[笔记][Learning Python][6. 类和面向对象编程]

Python

26. OOP：宏伟蓝图

27. 类代码编写基础

28. 一个更加实际的示例

29. 类代码编写细节

29.4 命名空间：结论

嵌套的类：重温 LEGB 作用域规则
尽管类能够访问外层函数的作用域，但它们不能作为类中其他代码的外层作用域：`Python` 搜索外层函数来访问被引用的名称，但从来不会搜索外层类。也就是说，类是一个可以访问其外层作用域的局部作用域，但其本身却不能作为一个外层作用域被访问。因为方法函数中对名称的搜索跳过了外层的类，所以类属性必须作为对象属性并使用继承来访问。

30. 运算符重载

30.2 索引和切片：`__getitem__` 和 `__setitem__`

拦截切片
在 `3.X` 中 `__getitem__` 也会被切片表达式调用，而在 `2.X` 中如果你不提供更具体的切片方法的话 `__getitem__` 将用于切片表达式。

切片语法只不过就是用切片对象来进行索引的语法糖。
所以你总是可以手动地传入一个切片对象。

```
>>> L[slice(2, 4)]             # Slice with slice objects
[7, 8]
>>> L[slice(1, None)]
[6, 7, 8, 9]
>>> L[slice(None, -1)]
[5, 6, 7, 8]
>>> L[slice(None, None, 2)]
[5, 7, 9]
```

在 `3.X` 中带有 `__getitem__` 的类既可以被基础索引（有一个索引）调用，又能被切片（带有一个切片对象）调用。

```
>>> class Indexer:
        data = [5, 6, 7, 8, 9]
        def __getitem__(self, index):  # Called for index or slice
```

```
            print('getitem:', index)
            return self.data[index]    # Perform index or slice

>>> X = Indexer()
>>> X[0]         # Indexing sends __getitem__ an integer
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: -1
9

>>> X[2:4]       # Slicing sends __getitem__ a slice object
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, -1, None)
[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)
[5, 7, 9]
```

当需要时，`__getitem__` 可以检测它接收的参数类型，并提取切片对象的边界。
切片对象有 `start`、`stop` 和 `step` 这些属性，任何一项被省略的话都是 `None`。

```
>>> class Indexer:
        def __getitem__(self, index):
            if isinstance(index, int):    # Test usage mode
                print('indexing', index)
            else:
                print('slicing', index.start, index.stop, index.step)

>>> X = Indexer()
>>> X[99]
indexing 99
>>> X[1:99:2]
slicing 1 99 2
>>> X[1:]
slicing 1 None None
```

如果你使用了 `__setitem__` 索引赋值方法的话，它能类似地拦截索引赋值和切片赋值。

```
class IndexSetter:
```

```
    def __setitem__(self, index, value):  # Intercept index or slice assi
gnment
...
        self.data[index] = value          # Assign index or slice
```

实际上，`__getitem__` 不只可以在索引和切片中被自动调用，它同时是迭代的一个退路选项。

30.3 索引迭代：`__getitem__`

30.5 成员关系：`__contains__`、`__iter__` 和 `__getitem__`

> 我认为 索引取值、切片取值、索引赋值、切片赋值 是表述清晰的术语，以后在写技术文章的时候可以采用。

30.8 右侧加法和原位置加法：`__radd__` 和 `__iadd__`

```
class Commuter1:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other

    def __radd__(self, other):
        print('radd', self.val, other)
        return other + self.val


x = Commuter1(88)
y = Commuter1(99)

print(x + 1)
"""
add 88 1
```

```
89
"""

print(1 + y)
"""
radd 99 1
100
"""

print(x + y)
"""
add 88 <__main__.Commuter1 object at 0x000002C0C5CAC4E0>
radd 99 88
187
"""
```

> 个人理解：
> 当不同类的实例混合出现在加法表达式时，`Python` 优先选择左侧的那个类的 `__add__` 进行处理，如果处理不了，就会使用右侧的那个类的 `__radd__` 进行处理。

译注：如果把 `__add__` 中的 `return self.val + other` 写成 `return other + self.val`，那么 `x + y` 会如何变化？
我做了相应实验：

```
'Commuter1 的变体'

class Commuter1:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        print('add', self.val, other)
        # return self.val + other
        return other + self.val

    def __radd__(self, other):
        print('radd', self.val, other)
        return other + self.val


x = Commuter1(88)
y = Commuter1(99)

print(x + 1)
print()
"""
add 88 1
89
```

```
"""

print(1 + y)  # 这里调用 1 的 __add__ 行不通，所以调用 y 的 __radd__
print()
"""
radd 99 1
100
"""

print(x + y)
print()
# 首先调用 x 的 __add__
# 变成 y + 88
# 然后调用 y 的 __add__
# 变成 88 + 99
# 所以出现了两次 add
"""
add 88 <__main__.Commuter1 object at 0x0000021A64DFC4E0>
add 99 88
187
"""
```

