MyTest("test\_a"))

*# 方法2：批量添加用例，调用addTests*

    cases = [MyTest("test\_b"), MyTest("test\_a")]

    suite.addTests(cases)

*# 方法3：从文件中加载*

*# discover = unittest.defaultTestLoader.discover("./", pattern="s\*.py")*

*# runner = unittest.TextTestRunner()*

*# runner.run(discover)*

*# 运行*

    runner = unittest.TextTestRunner()

runner.run(suite)

### unittest.skip()

## ddt（Data-Driver tests）

### @data：用于设定参数

### @unpack：用于解析参数

### @file\_data：用于从文件中读取参数

import unittest

from ddt import ddt, data, unpack, file\_data

@ddt

class MyDdtTest(unittest.TestCase):

*# 类的初始化*

    @classmethod

    def setUpClass(cls) -> None:

        print("set up class")

        return super().setUpClass()

*# 类的释放*

    @classmethod

    def tearDownClass(cls) -> None:

        print("tear down class")

        return super().tearDownClass()

*# 测试用例的初始化*

    def setUp(self) -> None:

        print("set up")

        return super().setUp()

*# 测试用例的释放*

    def tearDown(self) -> None:

        print("tear down")

        return super().tearDown()

*# 自定义函数*

    def add(self, num1, num2):

        return num1 + num2

*# 测试用例1*

    @data(1, 2, 3)

    def test\_1(self, num):

        print("num =", num)

*# 测试用例2*

    @data((1, 2), (3, 4))

    @unpack

    def test\_2(self, num1, num2):

        print("num1 =", num1, end=", ")

        print("num2 =", num2, end=", ")

        print("num1 + num2 =", self.add(num1, num2), end="\n")

*# 测试用例3*

    @file\_data("ddt\_data.yml")

    def test\_3(self, \*\*kwargs):

        print("name =", kwargs["name"])

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

    unittest.main(verbosity=2)

# 浮点运算与decimal模块

## round()四舍五入

### 对于精确存储的数字，四舍五入，奇进偶舍

### 对于非精确存储的数字，四舍五入，奇偶不定。

## floor()向下取整

## ceil()向上取整

## decimal模块

### 使用字符串构造小数而不是浮点数。

# IDE（集成开发环境）

## Visual Studio Code

### 插件

#### pyright

## Pycharm

### 插件

#### 汉化——Chinese (Simplified) Language Pack / 中文语言包

#### 翻译——Translation

#### 括号显示——Rainbow Brachets

#### CSV显示——Rainbow CSV

#### 层级显示——Indent Rainbow

#### 代码统计——statistic

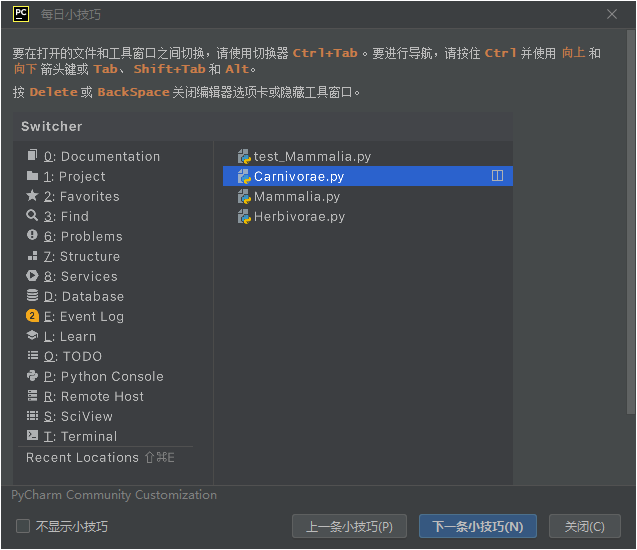
#### 主题——Material Theme UI

#### 自动补全——Tabnine

#### 格式化json——Json Parser

### 快捷键

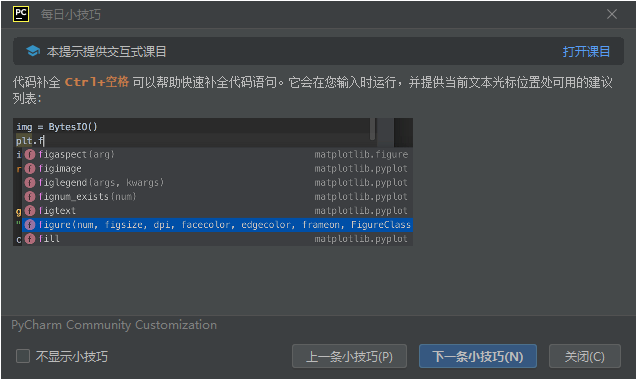
|  |  |  |
| --- | --- | --- |
| 作用 | | 快捷键 |
| 代码优化 | 优化导入 | Ctrl + Alt + O |
| 代码格式化 | Ctrl + Alt + L |
| 显示格式化配置 | Ctrl + Alt + Shift + L |
| 自动缩进行 | Ctrl + Alt + I |
| 包装代码 | Ctrl + Alt + T |
| 解开包装代码 | Ctrl + Shift + Delete |
| 合并多行代码到一行 | Crtl + Shift + J |
| 查找与替换 | 在当前文件中查找 | Ctrl + F |
| 在当前文件中替换 | Ctrl + R |
| 全局查找 | Ctrl + Shift + F |
| 全局替换 | Ctrl + Shift + R |
| 下一个匹配项 | F3 |
| 上一个匹配项 | Shift + F3 |
| 导航 | 查找所有 | Double Shift |
| 查找类 | Ctrl + N |
| 查找文件 | Ctrl + Shift + N |
| 查找符号 | Ctrl + Alt + Shift + N |
| 运行 | 运行 | Shift + F10 |
| 调试 | Shift + F9 |
| 运行模式配置 | Alt + Shift + F10 |
| 调试模式配置 | Alt + Shift + F9 |
| 调试 | 单步调试 | F8 |
| 进入内部 | F7 |
| 退出 | Shift + F8 |
| 在当前行加上断点 | Ctrl + F |
| 查看所有断点 | Ctrl + Shift + F8 |
| 注释 | 行注释，取消行注释 | Ctrl + / |
| 块注释，取消块注释 | Ctrl+Shift+/ |
| 跳转 | 跳转到指定行 | Ctrl + G |
| 跳转到变量、函数定义 | Ctrl + B / Ctrl + 鼠标左键 |
| 跳转到变量、函数使用 | Ctrl + B / Ctrl + 鼠标左键 |
| 跳转到代码块的开始 | Ctrl + [ |
| 跳转到代码块的结束 | Ctrl + ] |
| 跳转到代码块的开始并选中 | Ctrl + Shift + [ |
| 跳转到代码块的结束并选中 | Ctrl + Shift + ] |
| 重构 | 复制文件 | F5 |
| 移动文件 | F6 |
| 重命名文件 | Shfit + F6 |
| 预览重构列表 | Ctrl + Alt + Shift + T |
| 将代码提取为函数 | Ctrl + Alt + M |
| 将代码提取为变量 | Ctrl + Alt + V |
| 将代码提取为常数 | Ctrl + Alt + C |
| 将代码提取为字段 | Ctrl + Alt + F |
| 将代码提取为参数 | Ctrl + Alt + P |
| 代码折叠 | 展开当前代码块 | Ctrl + + |
| 收缩当前代码块 | Ctrl + - |
| 展开所有代码块 | Ctrl + Shift + + |
| 收缩所有代码块 | Ctrl + Shift + - |
| 转成全大写、全小写 | 转成全大写、全小写 | Ctrl + Shift + U |
| 高亮 | 高亮 | Ctrl + F7 |
| 查看 | 快速插入代码 | Ctrl + J |
| 查看方法签名 | Ctrl + P |
| 查看文档描述 | Ctrl + Q |
| 查看方法的声明与调用 | Alt + F7 |
| 查看源码实现 | Ctrl + Shift + I |
| 查看文件结构 | Ctrl + F12 / Alt + 7 |
|  | 展示意图行为 | Ctrl + Alt |
| 选取操作 | 扩大选区 | Ctrl + W |
| 缩小选取 | Ctrl + Shift + W |
| 选中光标处的字符串（不含引号） | Double Ctrl + W |
| 选中光标处的字符串（含引号） | Triple Ctrl + W |
| 操作代码行 | 复制行 | Ctrl + D |
| 删除行 | Ctrl + Y |
| 向上拉取一行 | Alt + Shift + 向上箭头 |
| 向下拉取一行 | Alt + Shift + 向下箭头 |
| 向上移动整个方法（光标放在方法开头） | Ctrl + Shift + 向上箭头 |
| 向下移动整个方法（光标放在方法开头） | Ctrl + Shift + 向下箭头 |
| git | 更新 | Ctrl + T |
| 提交 | Ctrl + K |
| 推送 | Ctrl + Shift + K |
| 查看历史提交记录 | Alt + 9 |
| 修正最后一次提交 | Alt + M |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |



