命令式编程 VS 函数式编程

首先从大家熟悉的命令式编程开始，我们先回顾下平时在写代码时主要的情景。

其实，不管我们的业务代码有多复杂，都离不开以下几类操作：

- 函数定义：def
- 条件控制：if, elif, else
- 循环控制：for, break, continue, while

当然，这只是部分操作类型，除此之外还应该有类和模块、异常处理等等。但考虑到是入门，我们就先只关注上面这三种最常见的操作。

对应地，函数式编程也有自己的关键字。在Python语言中，用于函数式编程的主要由3个基本函数和1个算子。

- 基本函数：map()、reduce()、filter()
- 算子(operator)：lambda

令人惊讶的是，仅仅采用这几个函数和算子就基本上可以实现任意Python程序。

当然，能实现是一回事儿，实际编码时是否这么写又是另外一回事儿。估计要真只采用这几个基本单元来写所有代码的话，不管是在表达上还是在阅读上应该都挺别扭的。不过，尝试采用这几个基本单元来替代上述的函数定义、条件控制、循环控制等操作，对理解函数式编程如何通过函数和递归表达流程控制应该会很有帮助。

在开始尝试将命令式编程转换为函数式编程之前，我们还是需要先熟悉下这几个基本单元。

Python 内建函数大全

> 翻译总结自官方文档：原文地址

Python 解释器内置了许多函数和类型，列表如下（按字母排序）（省略了几个不常用的）。

		内建函数表		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	`__import__()`
complex()	hasattr()	max()	round()	

常见内建函数

print

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

将 objects 打印到文本流 file 中，以 sep 分隔，然后以 end 结尾。必须将 sep，end，file 和 flush（如果存在）作为关键字参数给出。

所有非关键字参数都会转换为像 `str()` 那样的字符串并写入流中，由 sep 隔开，然后结束。sep 和 end 都必须是字符串；它们也可以是 None，这意味着使用默认值。如果没有给出对象，`print()` 将只写入 end。

文件参数必须是带有 `write(string)` 方法的对象；如果它不存在或是 None，则将使用 `sys.stdout`。由于打印的参数会转换为文本字符串，`print()` 不能用于二进制模式文件对象。对于这些，请改用 `file.write(...)`。

输出是否缓冲通常由 file 决定，但如果 flush 关键字参数为 true，则强制刷新流。

input

如果 prompt 参数存在，则将其写入标准输出而没有尾随换行符。然后该函数从输入中读取一行，将其转换为一个字符串（剥离尾随的换行符），然后返回该行。读取 EOF 时，引发 EOFError。例：

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"

```

open

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
```

打开 file 并返回相应的文件对象。如果文件无法打开，则会引发 `OSError`。

file 是一个类似路径的对象，它提供要打开的文件的路径名（绝对或相对于当前工作目录）或要包装的文件的整数文件描述符。（如果给出文件描述符，则在返回的 I/O 对象关闭时关闭，除非 closefd 设置为 `False`。）

mode 是一个可选字符串，用于指定打开文件的模式。它默认为 `'r'`，表示使用文本的方式打开文件来读取。其他常见的值是 `'w'` 用于写入（如果文件已经存在，则覆盖该文件），`'x'` 用于独占创建，`'a'` 用于附加（在某些 Unix 系统上，这意味着无论当前的搜索位置如何，所有写操作都会附加到文件末尾）。在文本模式下，如果未指定编码，则使用的编码与平台相关：调用 `locale.getpreferredencoding(False)` 以获取当前语言环境编码。（为了读取和写入原始字节，使用二进制模式并且不用指定编码）可用的模式有：

字符	含义
`'r'`	用于读取（默认）
`'w'`	用于写入，首先覆盖文件
`'x'`	用于独占创建，如果文件已经存在则失败
`'a'`	用于写入，追加到文件末尾（如果存在）
`'b'`	二进制模式
`'t'`	文本模式（默认）
`'+'`	打开磁盘文件进行更新（读取和写入）
`'U'`	通用换行符模式（已弃用）

默认模式是 `'r'`（用于读取文本，`'rt'` 的同义词）。对于二进制读写访问，模式 `'w+b'` 打开并将文件删减为 0 字节。 `'r+b'` 打开文件而不删减。

如概述中所述，Python 区分二进制和文本 I/O。以二进制模式打开的文件（mode参数中包括 `'b'`）将内容作为字节对象返回，而不进行任何解码。在文本模式下（默认情况下，或当 `'t'` 包含在 mode 参数中时），文件内容以 str 形式返回，字节首先使用平台相关编码进行解码，或者使用指定的编码（如果给出）。