与译注一致，会出现两次 `add`，原因在代码中写了。
译注又说，如果把 `__radd__` 中的语句 `return other + self.val` 改写成 `return self.val + other` 有影响吗？答案是无影响，实验代码如下：

```
class Commuter1:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other

    def __radd__(self, other):
        print('radd', self.val, other)
        # return other + self.val
        return self.val + other


x = Commuter1(88)
y = Commuter1(99)

print(x + 1)
print()
"""
add 88 1
89
"""
```

```
print(1 + y)
print()
"""
radd 99 1
100
"""

print(x + y)
print()
"""
add 88 <__main__.Commuter1 object at 0x000001CFDDE9C4E0>
radd 99 88
187
"""
# 这里与原来结果一样
# 因为 x + y 首先看 x 的 __add__ 能否处理
# 能处理，输出 add
# 变成 88 + y
# 然后 88 的 __add__ 无法处理，所以看 y 的 __radd__ 能否处理
# 能处理，输出 radd
# 变成 99 + 88
# 输出 187
```

总之把握住刚才写的一点即可：
当不同类的实例混合出现在加法表达式时，`Python` 优先选择左侧的那个类的 `__add__` 进行处理，如果处理不了，就会使用右侧的那个类的 `__radd__` 进行处理。

类类型的传播
类类型可能需要作为结果传播。

> `propagate` 传播

```
class Commuter5:  # Propagate class type in results
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        if isinstance(other, Commuter5):  # Type test to avoid object nes
ting
            other = other.val
        return Commuter5(self.val + other)  # Else + result is another Co
mmuter

    def __radd__(self, other):
        return Commuter5(other + self.val)

    def __str__(self):
```

```
        return '<Commuter5: %s>' % self.val

x = Commuter5(88)
y = Commuter5(99)

print(x + 10)
print()
"""
<Commuter5: 98>
"""

print(10 + y)
print()
"""
<Commuter5: 109>
"""

z = x + y
print(z)
print()
"""
<Commuter5: 187>
"""
# 如果不进行类型判断，会变成
# <Commuter5: <Commuter5: 187>>

print(z + 10)
print()
"""
<Commuter5: 197>
"""
# 如果不进行类型判断，会变成
# <Commuter5: <Commuter5: 197>>
```

> `commutative` 书中翻译成 “对易性”，我觉得 “可交换的” 更好些或者叫 “互换性”。

30.9 调用表达式 `__call__`

> 个人理解：注册 这个名词的意思，就是传入一个能够适配 `API` 的函数。
> 比如 “把某某某注册成回调函数”，或者 “把某某某注册成事件处理器 `handler`”。

---

31. 类的设计

31.1 Python 和 OOP

多态意味着接口，不是函数调用签名

> Some OOP languages also define polymorphism to mean overloading functions based on the type signatures of their arguments—the number passed and/or their types.

`C++` 的美好时光在 `Python` 中行不通：

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

31.6 方法是对象：绑定或未绑定

在 Python 3.X 中，未绑定方法是函数

```
class Test:
    def test():
        print('hello')


Test.test()

# Python 3.X
"""
hello
"""

# Python 2.X
"""
Traceback (most recent call last):
  File "C:/Users/jpch89/Desktop/test.py", line 5, in <module>
    Test.test()
TypeError: unbound method test() must be called with Test instance as fir
st argument (got nothing instead)
"""
```

31.8 多继承：“mix-in” 类

`getattr` 使用了继承搜索协议
改良版本的 `__attrnames` 函数：它对双下划线变量名单独分组，并对长属性值自动换行，注意它是如何使用 `%%` 来转义一个百分号 `%` 的。

> 其实是截断了长属性值

```
def __attrnames(self, indent=' '*4):
    result = 'Unders%s\n%s%%s\nOthers%s\n' % ('-'*77, indent, '-'*77)
    unders = []
    for attr in dir(self): # Instance dir()
        if attr[:2] == '__' and attr[-2:] == '__': # Skip internals
            unders.append(attr)
        else:
            display = str(getattr(self, attr))[:82-(len(indent) + len(att
r))]
            result += '%s%s=%s\n' % (indent, attr, display)
    return result % ', '.join(unders)
```

因为类对象是可哈希化的，所以它们可以作为字典键；集合也可以提供类似的功能。

> 我专门为这个做了个测试，的确如此。

```
class A:
    pass


class B:
    pass


class C:
    pass


d = dict()
d[A] = True
d[B] = False
d[C] = '你好啊'

print(d)
"""
```

```
{<class '__main__.A'>: True, <class '__main__.B'>: False, <class '__main_
_.C'>: '你好啊'}
"""
```