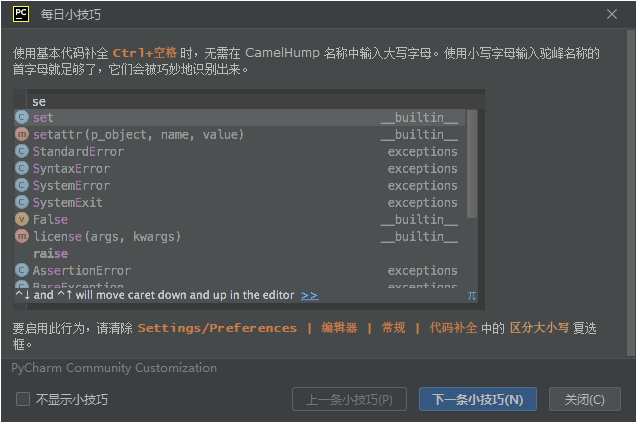
### 技巧

#### 补全

* 代码补全 Ctrl+空格 可以帮助快速补全代码语句。它会在您输入时运行，并提供当前文本光标位置处可用的建议列表:

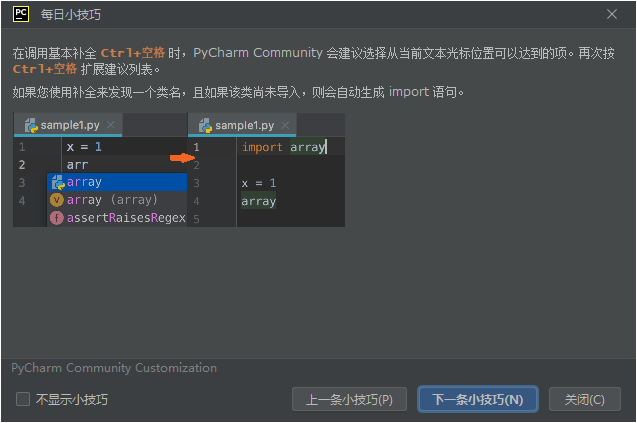


* 使用基本代码补全 Ctrl+空格 时，无需在 CamelHump 名称中输入大写字母。使用小写字母输入驼峰名称的首字母就足够了，它们会被巧妙地识别出来



* 在调用基本补全 Ctrl+空格 时，PyCharm Community 会建议选择从当前文本光标位置可以达到的项。再次按 Ctrl+空格 扩展建议列表。

如果您使用补全来发现一个类名，且如果该类尚未导入，则会自动生成 import 语句。



* 在交互式控制台中工作时，您无需记住命令行语法或可用的函数。相反，您可以使用熟悉的代码补全 Ctrl+空格 。此外，在查找列表中，您可以按 Ctrl+Q 查看条目的文档。

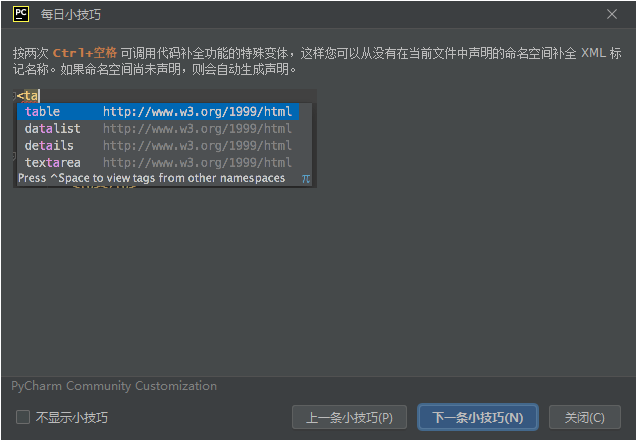


* 使用基本代码补全 Ctrl+空格 时，无需在 CamelHump 名称中输入大写字母。使用小写字母输入驼峰名称的首字母就足够了，它们会被巧妙地识别出来

要启用此行为，请清除 Settings/Preferences | 编辑器 | 常规 | 代码补全 中的 区分大小写 复选框。



* 按两次 Ctrl+空格 可调用代码补全功能的特殊变体，这样您可以从没有在当前文件中声明的命名空间补全 XML 标记名称。如果命名空间尚未声明，则会自动生成声明。