!> Python 不依赖于底层操作系统的文本文件概念；所有的处理都由 Python 自己完成，因此是平台无关的。

range

```
range(stop)
range([start], stop[, step])
```

range 不是一个函数，它实际上是一个不可变的序列类型

```
In [8]: list(range(10))
Out[8]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [9]: list(range(0, 10, 2))
Out[9]: [0, 2, 4, 6, 8]

```

常见高级函数

lambda

它们在其他语言中也被称为匿名函数。如果你不想在程序中对一个函数使用两次，你也许会想用lambda表达式，它们和普通的函数完全一样。

`lambda argument: manipulate(argument)`

`lambda 参数:操作(参数)`

```
In [1]: add = lambda x, y: x + y

In [2]: add(3, 5)
Out[2]: 8
```

```
a = [(1, 2), (4, 1), (9, 10), (13, -3)]

def f(x):
    return x[1]

# a.sort(key=f)
a.sort(key=lambda x: x[1])

print(a)
# Output: [(13, -3), (4, 1), (1, 2), (9, 10)]
```

sorted

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

从 iterable 中的 item 中返回一个新的排序列表。

有两个可选参数，必须将其指定为关键字参数。

key 指定一个带有一个参数的函数，用于从每个列表元素中提取比较键：`key=str.lower`。默认值是`None`（直接比较元素）。

reverse 是一个布尔值。如果设置为 `True`，那么列表元素按照每个比较被颠倒的顺序进行排序。

内置的 `sorted()` 函数排序是稳定的。如果确保不会更改比较相等的元素的相对顺序，则排序是稳定的。

map

map(function, iterable, ...)

返回一个将 function 应用于每个 iterable item 的迭代器，从而产生结果。如果传递额外的 iterable 参数，function 必须采用多个参数并应用于并行所有迭代中的项目。使用多个迭代器时，当最短迭代器耗尽时，迭代器停止。 In [54]: list1 = [1, 2, 3,

```
In [3]: list1 = [4, 3, 7, 1, 9]

In [4]: list2 = list(map(str, list1))

In [5]: list2
Out[5]: ['4', '3', '7', '1', '9']

In [6]: list(map(lambda x, y: x * y, list1, list2))
Out[6]: ['4444', '333', '7777777', '1', '999999999']
```

enumerate

enumerate(*iterable*, *start=0*)

返回一个枚举对象。 iterable 必须是一个序列，一个迭代器或其他支持迭代的对象。由 `enumerate()` 返回的迭代器的 `__next__()` 方法返回一个元组，该元组包含一个计数（从 start 开始，默认值为 0）以及遍历迭代获得的值。

```
In [1]: enum = ['Spring', 'Summer', 'Fall', 'Winter']

In [2]: for e in enumerate(enum):
   ...:     print(e)
   ...:
(0, 'Spring')
(1, 'Summer')
(2, 'Fall')
(3, 'Winter')
```

zip

zip(*iterables)

制作一个迭代器，用于聚合来自每个迭代器的元素。

返回元组的迭代器，其中第 i 个元组包含来自每个参数序列或迭代的第 i 个元素。当最短的输入迭代耗尽时，迭代器停止。使用单个迭代参数，它将返回 1 元组的迭代器。没有参数，它返回一个空的迭代器。

与 `*` 操作符一起使用 `zip()` 可用于解压缩列表：

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

```
data = zip(list1, list2)
data = sorted(data)
list1, list2 = map(lambda t: list(t), zip(*data))

```

filter

filter(*function, iterable*)

用那些 function 返回 true 的 iterable 元素构造一个迭代器。iterable 可以是序列，支持迭代的容器或迭代器。如果 function 为 `None`，则假定标识函数为 false，即为 false 的所有元素都被删除。

```
# 过滤0-10之间的偶数
In [8]: list(filter(lambda x: x%2==0, range(10)))
Out[8]: [0, 2, 4, 6, 8]

```

三元表达式

三元运算符通常在Python里被称为条件表达式，这些表达式基于真(true)/假(false)的条件判断.

它允许用简单的一行快速判断，而不是使用复杂的多行`if`语句。 这在大多数时候非常有用，而且可以使代码简单可维护。

```
# 如果条件为真，返回真 否则返回假
condition_is_true if condition else condition_is_false
```

```
if condition:
    result = condition_is_true
else:
    result = condition_is_false
```

反射

主要是指程序可以访问、检测和修改它本身状态或行为的一种能力。

id(object)

