Python Library

Pionpill [1]

本文为 Python 相关的内置方法，模块做的简单笔记

2022 年 1 月 21 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

笔者为软件工程系在校本科生，主要利用 Python 进行数据科学与机器学习使用，也有一定游戏开发经验。

本文主要记录 Python 常用的一些内置函数，魔法函数，标准库，装饰器等用法。

在阅读本篇笔记时，本人默认读者已经看过我的 «Fluent Python» 学习笔记或对 Python 有一定程度的了解。本文默认读者都具备中阶 Python 语法。这篇文章写完也就可以进入高阶了捏。

本人的书写环境:

- Python: 3.8.5

2022 年 1 月 21 日

目录

第一部分

内置函数

I 魔法函数

1 数学相关

1.1 一元运算符

1.1.1 __neg__(-)

__neg__ 用于获取负数，没什么好说的，注意他不是减法运算。

1.1.2 __pos__(+)

__pos__ 用于获取正数，没什么好说的，注意他不是加法运算。

1.1.3 __abs__

__abs__ 用于获取绝对值，对应的内置函数为 abs.

1.2 二元运算符

这里所有的二元运算符都是比较运算符。

1.2.1 __lt__(<)

__lt__ 用于实现小于比较。

1.2.2 __le__(<=)

__le__ 用于实现小于等于比较。

1.2.3 __eq__(==)

__eq__ 用于实现等于比较。

1.2.4 __ne__(!=)

__ne__ 用于实现不等比较。

1.2.5 __gt__(>)

__gt__ 用于实现大于比较。

1.2.6 __ge__(>=)

__ge__ 用于实现大于等于比较。

1.3 算数运算符

1.3.1 __add__(+)

__add__ 用于实现加法运算。

1.3.2 __sub__(-)

__sub__ 用于实现减法运算。

1.3.3 __mul__(*)

__mul__ 用于实现乘法运算。

1.3.4 __truediv__(/)

__truediv__ 用于实现除法运算。注意这里是数学意义上的除法运算:

```
    >>> 3/2
1.5
```

1.3.5 __floordiv__(//)

__floordiv__ 用于实现整除运算。

1.3.6 __mod__(%)

__mod__ 用于实现取模运算。

1.3.7 __divmod__

__divmod__ 用于除法运算，对应内置函数 divmod(x,y):

```
>>> help(divmod)
divmod(x, y, /)
    Return the tuple (x//y, x%y). Invariant: div*y + mod == x.
```

1.3.8 __pow__(**)

__pow__ 用于实现幂运算，对应内置函数为 pow(x,y)，对应运算符为 **。

1.3.9 __round__

__round__ 用于实现内置函数 round(number, ndigits)。round() 函数在基本数值类型运算中起到四舍五入的作用。

1.4 位运算符

1.4.1 __invert__(~)

__invert__ 用于实现取反运算。

1.4.2 __lshift__(«)

__lshift__ 用于实现左移运算。

1.4.3 __rshift__(»)

__rshift__ 用于实现右移运算。

1.4.4 __and__(&)

___ 用于实现与运算。

1.4.5 __or__(|)

__or__ 用于实现或运算。

1.4.6 __xor__(^)

__xor__ 用于实现非运算。

1.5 反向与增量运算符

1.5.1 反向运算符的规则

反向算数运算符即: 将算术运算的两个主要参数调换位置进行运算。如果是相同数据类型，那么反向算数运算可以直接委托给正向运算函数。

```
def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue = 0.0)
    return Vector(a + b for a,b in pairs)

def __radd__(self, other):
    return self + other   # 直接委托给 __add__
```

此外，Python 有一个规定：如果由于类型不兼容而导致运算符特殊方法无法返回有效的结果，那么应该返回 NotImplemented，而不是抛出 TypeError。返回 NotImplemented 时，另一个操作数所属的类型还有机会执行运算，即 Python 会尝试调用反向方法。

这也就要求我们在正向运算符中尝试捕获 TypeError 错误并返回 NotImplemented 错误:

```
def __add__(self, other):
    try:
        pairs = itertools.zip_longest(self, other, fillvalue = 0.0)
        return Vector(a + b for a,b in pairs)
    except TypeError:
        return NotImplemented
```