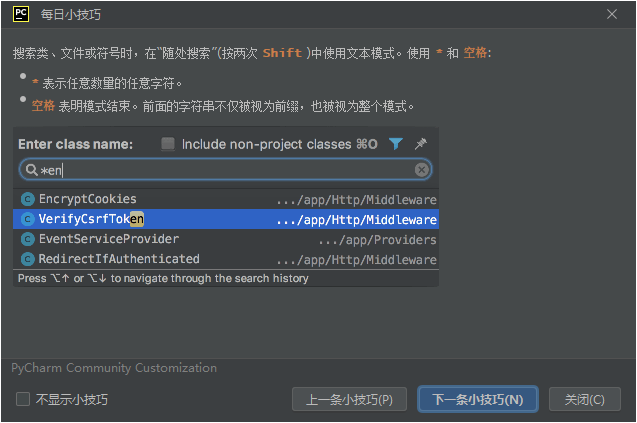
#### 搜索

* 按两次 Shift 可在项目中搜索文件、操作、符号、UI 元素、Git 分支和注释。再次按两次 Shift ，可将搜索扩展到非项目条目。
* 对类使用选项卡或直接快捷键 Ctrl+N ，对文件使用 Ctrl+Shift+N ，对符号使用 Ctrl+Alt+Shift+N ，对操作使用 Ctrl+Shift+A ，可以缩小搜索结果的范围。

#### 搜索类、文件或符号时，在“随处搜索”(按两次 Shift )中使用文本模式。使用 \* 和 空格:

\* 表示任意数量的任意字符。

空格 表明模式结束。前面的字符串不仅被视为前缀，也被视为整个模式。



#### 要快速打开某个文件，请按 Ctrl+Shift+N (导航 | 文件 )并开始输入其名称。使用筛选器图标可以仅在所选文件类型中进行搜索:

#### PyCharm Community 2021.3 最新变化

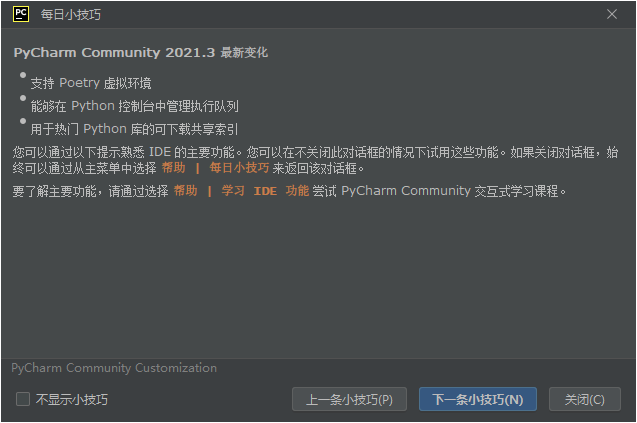
支持 Poetry 虚拟环境

能够在 Python 控制台中管理执行队列

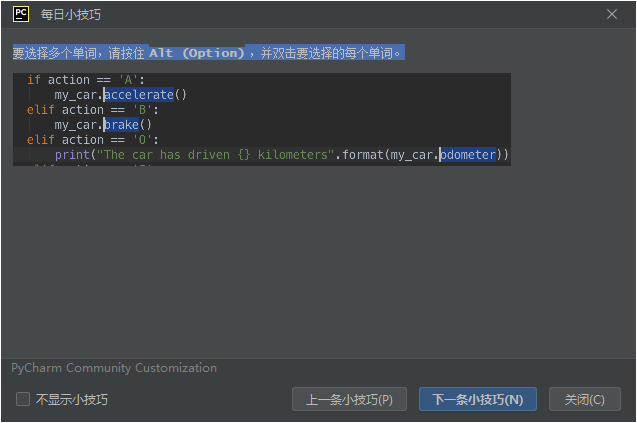
用于热门 Python 库的可下载共享索引

您可以通过以下提示熟悉 IDE 的主要功能。您可以在不关闭此对话框的情况下试用这些功能。如果关闭对话框，始终可以通过从主菜单中选择 帮助 | 每日小技巧 来返回该对话框。

要了解主要功能，请通过选择 帮助 | 学习 IDE 功能 尝试 PyCharm Community 交互式学习课程。

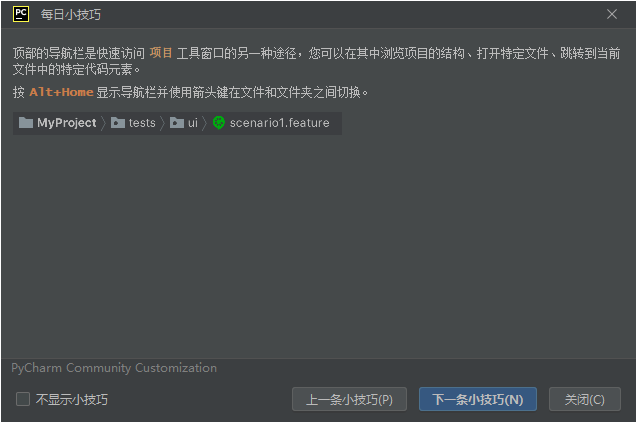


#### 要选择多个单词，请按住 Alt (Option) ，并双击要选择的每个单词。



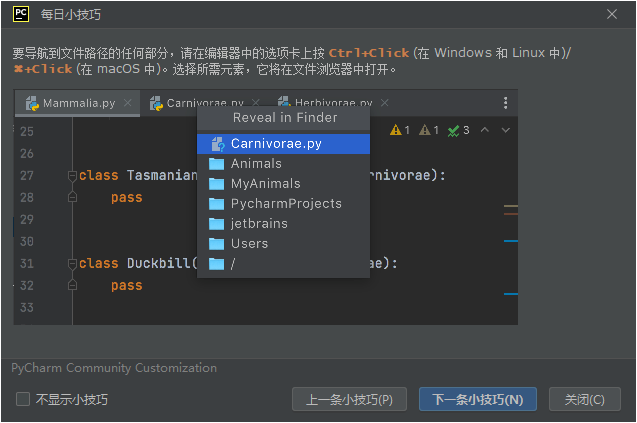
#### 顶部的导航栏是快速访问 项目 工具窗口的另一种途径，您可以在其中浏览项目的结构、打开特定文件、跳转到当前文件中的特定代码元素。

按 Alt+Home 显示导航栏并使用箭头键在文件和文件夹之间切换。



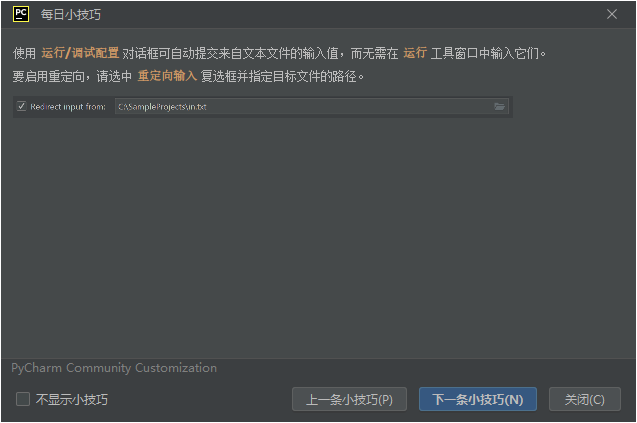
#### 要比较任意两个文件或文件夹，请在 项目 工具窗口中将其选中，然后按 Ctrl+D。