返回一个对象的 “identity”。它是一个整数，它在其生命周期中保证对这个对象唯一且恒定。具有非重叠生命周期的两个对象可能具有相同的 id() 值。

CPython 实现细节：这是内存中对象的地址。

type

```
class type(object)
class type(name, bases, dict)
```

有一个参数时，返回 object 的类型。返回值是一个类型对象，通常与`object.__class__`返回的对象相同。

建议使用`isinstance()`内置函数来测试对象的类型，因为它会考虑子类。

有三个参数时，返回一个新的类型对象。这实质上是类声明的一种动态形式。name 字符串是类名，并成为`__name__`属性；bases 元组逐项列出基类，并成为`__bases__`属性；dict 是包含类体的定义的命名空间，并被复制到标准字典中以变为`__dict__`属性。例如，以下两条语句会创建相同的类型对象：

```
>>> class X:
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

len(s)

返回对象的长度（条目数量）。参数可以是一个序列（如 string，bytes，tuple，list 或 range）或集合（如字典，set 或 frozenset）。

也可用于实现了`__len__()`方法的任意对象

```
In [40]: class A:
    ...:     def __len__(self):
    ...:         return 10
    ...:
In [41]: a = A()

In [42]: len(a)
Out[42]: 10

```

dir

dir([*object*])

尝试返回 object 的有效属性列表。如果没有参数，则返回当前本地作用域中的名称列表。

如果对象具有名为 `__dir__()` 的方法，则将调用此方法，并且必须返回属性列表。这允许实现自定义 `__getattr__()` 或 `__getattribute__()` 函数的对象自定义 `dir()` 报告其属性。

默认的 `dir()` 机制对不同类型的对象有不同的表现，因为它试图产生最相关的信息，而不是完整的信息：

- 如果对象是模块对象，则列表包含模块属性的名称。
- 如果对象是一个类型或类对象，则该列表包含其属性的名称，并递归地显示其基础的属性。
- 否则，该列表包含对象的属性名称，其类属性的名称以及其类的基类的属性的递归。

结果列表按字母顺序排序。例如：

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']

```

isinstance

isinstance(*object*, *classinfo*)

如果 object 参数是 classinfo 参数的实例或其（直接，间接或虚拟）子类的实例，则返回 true。如果 object 不是给定类型的对象，则该函数总是返回 false。如果 classinfo 是类型对象的元组， object 是其中任何一个类型的实例，则返回 true。如果 classinfo 不是类型或一组类型的元组，则会引发 `TypeError` 异常。

```
In [30]: isinstance(10, int)
Out[30]: True

In [31]: isinstance("str", (int, str))
Out[31]: True

In [32]: isinstance(max, int)
Out[32]: False
```

issubclass

issubclass(*class*, *classinfo*)

如果 class 是 classinfo 的子类（直接，间接或虚拟），则返回 true。一个类被认为是它自己的一个子类。 classinfo 可以是类对象的元组，在这种情况下，将检查 classinfo 中的每个条目。在任何其他情况下，都会引发 `TypeError` 异常。

```
In [34]: issubclass(int, int)
Out[34]: True

In [35]: issubclass(10, int)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-37910f193c07> in <module>()
----> 1 issubclass(10, int)

TypeError: issubclass() arg 1 must be a class

In [36]: issubclass(int, str)
Out[36]: False

```

setattr

setattr(*object, name, value*)

它和 `getattr()` 是一对。参数是一个对象，一个字符串和一个任意值。该字符串可以是现有的属性名或新的属性名。如果该对象允许，该函数将 value 分配给该属性。例如，`setattr(x, 'foobar', 123)` 等同于 `x.foobar = 123`。

hasattr

hasattr(*object*, *name*)

参数是一个对象和一个字符串。如果字符串是 object 属性之一的名称，则结果为 `True`，否则为 `False`。（这是通过调用 `getattr(object, name)` 并查看它是否引发 `AttributeError` 实现的。）

getattr

getattr(*object*, *name*[, *default*])

返回 object 的指定属性的值。name 必须是字符串。如果字符串是 object 属性之一的名称，则结果是该属性的值。例如，`getattr(x, 'foobar')` 等同于 `x.foobar`。如果指定的属性不存在，则返回默认值（如果提供），否则引发 `AttributeError`。

delattr

delattr(object, name)

参数是一个对象和一个字符串。该字符串必须是对象属性之一的名称。该函数删除指定的属性（只要该对象允许）。例如， `delattr(x, 'foobar')` 等价于 `del x.foobar`。

```
class Example:
    pass


e = Example()

setattr(e, 'a', 1)
print(e.a)
print(hasattr(e, 'a'))
print(getattr(e, 'a'))
delattr(e, 'a')
print('e.a', e.a)
```

globals()

返回表示当前全局符号表的字典。它总是当前模块的字典（在函数或方法内部，它是定义它的模块，而不是从中调用它的模块）。

locals()