1.5.2 运算表

表 1.1 中缀运算符方法名

运算符	正向方法	反向方法	就地方法	说明
+	__add__	__radd__	__iadd__	加法或拼接
-	__sub__	__rsub__	__isub__	减法
*	__mul__	__rmul__	__imul__	乘法或重复复制
/	__truediv__	__rtruediv__	__itruediv__	除法
//	__floordiv__	__rfloordiv__	__ifloordiv__	整除
%	__mod__	__rmod__	__imod__	取余
divmod()	__divmod__	__rdivmod__	__idivmod__	商和模组成的元组
**, pow()	__pow__	__rpow__	__ipow__	幂运算
@	__matmul__	__rmatmul__	__imatmul__	矩阵乘法
&	__and__	__rand__	__iand__	位与
|	__or__	__ror__	__ior__	位或
^	__xor__	__rxor__	__ixor__	位异或
<<	__lshift__	__rlshift__	__ilshift__	按位左移
>>	__rshift__	__rrshift__	__irshift__	按位右移

2 类相关

2.1 字符串表示

2.1.1 __repr__

__repr__ 是由 object 对象提供的，所有类都会继承这个方法。该方法用于提供一个“自我描述”。当直接打印类的实例化对象时，系统会输出对象的自我描述信息。

如果没有重写该方法,我们使用 print() 打印对象时会返回一个“xxx object at 0x...”的形式。

此外，__repr__ 方法返回的信息应该是面向程序员的。在交互环境下返回的是该方法。

2.1.2 __str__

__str__ 方法的作用和 __repr__ 类似，它是面向用户的，应该更为人性化。

我们在使用 print() 打印对象实例时，如果定义了 __str__ 方法，就会采用对应的输出，如果没有，则使用 __repr__ 对应的输出，如果两个方法都没有实现，则采用最原始的 “xxx object at 0x...” 形式。

此外，__str__ 有一个对应的内置函数 str()。

2.1.3 __unicode__

__unicode__ 对应的内置函数为 unicode()。unicode() 与 str() 都是 basestring 的子类。不同的是，unicode() 返回值是 unicode，str() 返回值是 str。

如果我们没有定义 __unicode__,unicode() 函数会返回 unicode(self.__str__())。反之不成立。

2.2 构造相关

2.2.1 __init__

__init__ 是类的构造函数，如果需要创建类的实例，就会调用这个方法。__init__ 的第一个参数是 self 指代实例本身，涉及到实例本身成员的方法都需要包含这个参数。

如果某个类没有定义 __init__ ，且它的继承树上的类也没有定义该魔法函数，则他不可以被实例化。

2.2.2 __new__

__new__ 实现的是让类去创建实例。

`__new__` 方法和 `__init__` 方法作用类似，都是用于构建实例的构造函数。如果两个魔法函数都存在，则调用 `__new__`，由 `__new__` 决定是否要调用 `__init__`。

可以这样理解: 类的 `__init__` 负责将类实例化，而在调用它之前，`__new__` 决定是否要使用 `__init__` 方法，因为，`__new__` 除了调用类中的 `__init__` 方法，还可以调用其他类的构造方法或者工厂函数等。

构建 `__new__` 方法的注意点:

- 第一个参数是 `cls` 代表类，而不是代表实例的 `self`。
- `__new__` 方法始终是类的静态方法，无论是否被加上装饰器。

2.2.3 __call__

Python 中，一切皆为对象，函数也是对象，同时也是可调用对象，实例也可以成为可调用对象。可调用对象可以通过 `callable()` 函数判断。

如果某个类实现了 `__call__` 方法，那么类的实例可以像函数一样执行。

2.2.4 __class__

`__class__` 的作用是返回实例所属的类，一般不需要实现，仅在 DEBUG 中使用。

2.3 属性相关

2.3.1 __getattr__

魔法函数 `__getattr__` 用于获取属性，它有一个对应的内置方法 `getattr()`。下面两种语法是等效的:

```
a.name
getattr(a,"name")
```

当编译器查找某个对象的属性时,会先才用. 运算符查询 `__dict__` 表,如果查不到该属性，则调用 `getattr(a,"attr")` 方法查找属性。若仍然查不到对应的属性，则会报 AttributeError 异常。

除非需要特别处理，一般情况下不需要实现该方法。

2.3.2 __setattr__

`__setattr__` 用于给属性赋值，下面两种写法等效:

```
x.y = v
setattr(x,'y',v)
```

我们在使用第一种方式为实例的属性赋值时，实际上就是调用了 `__setattr__` 方法。

除非需要特别处理，一般情况下不需要实现该方法。

2.3.3 __getattribute__

__getattr__ 是作为取属性的最后方案存在的，而 __getattribute__ 则是用于取属性时拦截，当属性被访问时，将自动调用该方法。因此常用于实现一些访问某属性时执行一段代码的特性。

注意，在访问属性时，最先调用 __getattribute__ 执行对应的代码，其次再有可能获取 __dict__ 表，最后可能调用 __getattr__ 方法。