技术上讲，类继承树中的继承循环一般不太可能出现——类在用作父类之前必须已经被定义。如果你试图修改 `__bases__` 来创建一个循环，`Python` 一般会引发异常。

```
>>> class C: pass
...
>>> class B(C): pass
...
>>> C.__bases__ = (B, )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a __bases__ item causes an inheritance cycle
```

实际上，这一测试甚至在当前一些最新的 `3.X` 发行版中无法运行。
因为 `str.format` 调用不再支持一些内置属性，所以最好省略这些名称的属性。

```
c:\code> py -3.1
>>> '{0}'.format(object.__reduce__)
"<method '__reduce__' of 'object' objects>"
c:\code> py -3.3
>>> '{0}'.format(object.__reduce__)
TypeError: Type method_descriptor doesn't define __format__
```

> 经我测试，`3.6.6` 版本可以：

```
>>> import sys
>>> sys.version_info
sys.version_info(major=3, minor=6, micro=6, releaselevel='final',
serial=0)
>>> sys.version
'3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD
64)]'
>>> '{0}'.format(object.__reduce__)
"<method '__reduce__' of 'object' objects>"
```

奇怪的是 `{0}` 和 `{0:s}` 字符串目标都失败了。
但是手动 `str` 转换和 `{0!s}` 可以。

```
c:\code> py -3.3
>>> '{0:s}'.format(object.__reduce__)
TypeError: Type method_descriptor doesn't define __format__
```

```
>>> '{0!s}'.format(object.__reduce__)
"<method '__reduce__' of 'object' objects>"

>>> '{0}'.format(str(object.__reduce__))
"<method '__reduce__' of 'object' objects>"
```

修复方法：使用 `%` 或者用 `try` 捕获异常。

```
c:\code> py -3.3
>>> '%s' % object.__reduce__
"<method '__reduce__' of 'object' objects>"
```

树列举器的代码，可以这么改：

```
result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))
result += spaces + '%s=%s\n' % (attr, getattr(obj, attr))
```

`2.7` 同样退化了，显然是继承了 `3.2` 版本的修改。

> 所以说 `format` 这种新功能往往是不稳定的。

用法变化：在大型模块上运行
通常，我们需要在一个 `class` 的头部首先列出 `ListTree`，在最左端，这样它的 `__str__` 方法才会被选取。在多继承最左端的父类总是被优先搜索。

---

32. 类的高级主题

32.1 扩展内置类型

对所有这里没有定义的方法的调用，都会被直接路又道 `list` 中的方法那里去：

> 注意：这里的 `list__init__([])` 这一句话实属多余

```
from __future__ import print_function  # 2.X compatibility


class Set(list):
    def __init__(self, value=[]):      # Constructor
        list.__init__([])              # Customizes list
```

```
            self.concat(value)              # Copies mutable defaults

        def intersect(self, other):         # other is any sequence
            res = []                        # self is the subject
            for x in self:
                if x in other:              # Pick common items
                    res.append(x)
            return Set(res)                 # Return a new Set

        def union(self, other):             # other is any sequence
            res = Set(self)                 # Copy me and my list
            res.concat(other)
            return res

        def concat(self, value):            # value: list, Set, etc.
            for x in value:                 # Removes duplicates
                if not x in self:
                    self.append(x)

        def __and__(self, other): return self.intersect(other)

        def __or__(self, other): return self.union(other)

        def __repr__(self): return 'Set:' + list.__repr__(self)


    if __name__ == '__main__':
        x = Set([1, 3, 5, 7])
        y = Set([2, 1, 4, 5, 6])
        print(x, y, len(x))
        print(x.intersect(y), y.union(x))
        print(x & y, x | y)
        x.reverse()
        print(x)

    """
    Set:[1, 3, 5, 7] Set:[2, 1, 4, 5, 6] 4
    Set:[1, 5] Set:[2, 1, 4, 5, 6, 3, 7]
    Set:[1, 5] Set:[1, 3, 5, 7, 2, 4, 6]
    Set:[7, 5, 3, 1]
    """
```

32.3 新式类变化

MRO 算法

`MRO` 的核心工作原理如下：

1. 采用经典类的 `DFLR` 查找规则来列出一个实例继承的所有父类，同时如果一个类被多次访问的话也相应列举所有的出现。
2. 在上一步列出的列表中扫描重复的类，依次删除并保留每个类的最后一次出现。
3. 最终一个给定的 `MRO` 列表包括了这个类本身、它的父类、以及直到继承树顶端 `object` 的所有高级父类。在这个列表中，每个类都出现在它的父类之前，而且多个父类保持了它们在 `__base__` 元组中的出现顺序。