更新并返回表示当前本地符号表的字典。在函数块中调用时，locals() 返回自由变量，但不能在类块中调用。

!> 不应该修改其中的内容；更改可能不会影响解释器使用的本地变量和自由变量的值。

vars([*object*])

返回一个模块、字典、类、实例或者其它任何一个具有 `__dict__` 属性的对象的 `__dict__` 属性。

模块和实例这样的对象的 `__dict__` 属性可以更新；但是其它对象可能对它们的 `__dict__` 属性的写操作有限制（例如，类使用 `types.MappingProxyType` 来阻止对字典直接更新）。

如果不带参数，`vars()` 的行为就像 `locals()`。注意，locals 字典只用于读取，因为对 locals 字典的更新会被忽略。

```
class Person:
  name = "John"
  age = 36
  country = "norway"

x = vars(Person)
```

案例演示(看看即可):

假设 names 是二十个长度的变量名列表, 获取所有变量名的字符串用于构造字典.

{'变量':变量,}

```
a, b, c, d, e, f, g = map(lambda x: chr(x+97), range(7))

names = [a, b, c, d, e, f, g]
# 利用 locals 获取所有的变量名
vk = {}
# 获取局部变量副本 保证不修改局部变量
p_locals = locals().copy()
# 将内容变为键值对, 并获取id与对象的隐射关系
for k, v in p_locals.items():
    vk[id(v)] = k
print("vk", vk)
data_dict = {}

# 获取变量名的字符串
for i in names:
    data_dict.update({
        vk[id(i)]: i
    })
# 打印字符串
print(data_dict)
```

super

super([*type*[, *object-or-type*]])

返回一个代理对象，它委托方法给父类或者 type 的同级类。这对于访问类中被覆盖的继承方法很有用。搜索顺序与 `getattr()` 使用的顺序相同，只不过 type 本身被跳过。

type 的 `__mro__` 属性列出 `getattr()` 和 `super()` 使用的方法解析顺序。该属性是动态的，并且可以在继承层次结构更新时更改。

如果省略第二个参数，则返回的 super 对象是未绑定的。如果第二个参数是一个对象，则 `isinstance(obj, type)` 必须为 true。如果第二个参数是类型，则 `issubclass(type2, type)` 必须为 true（这对类方法很有用）。

super 有两种典型的使用情况。在具有单继承的类层次结构中，可以使用 super 来引用父类，而不必明确命名它们，从而使代码更易于维护。这种使用非常类似于在其他编程语言中 super 的使用。

第二种使用情况是在动态执行环境中支持协同多继承。这种使用情况是 Python 独有的，在静态编译语言或仅支持单继承的语言中找不到。这使得可以实现 “菱形图”，其中多个基类实现相同的方法。良好的设计指出此方法在每种情况下具有相同的调用顺序（因为调用的顺序在运行时确定，因为该顺序适应类层次结构中的更改，并且因为该顺序可以包括在运行时之前未知的兄弟类）。

对于这两种用例，典型的超类调用如下所示：

```
class C(B):
    def method(self, arg):
        super().method(arg)    # This does the same thing as:
                               # super(C, self).method(arg)
```

!> 注意，`super()` 只实现显式点分属性查找的绑定过程，例如 `super().__getitem__(name)`。它通过实现自己的 `__getattribute__()` 方法来实现这一点，以便以支持协同多继承需要的以可预测的顺序搜索类。因此，`super()` 没有定义隐式的查找语句或操作，例如 `super()[name]`。

!> 另请注意，除了零参数形式外，`super()` 不限于在方法内部使用。如果两个参数的形式指定了准确的参数，就能进行正确的引用。零参数形式只能在类定义中使用，因为编译器会填充必要的细节以正确检索正在定义的类，以及访问普通方法的当前实例。

数据类型

int

class int(x=0)
class int(x, base=10)

返回一个由数字或字符串 x 构造的整数对象，如果没有给出参数，则返回 0。如果 x 不是数字，则返回 `x.__int__()`。

```
In [22]: class A:
    ...:     def __int__(self):
    ...:         return 10
    ...:
In [23]: a = A()

In [24]: int(a)
Out[24]: 10
```

如果 x 不是数字或给定了 base，那么 x 必须是一个 string， bytes 或 bytearray 实例，它表示以 base 为基数的整数文字。或者，文字可以在前面加上 `+`或 `-` （两者之间没有空格）。

```
In [25]: int('-10')
Out[25]: -10

In [26]: int('+10')
Out[26]: 10

In [27]: int('- 10')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-27-a62cc7794a18> in <module>()
----> 1 int('- 10')

ValueError: invalid literal for int() with base 10: '- 10'

In [28]: int('1000',2)
Out[28]: 8

In [29]: int('ff',16)
Out[29]: 255