2.3.4 __dir__

__dir__ 对应的内置函数为 dir(),它会返回实例的所有属性和方法。调用对象的 __dir__ 方法和使用内置函数 dir() 返回的列表是相同的，顺序有可能不同。

有别于 __dict__，仅会获取一部分实例成员，dir() 会获取所有 (包括父级) 的成员。详细区别请参考 __dict__ 小节。

2.3.5 __delattr__

魔法函数 __delattr__ 对应的内置函数为 delattr()，用于删除对象的某个属性。该魔法函数一般无被重写。

```
1 delattr(object,attribute)
```

2.3.6 __del__

魔法函数 __del__ 用于销毁对象，其对应的内置函数为 del()。在开发者编程过程中，很少会直接销毁对象，因为 Python 能很好地自动完成垃圾回收工作。但无论是手动还是自动销毁对象，都会调用对象的 __del__ 方法。

重写 __del__ 对象的主要目的是像 C++ 的析构函数，做一些终端提示等工作。

2.3.7 __all__

魔法函数 __all__ 的主要作用是在 from <module> import * 语句中暴露接口。

不像 Java，C++ 等 OOP 高级语句，Python 语言没有原生的可见性控制，而是靠一套” 约定” 下工作。比如下划线开头的属性/方法应该对应外部不可见。__all__ 提供了暴露接口的白名单，一些不以下划线开头的变量 (比如从其他模块导入的变量) 也将被排除出去。

书写 __all__ 的好处有如下:

- **控制 import * 的形为**

书写了 `__all__` 之后，`import *` 只会导入 `__all__` 中列出的成员。如果成员不存在，则会抛出异常。

- **代码检查**

  有些严格的代码检查工具，如果某个属性/方法被导入却没有应用则会报错，而我们在构建包的时候需要导出对应方法/属性，则可以在 `__all__` 中加入对应方法/属性，就不会再报错了。

- **提供接口**

  书写了 `__all__` 之后，能让模块的使用者清晰地区分哪些是具体的实现方法，哪些是可以调用的接口。

定义 `__all__` 需要注意的地方

- `__all__` 是 `list` 类型的。
- 不应该动态生成 `__all__`。
- 按照 PEP8 建议，`__all__` 应该写在所有 `import` 语句下面，成员上面。

2.3.8 `__dict__`

魔法函数 `__dict__` 用于存储类或实例的成员信息。它并不能直接在类中书写，但程序运行时任何属性/方法的调用都涉及到 `__dict__`，作为开发者 `__dict__` 的主要作用是 DEBUG，和它相关的内置函数为 `dir()`。

`__dict__` 属性:

该函数是用来存储对象成员的一个字典，键为属性名，值为属性值。在类和实例中的 `__dict__` 有所不同:

- **类中的 `__dict__`**

  类中的 `__dict__` 主要存储共享的变量和函数 (包括静态成员等)，类的 `__dict__` 并不包含父类的属性。

  在类定义中的 `cls` 是指类本身，通过它调用的成员都是类的 `__dict__` 中的成员。

- **实例中的 `__dict__`**

  实例中的 `__dict__` 仅存储与实例相关的实例属性。每个实例的属性各不影响。

  在类定义中的 `self` 是指实例。

`dir()` 函数

`dir()` 函数是 Python 提供的一个 API 函数，该函数会自动寻找一个对象的所有属性 (包括父类中继承的属性)。

一个实例的 `__dict__` 属性仅仅是那个实例的实例属性的集合，并不是该实例的所有有效属性。所以要找一个对象的所有有效属性应该使用 `dir`。

`__dict__` 是 `dir()` 的子集。

2.3.9 __slots__

Python 类的实例往往会在程序运行过程中增加成员，这使得我们可以非常灵活地使用实例对象。但如果某个实例的成员过多，在运行过程中维护它的 `__dict__` 字典表会消耗大量的内存。这时我们就可以使用 `__slots__` 魔法函数保存固定的属性，优化内存。

`__slots__` 一般写法如下:

```
__slots__ = ['name', 'identifier']
```

在书写 `__slots__` 时需要注意以下几项:

- 一般使用列表来保存属性，也可以使用集合等。
- 不要乱用这个魔法函数，只有实例属性上千时使用才有显著的优化效果。

2.4 属性描述符

2.4.1 __get__

2.4.2 __set__

2.4.3 __delete__

2.5 序列相关

2.5.1 __len__

2.5.2 __getitem__

2.5.3 __setitem__

2.5.4 __delitem__

2.5.5 __contains__

2.6 迭代相关

2.6.1 __iter__

2.6.2 __next__

2.7 其他

2.7.1 __version__

PEP8 鼓励使用的魔法函数，用于保存模块版本，但很少有人提及。json 包中这样写到:

```
__version__ = '2.0.9'
```

2.7.2 __author__

同样是 PEP8 鼓励使用的魔法函数，用于保存包作者信息。json 包中这样写到:

```
__author__ = 'Bob Ippolito <bob@redivi.com>'
```

3 其它魔法函数

3.1 上下文管理器

3.1.1 __enter__

3.1.2 __exit__

3.2 数值转换

3.2.1 __abs__

3.2.2 __bool__

3.2.3 __int__

3.2.4 __float__

3.2.5 __hash__

3.2.6 __index__

3.3 协程

3.3.1 __await__

3.3.2 __aiter__

3.3.3 __anext__

3.3.4 __aenter__

3.3.5 __aexit__

第二部分

Python 标准库

II 文件库

4 Json

Json 模块是用于读写 json 文件的一个轻量的标准模块。导入方式及内容如下:

```
>>> import json
>>> dir(json)
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', '__all__', '__author__', '__builtins__',
    '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
    '__path__', '__spec__', '__version__', '_default_decoder', '_default_encoder', 'codecs',
    'decoder', 'detect_encoding', 'dump', 'dumps', 'encoder', 'load', 'loads', 'scanner']
>>> json.__all__
['dump', 'dumps', 'load', 'loads', 'JSONDecoder', 'JSONDecodeError', 'JSONEncoder']
```

4.1 基础用法

Json 标准库主要提供了四个方法: `dumps`, `dump`, `loads`, `load` 其中，`dumps`, `loads` 函数不涉及文件，`dump`, `load` 涉及文件。

4.1.1 `dumps`, `dumps`

`dump` 与 `dumps` 函数用于对 Python 对象进行序列化。将一个 Python 对象序列化为 JSON 格式的编码。

`dump` 函数定义:

```
dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
    cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)
```

其中各个参数的意义如下:

- **obj**: 要序列化的对象
- **fp**: 文件描述符，将序列化的 str 保存到文件中。
- skipkeys: 默认为 False，若为 True，则跳过非基本类型的 dict 键。
- ensure_ascii: 默认为 True，将所有传入的非 ASCII 字符转义输出，False 则原样输出。
- check_circular: 默认为 True，若为 False 则跳过对容器类型的循环引用检查。
- allow_nan: 默认为 True，如果为 False 则严格遵守 JSON 规范，引发一些错误，若为 True，则使用错误对象的 JavaScript 等效值。
- indent: 缩进格式，默认最紧凑的方式缩进。

- separators: 去除分隔符后面的空格。
- default: 如果无法序列化，调用对应函数处理。
- sort_key: 如果为 True，则输出按键值排序。

`dumps` 函数定义

```
dumps(obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
    cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)
```

`dumps` 函数除了没有参数 `fp` 其他和 `dump` 函数相同。

4.1.2 `load，loads`

`load，loads` 函数使用反序列的方法将 json 对象解码为 python 对象。

`load` 函数如下:

```
load(fp, *, cls=None, object_hook=None, parse_float=None, parse_int=None,
    parse_constant=None, object_pairs_hook=None, **kw)
```

其中各个参数意义如下:

- **fp**: 文件描述符。
- object_hook: 可选函数，用于实现自定义解码器。指定一个函数，该函数负责把反序列化后的基本类型对象转换成自定义类型的对象。
- parse_float: 用于对 float 字符串进行解码。
- parse_int: 用于对 int 字符串进行解码。
- parse_constant: 用于对 -Infinity Infinity NaN 字符串进行调用。
- object_pairs_hook: 可选函数，暂时不知道干什么的。

`loads` 函数如下:

```
loads(s, *, cls=None, object_hook=None, parse_float=None, parse_int=None,
    parse_constant=None, object_pairs_hook=None, **kw)
```

其中部分参数意义如下:

- **s**: 将 s(包含 JSON 文档的 str,bytes,bytearray 实例) 反序列化为 Python 对象。

4.2 转换规则

Python 原始类型向 Json 类型的转换对照表如下:

表 2.1 json 转换表

Python	Json
dict	object
list,tuple	array
str,unicode	string
int,long,float	number
True/False	true/false
None	null