#### 要导航到文件路径的任何部分，请在编辑器中的选项卡上按 Ctrl+Click (在 Windows 和 Linux 中)/ ⌘+Click (在 macOS 中)。选择所需元素，它将在文件浏览器中打开



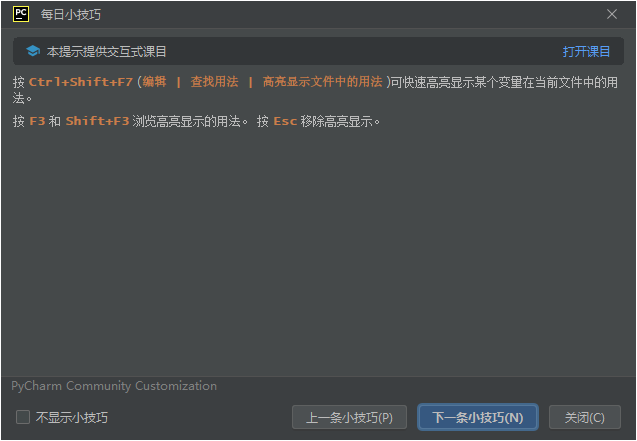
#### 使用 运行/调试配置 对话框可自动提交来自文本文件的输入值，而无需在 运行 工具窗口中输入它们。

要启用重定向，请选中 重定向输入 复选框并指定目标文件的路径。



#### 按 Ctrl+Shift+F7 (编辑 | 查找用法 | 高亮显示文件中的用法 )可快速高亮显示某个变量在当前文件中的用法。

按 F3 和 Shift+F3 浏览高亮显示的用法。 按 Esc 移除高亮显示。



#### 要验证正则表达式是否正确，请将文本光标置于要检查的表达式中，按 Alt+Enter ，然后选择 编辑正则表达式。

在弹出窗口中，键入应与正则表达式相匹配的示例字符串。 图标表明发现匹配。



#### 要在调试程序时评估任何表达式，请在编辑器中选择表达式并按 Alt+F8。

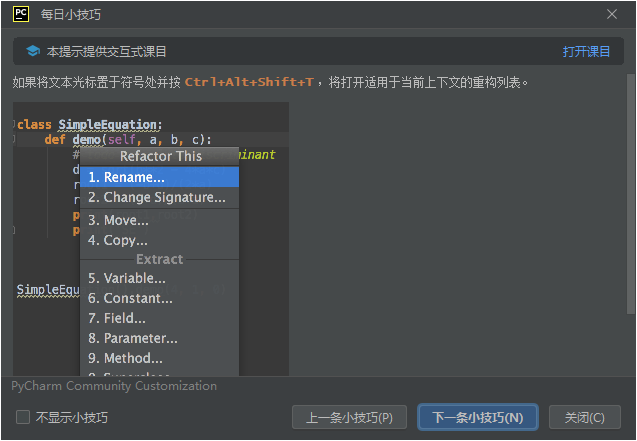
#### 要在编辑器中打开文件的特定行，请按 Ctrl+Shift+N (导航 | 文件 )，开始输入文件名，从建议列表中选择，然后输入 : ，后面跟随行号。

#### PyCharm Community 提供了一项快速修复，它可以自动安装您尝试导入的软件包: 如果在 import 关键字后面输入计算机上当前不可用的软件包的名称，快速修复将建议您忽略未解析的引用，或下载并安装缺失的软件包。

#### 无需显式调用快速文档 Ctrl+Q 。如果将鼠标指针悬停在所需符号上，快速文档弹出窗口将自动显示。

要启用此功能，请在 Settings/Preferences 对话框 (Ctrl+Alt+S) 的 编辑器 | 代码编辑 页面上选中 悬停时显示快速文档 复选框。

#### 如果将文本光标置于符号处并按 Ctrl+Alt+Shift+T ，将打开适用于当前上下文的重构列表。



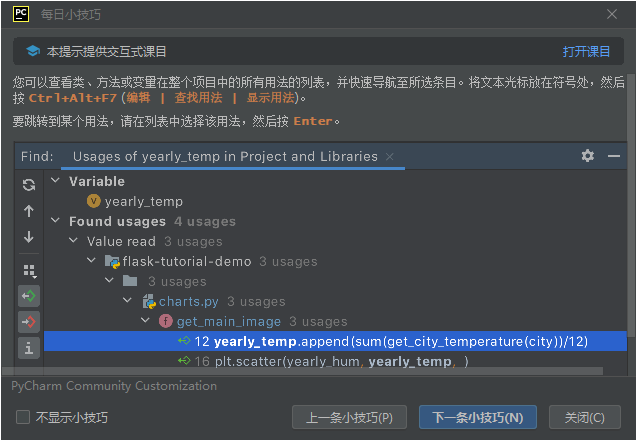
#### 为代码补全启用 输入时显示建议 选项时，可以使用 向上查找 和 向下查找 操作来关闭建议列表并在编辑器中上下移动文本光标。

在 Windows 中，使用 Ctrl+Down 和 Ctrl+Up 快捷键。

在 macOS 中，默认情况下未配置快捷键，因此您可以在 偏好设置 | 键盘映射 中分配任何方便的组合键。

#### 您可以查看类、方法或变量在整个项目中的所有用法的列表，并快速导航至所选条目。将文本光标放在符号处，然后按 Ctrl+Alt+F7 (编辑 | 查找用法 | 显示用法)。

要跳转到某个用法，请在列表中选择该用法，然后按 Enter。



#### 要让打印输出(文件 | 打印 )变得美观且内容丰富，可以使用关键字。

例如，使用 $DATE$ 和 $TIME$ 可以指定打印输出的确切日期和时间。

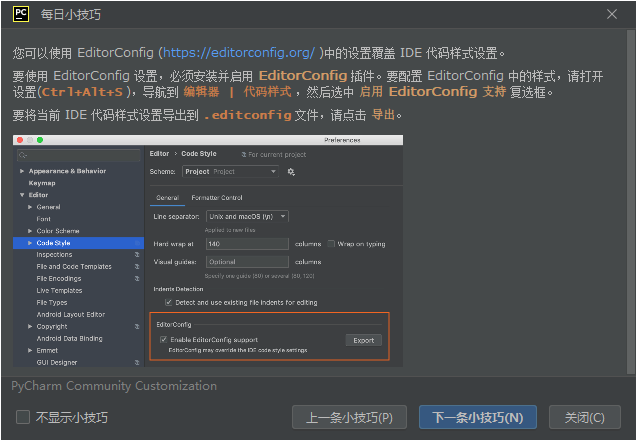
#### 要快速打开某个文件，请按 Ctrl+Shift+N (导航 | 文件 )并开始输入其名称。使用筛选器图标可以仅在所选文件类型中进行搜索:

#### 按 Alt+向上箭头 和 Alt+向下箭头 可在编辑器中的方法之间快速移动。

#### 您可以使用 EditorConfig (https://editorconfig.org/ )中的设置覆盖 IDE 代码样式设置。

要使用 EditorConfig 设置，必须安装并启用 EditorConfig 插件。要配置 EditorConfig 中的样式，请打开设置(Ctrl+Alt+S )，导航到 编辑器 | 代码样式 ，然后选中 启用 EditorConfig 支持 复选框。

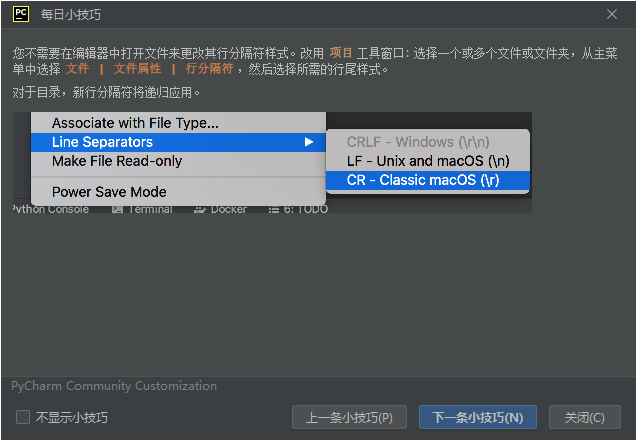
要将当前 IDE 代码样式设置导出到 .editconfig 文件，请点击 导出。



#### 要将两行合并为一行并移除不必要的空格，请按 Ctrl+Shift+J。

#### 您不需要在编辑器中打开文件来更改其行分隔符样式。改用 项目 工具窗口: 选择一个或多个文件或文件夹，从主菜单中选择 文件 | 文件属性 | 行分隔符 ，然后选择所需的行尾样式。

对于目录，新行分隔符将递归应用。

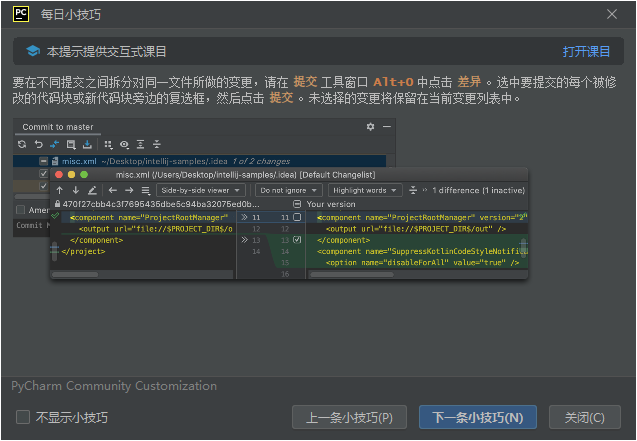


#### 搜索类、文件或符号时，在 随处搜索 弹出窗口(按两次 Shift )中使用驼峰拼写法可以筛选结果列表。

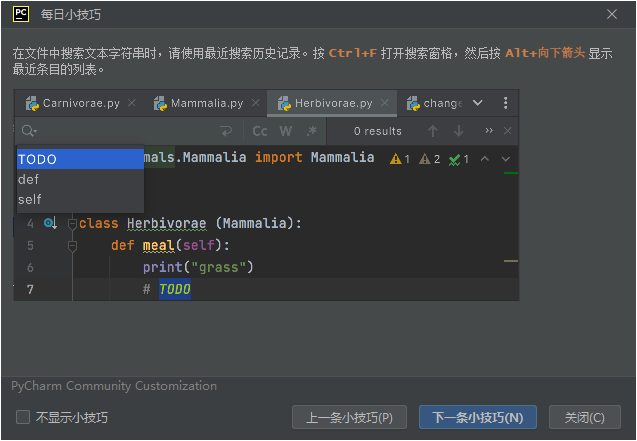
#### 您可以在代码中自定义 import 语句的结构。import 语句可以单独放置、按源代码分组或保持非结构化。

要进行选择，请转到 设置/偏好设置 (按 Ctrl+Alt+S ，选择 编辑器 | 代码样式 | Python ，然后选中 "from" import 的结构 下的相应复选框。

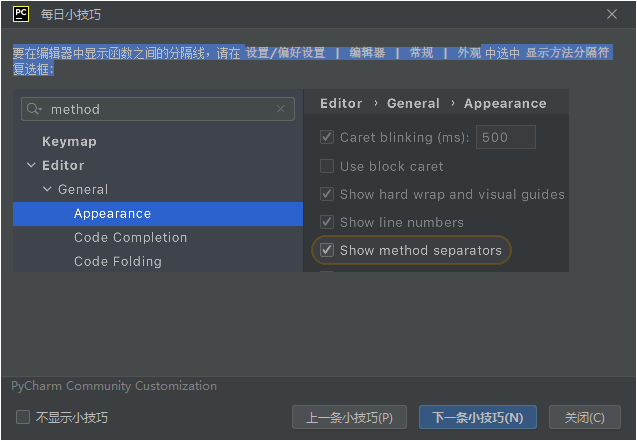
#### 要在不同提交之间拆分对同一文件所做的变更，请在 提交 工具窗口 Alt+0 中点击 差异 。选中要提交的每个被修改的代码块或新代码块旁边的复选框，然后点击 提交 。未选择的变更将保留在当前变更列表中