```

float

返回一个由数字或字符串 x 构造的浮点数。

在删除前后空白字符后，输入必须符合以下语法：

```
sign           ::=  "+" | "-"
infinity       ::=  "Infinity" | "inf"
nan            ::=  "nan"
numeric_value  ::=  floatnumber | infinity | nan
numeric_string ::=  [sign] numeric_value

```

对于一般的 Python 对象 x，`float(x)` 委托给 `x .__float__()`。

如果没有给出参数，则返回 0.0。

例子：

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf

```

bool

返回一个布尔值，即 `True` 或 `False` 中的一个。 x 使用标准真值测试方式进行转换。如果 x 为 false 或省略，则返回 `False`；否则返回 `True`。 bool 类是 `int` 的子类。它不能进一步子类化。它唯一的实例是 `False` 和 `True`。

str

```
class str(object='')
class str(object=b'', encoding='utf-8', errors='strict')
```

返回一个字符串对象

list

list([*iterable*])

list 不是一个函数，它实际上是一个可变的序列类型。

tuple

tuple 不是一个函数，它实际上是一个不可变的序列类型

set

set([*iterable*])

返回一个新的集合对象，可选地使用来自 iterable 的元素。 set 是一个内置的类。

dict

class dict(`**kwarg`)
class dict(`mapping`, `**kwarg`)
class dict(`iterable`, `**kwarg`)

创建一个新的字典

```
In [38]: dict(name='jack',age=18)
Out[38]: {'name': 'jack', 'age': 18}

In [39]: dict({'name': 'jack'}, age=18)
Out[39]: {'name': 'jack', 'age': 18}

In [40]: dict([('name', 'jack'),('age', 18)])
Out[40]: {'name': 'jack', 'age': 18}

```

object

返回一个新的无特征的对象。object 是所有类的基类。它具有所有 Python 类实例通用的方法。这个函数不接受任何参数。

!> object 没有 `__dict__`，所以不能为 object 类的实例指定任意属性。

bytes

bytes([*source*[, *encoding*[, *errors*]]])

返回一个新的 “bytes” 对象，它是一个在 `0 <= x <256` 范围内的不可变整数序列。`bytes` 是 `bytearray` 的不可变版本 - 它具有相同的非变异方法和相同的索引和切片行为。

因此，构造函数参数解释请参考 `bytearray()`。

字节对象也可以使用文字创建。请参阅字符串和字节文字。

bytearray

bytearray([*source*[, *encoding*[, *errors*]]])

返回一个新的字节数组。 bytearray 类是一个在 `0 <= x < 256` 范围内的可变整数序列。

可选的 `source` 参数可以用几种不同的方式初始化数组：

- 如果它是一个字符串，则还必须给出 encoding（以及可选的 errors）参数; 然后 `bytearray()` 使用 `str.encode()` 将字符串转换为字节。
- 如果它是一个整数，则将其作为数组的长度，并将用空字节进行初始化。
- 如果它是符合缓冲区接口的对象，则将使用该对象的只读缓冲区来初始化字节数组。
- 如果它是一个 iterable，必须是 `0 <= x <256` 范围内的可迭代对象，它们将被用作数组的初始内容。

没有参数，就会创建一个大小为 0 的数组。

```
In [11]: bytearray(5)
Out[11]: bytearray(b'\x00\x00\x00\x00\x00')

In [12]: bytearray([23, 32, 4, 67, 9, 96, 123])
Out[12]: bytearray(b'\x17 \x04C\t`{')

In [13]: bytearray()
Out[13]: bytearray(b'')

```

complex

complex([*real*[, *imag*]])

返回值为 real + imag*1j 的复数或者将字符串或数字转换为复数。如果第一个参数是一个字符串，它将被解释为一个复数，并且该函数必须在没有第二个参数的情况下被调用。第二个参数不能是一个字符串。每个参数可以是任何数字类型（包括复数）。如果省略了 imag，它将默认为零，并且构造函数用作像 int 和 float 这样的数字转换。如果两个参数均被省略，则返回 0j。

!> 从字符串转换时，该字符串不得在 `+` 或 `-` 运算符周围包含空格。例如，`complex('1+2j')` 很好，但 `complex('1 + 2j')` 会引发 `ValueError`。

类装饰器

@staticmethod

将方法转换为静态方法。

静态方法不会收到隐式的第一个参数。要声明一个静态方法，习惯用法如下：

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...

```

它可以在类（如 `C.f()`）或实例（如 `C().f()`）上调用。

Python 中的静态方法类似于 Java 或 C++ 中的。

@classmethod

将方法转换为类方法。

类方法将类作为第一个参数接收（隐式的），就像实例方法接收实例一样。为了声明一个类方法，习惯用法如下：

```
class C:
    @classmethod
    def f(mcs, arg1, arg2, ...): ...