> 其实 `__mro__` 也是个元组，不是列表。而且只有类才有 `__mro__` 属性，实例是没有的！我为此做了一个实验：

```
class A:
    pass


class B(A):
    pass


class C(A):
    pass


class D(B, C):
    pass


print(D.__bases__)
print(type(D.__bases__))
print()

print('-' * 30)
print()
print(D.__mro__)
print(type(D.__mro__))

"""
(<class '__main__.B'>, <class '__main__.C'>)
<class 'tuple'>

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

(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
 '__main__.A'>, <class 'object'>)
<class 'tuple'>
"""
```

`super` 调用可以使用 `MRO` 列表中的下一个类，而这个类不一定是一个父类。

跟踪 `MRO`

```
>>> class A: pass
>>> class B(A): pass       # Diamonds: order differs for newstyle
>>> class C(A): pass       # Breadth-first across lower levels
>>> class D(B, C): pass
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>)
```

> `infer` 推断，推论
> `preclude` 妨碍；阻止

32.5 类方法和静态方法

为什么使用特殊方法

> 下面是个人总结：
>
> - 普通函数：只能通过类调用。
> - 静态方法：可以通过类和实例调用，在此过程中，`Python` 不会自动传参，当然也不用手动传递类或者实例作为参数。
> - 类方法：可以通过类和实例调用，`Python` 会自动传递类给 `cls` 形参。
> - 实例方法：可以通过类和实例调用。通过实例调用时，`Python` 会自动传递实例给 `self` 形参；通过类调用时，需要手动传递一个实例给 `self` 形参，否则会报错。

```
class Test:
    def normal_function():
        print('我是普通函数！')

    @staticmethod
    def static_method():
        print('我是静态方法！')

    @classmethod
    def class_method(cls):
        print(f'我是{cls}的类方法！')

    def instance_method(self):
        print(f'我是{self}的实例方法！')


t = Test()
```

```
# 普通函数：只能通过类名调用，不用手动传参
# t.normal_function()
"""
Traceback (most recent call last):
  File "test.py", line 19, in <module>
    t.normal_function()
TypeError: normal_function() takes 0 positional arguments but 1 was given
"""
Test.normal_function()
"""
我是普通函数!
"""

# 静态方法：可以通过实例和类名调用，不用手动传参
t.static_method()
Test.static_method()
"""
我是静态方法!
我是静态方法!
"""

# 类方法：可以通过实例和类名调用，自动传递类给 cls 形参
# 注意：即使通过实例调用类方法，Python 自动传递的也是类，而不是实例
t.class_method()
Test.class_method()
"""
我是<class '__main__.Test'>的类方法!
我是<class '__main__.Test'>的类方法!
"""

# 实例方法：
# 通过实例调用，自动传递实例给 self 形参
# 通过类调用，需要手动传递一个实例给 self 形参
t.instance_method()
Test.instance_method(t)
"""
我是<__main__.Test object at 0x00000189E6A3AC88>的实例方法!
我是<__main__.Test object at 0x00000189E6A3AC88>的实例方法!
"""
# 通过类调用，如果不手动传递实例给 self 形参，会缺少参数的错误
# Test.instance_method()
"""
Traceback (most recent call last):
  File "test.py", line 59, in <module>
    Test.instance_method()
TypeError: instance_method() missing 1 required positional argument: 'sel
f'
"""
```

注意：一个子类继承了父类，子类没有 `__init__` ，会调用父类的 `__init__`

```
class A:
    def __init__(self):
        print(f'{self}的初始化函数')

class B(A):
    pass

b = B()
print()
"""
<__main__.B object at 0x0000022A185CC1D0>的初始化函数
"""

a = A()
"""
<__main__.A object at 0x0000022A185CC588>的初始化函数
"""
```

32.7 super 内置函数：更好还是更糟

super内置函数：更好还是更糟

```
class C:
    def act(self):
        print('spam')


class E(C):
    def method(self):
        proxy = super()
        print(proxy)
        proxy.act()


E().method()

"""
<super: <class 'E'>, <E object>>
spam
"""
```

> 经我试验，`super()` 调用在方法里面可以成功得到一个代理对象。
> 但是直接在类级别下面使用 `super()` 会报错：`RuntimeError: super(): no arguments`

定制：方法替代

```
class A:
    def method(self):
        print('A.method')
        super().method()


class B(A):
    def method(self):
        print('B.method')
        super().method()


class C:
    def method(self):
        print('C.method')


class D(B, C):
    def method(self):
        print('D.method')
        super().method()


X = D()
X.method()
"""
D.method
B.method
A.method
C.method
"""
```