#### 在文件中搜索文本字符串时，请使用最近搜索历史记录。按 Ctrl+F 打开搜索窗格，然后按 Alt+向下箭头 显示最近条目的列表。



#### 要在编辑器中显示函数之间的分隔线，请在 设置/偏好设置 | 编辑器 | 常规 | 外观 中选中 显示方法分隔符 复选框:



#### PyCharm Community 提供了 检查 微件，该微件位于编辑器的右上角，可以显示当前文件中检测到的所有问题(从错误到拼写错误)数量。

点击该微件可在专门的工具窗口中仔细查看每个问题，并应用最合适的修正。点击箭头，即可从文件中的一个问题跳转到另一个问题。



#### PyCharm Community 包括一个内置的 IDE 功能培训工具。它可以提供交互式培训课程，帮助您学习如何执行一些常见任务，例如重构代码、浏览项目、调试和使用版本控制。

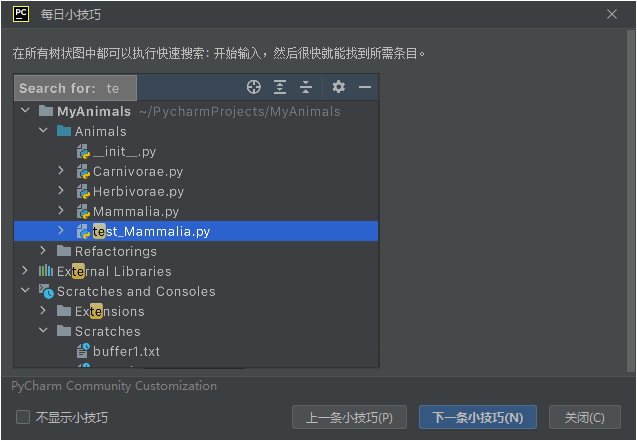
确保在 IDE 设置中 插件 页面上的 已安装 选项卡下启用 IDE Features Trainer 插件。

要打开 学习 工具窗口并开始练习，请从主菜单中选择 视图 | 工具窗口 | 学习 或 帮助 | 学习 IDE 功能 。工具窗口会列出可用的教程。

#### 使用代码补全时，可以按 Tab 来接受弹出列表中当前高亮显示的选择。

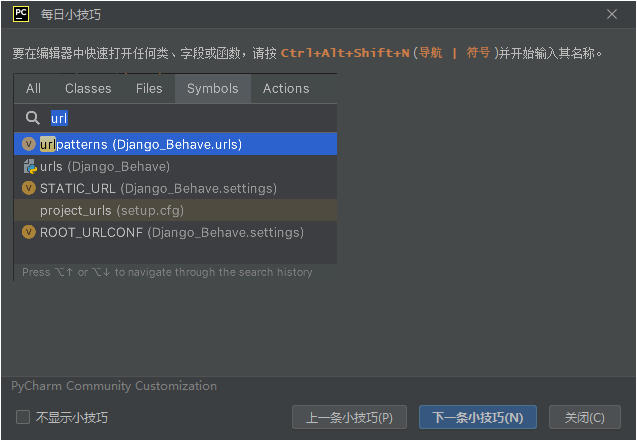
与使用 Enter 接受建议不同，所选名称将覆盖文本光标右侧名称的其余部分。这对于将一个方法或变量名称替换为另一个特别有用。

#### 在所有树状图中都可以执行快速搜索: 开始输入，然后很快就能找到所需条目



#### 按 Alt+Shift+F10 可在不使用鼠标的情况下访问主工具栏上的 运行/调试 配置列表。

#### 要在编辑器中快速打开任何类、字段或函数，请按 Ctrl+Alt+Shift+N (导航 | 符号 )并开始输入其名称。



#### 要比较任意两个文件或文件夹，请在 项目 工具窗口中将其选中，然后按 Ctrl+D。

#### 在任何工具窗口中按 Esc 都会将焦点移至编辑器。

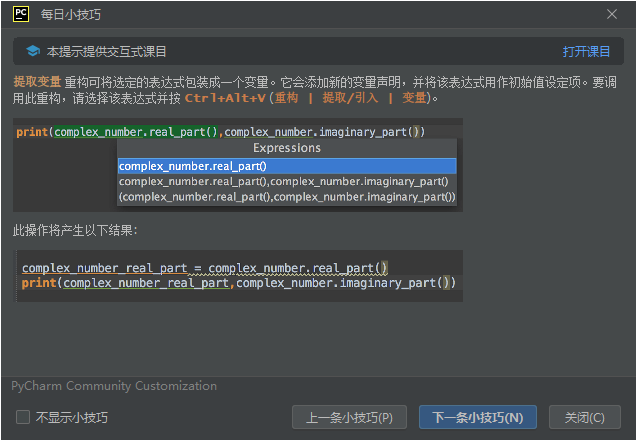
按 Shift+Esc 可将焦点移至编辑器，并隐藏当前或最后一个活动工具窗口。

按 F12 可将焦点从编辑器移至上次聚焦的工具窗口。

#### 使用 Alt+Shift+C 可以快速查看最近对项目进行的变更。

#### 按 Ctrl+向上箭头/ Ctrl+向下箭头 (在 Window/Linux 中)和 上/ 下 数组按钮(在 macOS 中)可滚动查看命令历史记录。

#### 提取变量 重构可将选定的表达式包装成一个变量。它会添加新的变量声明，并将该表达式用作初始值设定项。要调用此重构，请选择该表达式并按 Ctrl+Alt+V (重构 | 提取/引入 | 变量)。

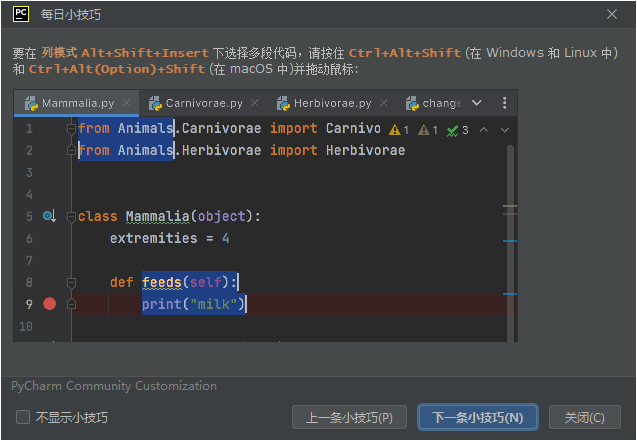


#### 拼写和语法检查默认处于启用状态。这些检查由与 IDE 捆绑在一起的 Grazie 插件提供。要配置校对设置，请按 Ctrl+Alt+S 打开设置，然后导航至 编辑器 | 自然语言 | 语法。

所有文本，包括注释、文本字符串和文字，都根据预定义或自定义字典进行检查。拼写错误会高亮显示，以便您可以选择正确的单词、接受当前的拼写或禁用检查。

要修正拼写或语法错误，请按 Alt+Enter 查看可用操作列表。

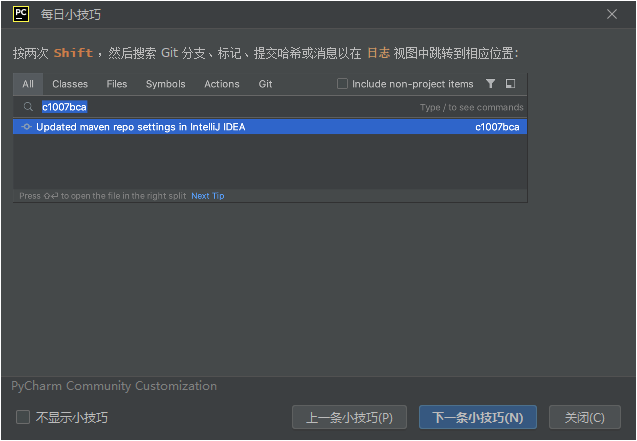
#### 要在 列模式 Alt+Shift+Insert 下选择多段代码，请按住 Ctrl+Alt+Shift (在 Windows 和 Linux 中)和 Ctrl+Alt(Option)+Shift (在 macOS 中)并拖动鼠标:



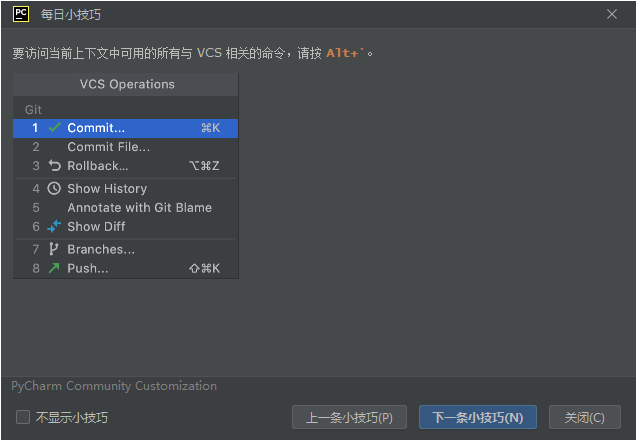
#### 使用代码补全时，可以按 Ctrl+. 选择第一个条目。

PyCharm Community 将根据当前上下文插入所选条目，后面跟随点号或 ->。

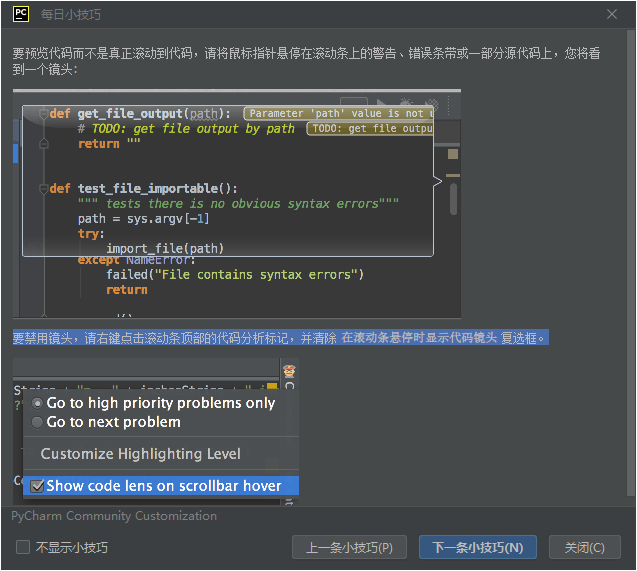
#### 按两次 Shift ，然后搜索 Git 分支、标记、提交哈希或消息以在 日志 视图中跳转到相应位置:



#### 要访问当前上下文中可用的所有与 VCS 相关的命令，请按 Alt+`



#### 要预览代码而不是真正滚动到代码，请将鼠标指针悬停在滚动条上的警告、错误条带或一部分源代码上，您将看到一个镜头: 要禁用镜头，请右键点击滚动条顶部的代码分析标记，并清除 在滚动条悬停时显示代码镜头 复选框。

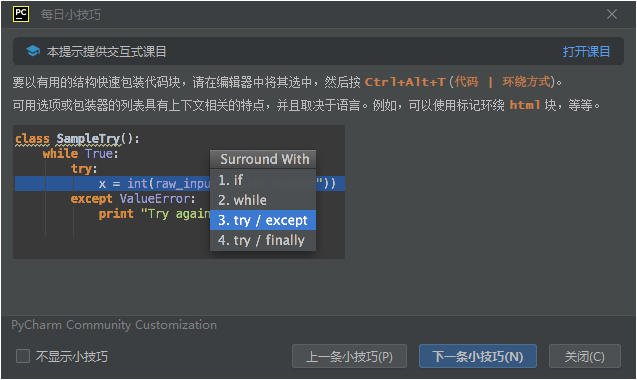


#### 在 PyCharm Community 中，可以使用多个文本光标。按住 Shift+Alt+Ctrl （Windows 和 Linux）或 Shift+Alt(Option)+Cmd (macOS)，然后在编辑器中点击不同位置，以设置其他文本光标。



#### 要以有用的结构快速包装代码块，请在编辑器中将其选中，然后按 Ctrl+Alt+T (代码 | 环绕方式)。

可用选项或包装器的列表具有上下文相关的特点，并且取决于语言。例如，可以使用标记环绕 html 块，等等



#### 要快速查找菜单命令、设置或操作，请按 Ctrl+Shift+A (帮助 | 查找操作)。

#### 通过意图操作(Alt+Enter )访问最常用的断点操作和筛选器。

#### 要展开选区，请按 Ctrl+W 。每次按 Ctrl+W 时，选区将扩展到代码的其他区域。

例如，选区从一个方法名称扩展到调用此方法的表达式，然后扩展到整个语句，继而扩展到包含的块，等等

#### 按 Ctrl+E (视图 | 最近的文件 )可查看最近打开的文件列表。

* 要在打开的文件和工具窗口之间切换，请使用切换器 Ctrl+Tab 。要进行导航，请按住 Ctrl 并使用 向上 和 向下 箭头键或 Tab、 Shift+Tab 和 Alt。

按 Delete 或 BackSpace 关闭编辑器选项卡或隐藏工具窗口。

# 刷题

## 必须先校验输入参数的list是否合法

## 判断列表是否为空：if seq:

if seq的原理返回的是bool(seq)，会调用seq.\_\_bool\_\_，如果seq对应的类没有\_\_bool\_\_，则调用\_\_len\_\_，如果也没有\_\_len\_\_且非None，会默认返回True。

## 同时给多个变量赋值：left, right = 0, size – 1

## for循环：for num in nums:，而不使用for i in range(len(nums)):

## 当遍历涉及范围和索引时，尽量使用enumerate替代range

## 一维列表的初始化：ans = [0] \* length

## 二维列表的初始化

dp = [[1 for i in range(n)]] \* m ##错误写法

dp = [[0 for i in range(n)] for j in range(m)]

dp = [[0] \* n for \_ in range(m)]

## 尽量不要使用全局变量

    def \_\_init\_\_(self):