```

!> 注意：类方法和静态方法不是一个概念

@property

```
class property(fget=None, fset=None, fdel=None, doc=None)
```

返回一个 property 属性。

fget 是获取属性值的函数。fset 是用于设置属性值的函数。fdel 是删除属性值时会调用的函数。doc 为该属性创建一个文档字符串。

典型的用法是定义一个托管属性 x：

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")

```

如果 c 是 C 的一个实例，`c.x` 将调用 getx，`c.x = value` 将调用 setx ，`del c.x` 将调用 delx。

如果给定，doc 将是 property 属性的文档字符串。否则，该属性将复制 fget 的文档字符串（如果存在）。这使得使用 `property()` 作为装饰器可以轻松创建只读属性：

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage

```

`@property` 修饰器将 `voltage()` 方法转换为具有相同名称的只读属性的 “getter”，并将 voltage 的文档字符串设置为 `“Get the current voltage.”`。

property 对象具有可用作装饰器的 getter，setter 和 deleter 方法，这些方法创建属性的副本并将相应的存取器函数设置为装饰函数。这可以用一个例子来解释：

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""

```

```
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x

```

此代码与第一个示例完全等效。请务必为附加函数提供与原始 property 相同的名称（当前为 x）。

返回的 property 对象也具有与构造函数参数相对应的属性 fget，fset 和 fdel。

聚类操作

max

```
max(iterable, *[, key, default])
max(arg1, arg2, *args[, key])
```

返回 iterable 中的最大项或两个或更多个参数中最大的项。

如果提供了一个位置参数，它应该是一个 iterable。iterable 中最大的 item 被返回。如果提供了两个或多个位置参数，则返回最大的位置参数。

有两个可选的关键字参数。 key 参数指定一个像 `list.sort()` 那样的单参数排序函数。如果提供的迭代器为空，则 default 参数指定要返回的对象。如果迭代器为空且未提供缺省值，则会引发 `ValueError`。

如果最大值包含多个 item，则该函数返回遇到的第一个 item。这与 `sorted(iterable, key=keyfunc, reverse=True)[0]` 和 `heapq.nlargest(1, iterable, key=keyfunc)` 等其他排序工具稳定性保持一致。

```
In [60]: list1 = [4, 3, 7, 1, 9]

In [61]: max(list1, key=lambda x: -x)
Out[61]: 1

In [62]: max([])
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-62-a48d8f8c12de> in <module>()
----> 1 max([])

ValueError: max() arg is an empty sequence

In [63]: max([], default=1)
Out[63]: 1

```

min

```
min(iterable, *[, key, default])
min(arg1, arg2, *args[, key])
```

返回 iterable 中的最小项或两个或更多个参数中的最小项。

如果提供了一个位置参数，它应该是一个 iterable。iterable 中的最小项被返回。如果提供两个或多个位置参数，则返回最小的位置参数。

有两个可选的关键字参数。 key 参数指定一个像 `list.sort()` 那样的单参数排序函数。如果提供的迭代器为空，则 default 参数指定要返回的对象。如果迭代器为空且未提供缺省值，则会引发 `ValueError`。

如果最小值包含多个 item，则该函数返回遇到的第一个 item。这与 `sorted(iterable, key=keyfunc, reverse=True)[0]` 和 `heapq.nlargest(1, iterable, key=keyfunc)` 等其他排序工具稳定性保持一致。

sum

sum(*iterable*[, *start*])

从 start 开始，从左到右对 iterable 中的元素求和。 start 默认是 0，迭代的 item 通常是数字，并且不允许 start 的值为字符串。

abs(*x*)

返回一个数字的绝对值。参数可以是整数或浮点数。如果参数是一个复数，则返回它的模。

pow

pow(*x*, *y*[, *z*])

返回 x 的 y 次方；返回 x 的 y 次方再除以 z 的余数（计算效率比 `pow(x, y) % z` 更高）。双参数形式 `pow(x, y)` 等价于使用幂运算符：`x**y`。

round

round(*number*[, *ndigits*])

返回在小数点后舍入到精度 ndigits 的 number 。如果 ndigits 被省略或者是 `None`，它将返回最接近的整数表示。

对于支持 `round()` 的内建类型，值舍入到 10 的最接近的负 ndigits 次幂的倍数；如果离两个倍数的距离相等，则舍入选择偶数（因此，`round(0.5)` 和 `round(-0.5)` 都是 0，而 `round(1.5)` 是 2 ）。ndigits 可以是任何整数值（正数，零或负数）。如果使用一个参数调用则返回值是一个 integer，否则与 number 的类型相同。

```
In [10]: type(round(10.9))
Out[10]: int

In [11]: type(round(10.9, 2))
Out[11]: float
```

对于一般的 Python 对象 xxx，`round(xxx, ndigits)` 委托给 `xxx.__round__(ndigits)`。

!> `round()` 对于浮点数的行为可能会令人惊讶：例如，`round(2.675, 2)` 给出 2.67，而不是预期的 2.68。这不是一个 bug：这是由于大多数小数不能完全表示为浮点数的结果。

all

all(*iterable*)

如果 `iterable` 的所有元素均为 True（或 `iterable` 为空），则返回 `True`。相当于：

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any

any(*iterable*)

如果 `iterable` 中有任何一个元素为 true，则返回 `True`。如果 `iterable` 为空，则返回 `False`。相当于：

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

divmod

divmod(*a*, *b*)

以两个（非复数）数字作为参数，并在使用整数除法时返回由它们的商和余数组成的一对数字。使用混合操作数类型时，适用二元算术运算符的规则。对于整数，结果与 `(a // b, a % b)` 相同。对于浮点数，结果是 `(q, a % b)`，其中 q 通常是 `math.floor(a / b)`，但可能小于 1。在任何情况下， `q * b + a % b` 都非常接近 a，如果 `a % b` 不为零，则它具有与 b 相同的符号，并且 `0 <= abs(a % b) < abs(b)`。

```
In [53]: divmod(10, 3)
Out[53]: (3, 1)

In [54]: divmod(10.1, 3)
Out[54]: (3.0, 1.0999999999999996)
```

进制转化

ascii

ascii(*object*)

类似 `repr()`，返回一个包含对象的可打印表示的字符串，但使用 `\x`， `\u` 或 `\U` 转义符转义由 `repr()` 返回的字符串中的非 ASCII 字符。这会生成一个类似于 Python 2 中 `repr()` 返回的字符串。

```
In [1]: s = 'python \n 中文'

In [2]: ascii(s)
Out[2]: "'python \\n \\u4e2d\\u6587'"

In [3]: repr(s)
Out[3]: "'python \\n 中文'"
```

bin(*x*)

将整数转换为以 “0b” 为前缀的二进制字符串。结果是一个有效的 Python 表达式。如果 `x` 不是Python `int` 对象，则必须定义返回整数的 `__index __()` 方法。一些例子：

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

可以使用以下任意方式，控制是否需要前缀 “0b”：

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

有关更多信息，另请参阅 `format()`。

当 `x` 不是 `int` 类型时

```
In [1]: class Test:
   ...:     def __init__(self, n):
   ...:         self.n = n
   ...:
   ...:     def __index__(self):
   ...:         return self.n
   ...:

In [2]: t = Test(10)

In [3]: bin(t)
Out[3]: '0b1010'
```

chr(*i*)

返回表示 Unicode 代码点为整数 i 的字符的字符串。例如，`chr(97)` 返回字符串 `'a'`，而 `chr(8364)` 返回字符串 `'€'`。这是 `ord()` 的逆过程。

参数的有效范围是从 0 到 1,114,111（基于 16 的 0x10FFFF）。如果超出这个范围，将会抛出 ValueError。

ord(*c*)

给定一个代表一个Unicode字符的字符串，返回一个表示该字符的 Unicode code 点的整数。例如，`ord('a')` 返回整数 97，`ord('€')`（欧元符号）返回 8364。这是 `chr()` 的逆过程

oct(*x*)

将整数转换为以 “0o” 为前缀的八进制字符串。结果是一个有效的 Python 表达式。如果 x 不是 Python int 对象，则必须定义返回整数的 `__index__()` 方法。例如：

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'

```

如果要将整数转换为八进制字符串，控制是否显示前缀 “0o”，则可以使用以下任一方式。

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')

```

hex(*x*)

将整数转换为以 “0x” 为前缀的小写十六进制字符串。如果 x 不是 Python int 对象，则必须定义返回整数的 `__index __()` 方法。一些例子：

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'