**self.ans = 0**

    def dfs(self, root):

        if not root:

            return 0

        left = self.dfs(root.left)

        right = self.dfs(root.right)

        node\_sum = left + right + root.val

        self.ans += (abs(left - right))

        return node\_sum

    def findTilt(self, root: TreeNode) -> int:

        if not root:

            return 0

        ans = []

        self.dfs(root)

    return self.ans

## 判断变量：if exist:，不是if exist == True: or if exist is True:

## int的最大值sys.maxsize和float('inf')

## 整除是//

## collections.defaultdict(int)#建立目标字符字典

## == 运算符比较两个对象的值（对象中保存的数据），而 is 比较对象的标识。

## isinstance()与type()

## 数据结构

### 堆栈

stack = [3, 4, 5]

stack.append(6)

stack.pop()

### 队列

**from collections import deque**

queue = deque(["Eric", "John", "Michael"])

queue.append("Terry")

queue.popleft()

### 优先级队列

class PriorityQueue:

    '''实现了优先级队列'''

    def \_\_init\_\_(self):

        self.queue = []

        self.index = 0

    def push(self, item, priority):

        heapq.heappush(self.queue, (-priority, self.index, item))

        self.index += 1

        return

    def pop(self):

        item = heapq.heappop(self.queue)[-1]

        self.index -= 1

        return item

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

    queue = PriorityQueue()

    queue.push("A", 5)

    queue.push("B", 6)

    queue.push("C", 2)

    queue.push("D", 7)

    queue.push("E", 6)

    print(queue.index)

    print(queue.pop())

    print(queue.index)

    print(queue.pop())

    print(queue.index)

    print(queue.pop())

print(queue.index)

## 迭代与排序

### 迭代

#### 在序列中循环时，用 enumerate() 函数可以同时取出位置索引和对应的值：

**>>> for** i, v **in** enumerate(['tic', 'tac', 'toe']):

**...**  print(i, v)

#### 同时循环两个或多个序列时，用 zip() 函数可以将其内的元素一一匹配：

**>>>** questions = ['name', 'quest', 'favorite color']

**>>>** answers = ['lancelot', 'the holy grail', 'blue']

**>>> for** q, a **in** zip(questions, answers):

**...**  print('What is your **{0}**? It is **{1}**.'.format(q, a))

#### 在字典中循环时，用 items() 方法可同时取出键和对应的值：

**>>>** knights = {'gallahad': 'the pure', 'robin': 'the brave'}

**>>> for** k, v **in** knights.items():

**...**  print(k, v)

#### 逆向循环序列时，先正向定位序列，然后调用 reversed() 函数：

**>>> for** i **in** reversed(range(1, 10, 2)):

**...**  print(i)

#### 按指定顺序循环序列，可以用 sorted() 函数，在不改动原序列的基础上，返回一个重新的序列：

**>>>** basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

**>>> for** i **in** sorted(basket):

**...**  print(i)

#### 使用 set() 去除序列中的重复元素。使用 sorted() 加 set() 则按排序后的顺序，循环遍历序列中的唯一元素：

**>>>** basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

**>>> for** f **in** sorted(set(basket)):

**...**  print(f)

### 排序

#### 排序方法list.sort()与排序函数sorted()

Python 列表有一个内置的 list.sort() 方法可以直接修改列表，该方法仅被定义在list中。还有一个 sorted() 内置函数，它会从一个可迭代对象构建一个新的**排序列表**，该函数适用于所有的可迭代序列。

##### key参数——比较函数

函数原型

list.sort(cmp=None, key=None, reverse=False)

sorted(iterable, cmp=None, key=None, reverse=False)

list.sort() 和 sorted() 都有一个 key 形参用来指定在进行比较前要在每个列表元素上调用的函数（或其他可调用对象）。key 形参的值应该是个函数（或其他可调用对象），它接受一个参数并返回一个用于排序的键。 这种机制速度很快，因为对于每个输入记录只会调用一次键函数。

1. 键函数模式

**>>>** sorted("This is a test string from Andrew".split(), key=str.lower)

['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']

**>>>** student\_tuples = [

**...**  ('john', 'A', 15),

**...**  ('jane', 'B', 12),

**...**  ('dave', 'B', 10),

**...** ]

**>>>** sorted(student\_tuples, key=**lambda** student: student[2]) *# sort by age*

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

**>>> class** **Student**:

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

**...**  self.name = name

**...**  self.grade = grade

**...**  self.age = age

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

**...**  **return** repr((self.name, self.grade, self.age))

**>>>** student\_objects = [

**...**  Student('john', 'A', 15),

**...**  Student('jane', 'B', 12),

**...**  Student('dave', 'B', 10),

**...** ]

**>>>** sorted(student\_objects, key=**lambda** student: student.age) *# sort by age*

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

1. operator模块函数

operator模块有 itemgetter() 、 attrgetter() 和 methodcaller() 函数。

**>>> from** **operator** **import** itemgetter, attrgetter

**>>>** sorted(student\_tuples, key=itemgetter(2))

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

**>>>** sorted(student\_objects, key=attrgetter('age'))

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

**>>>** sorted(student\_tuples, key=itemgetter(1,2))

[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

**>>>** sorted(student\_objects, key=attrgetter('grade', 'age'))

[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

##### reverse 参数——升降序

**>>>** sorted(student\_tuples, key=itemgetter(2), reverse=**True**)

[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

**>>>** sorted(student\_objects, key=attrgetter('age'), reverse=**True**)

[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

1. 多重排序

**>>> def** multisort(xs, specs):

**...**  **for** key, reverse **in** reversed(specs):

**...**  xs.sort(key=attrgetter(key), reverse=reverse)

**...**  **return** xs

**>>>** multisort(list(student\_objects), (('grade', **True**), ('age', **False**)))

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

<https://docs.python.org/zh-cn/3/howto/sorting.html#sortinghowto>

#### 对字典分别按键（key）和值（value）进行排序

##### 对字典按键（key）进行排序

import operator

test\_data\_0=sorted(dict\_data.keys())

test\_data\_1=sorted(dict\_data.items(), key=lambda x:x[0])

test\_data\_2=sorted(dict\_data.items(), key=operator.itemgetter(0))

##### 对字典按值（value）进行排序

import operator

test\_data\_0=sorted(dict\_data.items(), key=lambda x:x[1])

test\_data\_1=sorted(dict\_data.items(), key=operator.itemgetter(1))

## lambda 表达式重构

如果使用 lambda 表达式导致一段代码难以理解，Fredrik Lundh 建议像下面这样重构。

(1) 编写注释，说明 lambda 表达式的作用。

(2) 研究一会儿注释，并找出一个名称来概括注释。

(3) 把 lambda 表达式转换成 def 语句，使用那个名称来定义函数。

(4) 删除注释。

## 经常使用的标准库

### 高级数据类型

from array import array

from collections import defaultdict, deque, namedtuple

import heapq

from sys import maxsize

### typing模块

from typing import List, Tuple, Dict, FrozenSet, Set

from typing import Iterable, Iterator, Generator

from typing import Callable

from typing import Literal