```

如果要将整数转换为带有前缀或不带前缀的大写或小写十六进制字符串，可以使用以下任一方式：

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')

```

!> 要获取浮点数的十六进制字符串表示形式，请使用 `float.hex()` 方法。

特殊函数

iter

iter(*object*[, *sentinel*])

返回一个迭代器对象。根据第二个参数是否存在，第一个参数的解释有所不同。如果没有第二个参数，object 必须是支持迭代协议（`__iter__()` 方法）的集合对象，或者它必须支持序列协议（整数参数从 0 开始的 `__getitem__()` 方法）。如果它不支持这两种协议，则会引发 `TypeError`。如果给出了第二个参数 sentinel，那么 object 必须是可调用的对象。在这种情况下创建的迭代器将调用没有参数的 object，以便对其 `__next__()` 方法进行调用；如果返回的值等于 sentinel，则会触发 `StopIteration`，否则将返回该值。

第二种形式的 `iter()` 的一个例子是按行读取文件，直到到达某一行。以下示例读取文件，直到 `readline()` 方法返回空字符串：

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

next

next(*iterator*[, *default*])

通过调用 `__next__()` 方法从 iterator 中检索下一个 item。如果给出了 default，则在迭代器耗尽时返回它，否则引发 `StopIteration`。

repr(*object*)

返回一个包含对象可打印表示的字符串。对于许多类型，此函数尝试返回一个字符串，该字符串在传递给 `eval()` 时会产生一个具有相同值的对象，否则该表示是一个用尖括号括起来的字符串，其中包含对象类型的名称以及其他信息包括对象的名称和地址。一个类可以通过定义 `__repr__()` 方法来控制此函数为其实例返回的内容。

callable(*object*)

如果 object 参数可调用，则返回 `True`，否则返回 `False`。如果返回 true，调用失败仍然是可能的，但如果是 false，调用 object 将永远不会成功。请注意，类是可调用的（调用一个类返回一个新的实例）；如果类有一个 `__call__()` 方法，则实例可以被调用。

3.2版本中的新功能：此功能在 Python 3.0 中首先被删除，然后在 Python 3.2 中恢复。

```
In [19]: a = 1

In [20]: callable(a)
Out[20]: False

In [21]: def func():
    ...:     pass
    ...:

In [22]: callable(func)
Out[22]: True

```

```
In [23]: class A:
    ...:     pass
    ...:

In [24]: a = A()

In [25]: callable(a)
Out[25]: False

In [26]: class A:
    ...:     def __call__(self, *args, **kwargs):
    ...:         pass
    ...:

In [27]: a = A()

In [28]: callable(a)
Out[28]: True

```

slice

```
class slice(stop)
class slice(start, stop[, step])
```

返回表示由 `range(start, stop, step)` 指定的一组索引的切片对象。start 和 step 参数默认为 `None`。切片对象具有只读数据属性 start、stop 和 step，它们只返回参数值（或它们的默认值）。他们没有其他明确的功能；然而，它们被 Numerical Python 和其他第三方扩展使用。当使用扩展索引语法时，也会生成切片对象。例如：`a[start:stop:step]` 或 `a[start:stop, i]`。

```
In [5]: a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [6]: s = slice(1, 8, 2)

In [7]: a[s]
Out[7]: [1, 3, 5, 7]
```

hash(*object*)

返回对象的散列值（如果有）。哈希值是整数。它们用于在字典查找期间快速比较字典键。比较相等的数值具有相同的散列值（即使它们具有不同的类型，就像 1 和 1.0 一样）。

!> 对于具有自定义 `__hash__()` 方法的对象，请注意，`hash()` 会根据主机的位宽截断返回值。

```
In [1]: class A:
   ...:     pass

In [2]: a = A()

In [3]: hash(a)
Out[3]: 1552656422630569496
```

```
In [4]: class A:
   ...:     def __hash__(self):
   ...:         return 11111111111
   ...:
   ...:

In [5]: a = A()

In [6]: hash(a)
Out[6]: 11111111111

```

format

format(*value*[, *format_spec*])

将值转换为 “格式化” 表示，由 format_spec 控制。 format_spec 的解释将取决于 value 参数的类型，不过，大多数内置类型都使用标准格式化语法：格式化规范迷你语言。

默认 format_spec 是一个空字符串，通常与调用 str(value) 的效果相同。

对 `format(value, format_spec)` 的调用被转换为 `type(value).__format__(value, format_spec)`，它在搜索 value 的 `__format__()` 方法时绕过实例字典。如果方法搜索到达 object 并且 format_spec 非空，或者 format_spec 或返回值不是字符串，则会引发 TypeError 异常。

在 version 3.4 中：如果 format_spec 不是空字符串，则 `object().__format__(format_spec)` 会引发 `TypeError`。

help([*object*])

调用内置的帮助系统。 （此功能用于交互式使用。）如果未提供参数，则交互式帮助系统将在解释器控制台上启动。如果参数是一个字符串，那么该字符串将被查找为模块，函数，类，方法，关键字或文档主题的名称，并在控制台上打印帮助页面。如果参数是任何其他类型的对象，则会生成对象上的帮助页面。

reversed(*seq*)

返回一个反向迭代器。seq 必须是具有 `__reversed__()` 方法或支持序列协议（ `__len__()` 方法和整数参数从 0 开始的 `__getitem__()` 方法）的对象。