# 《源码探秘 CPython》78. 魔法方法都有哪些？作用是什么？

原创 古明地觉 古明地觉的编程教室 2022-04-28 08:30 发表于北京

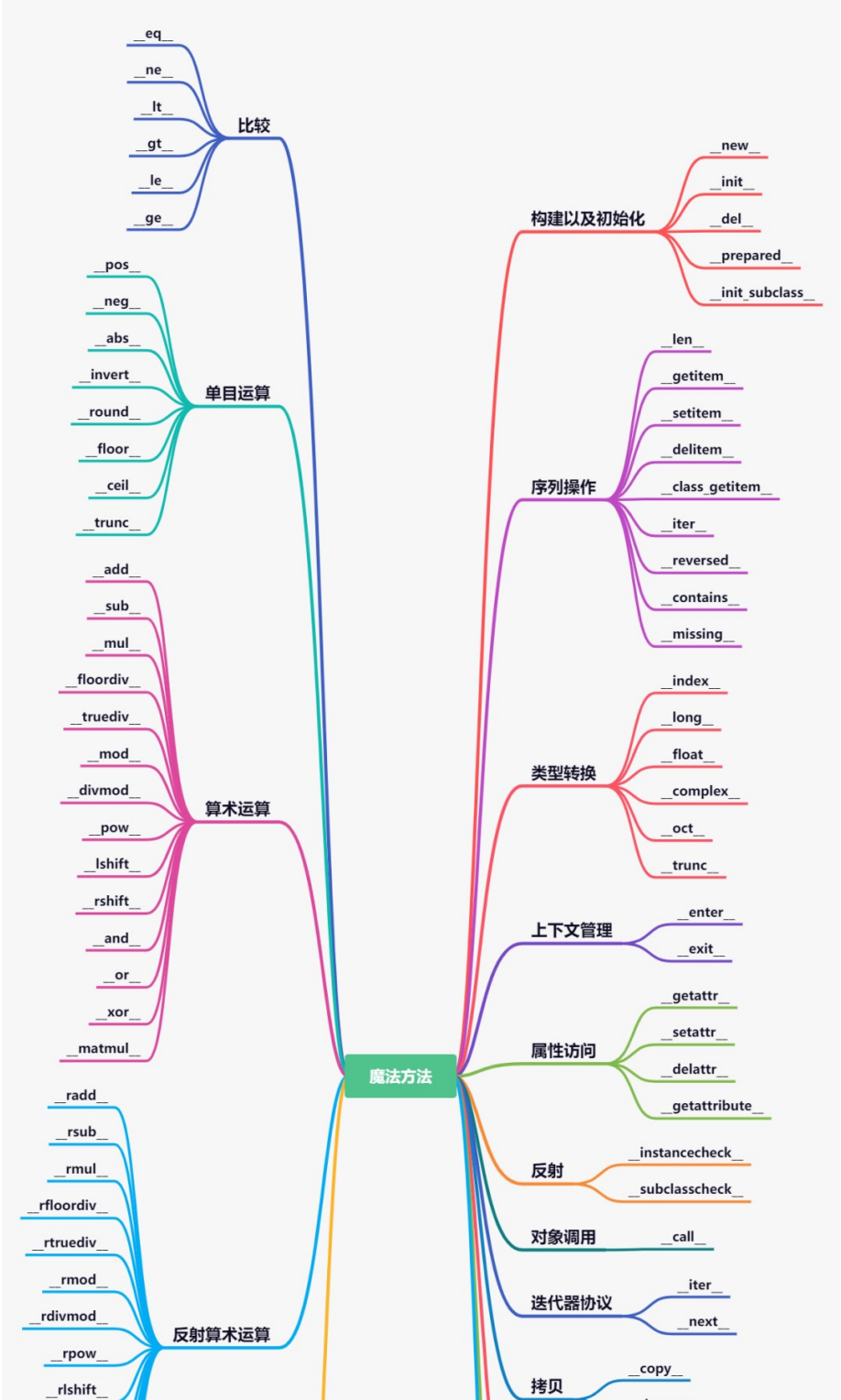
微信扫一扫
关注该公众号

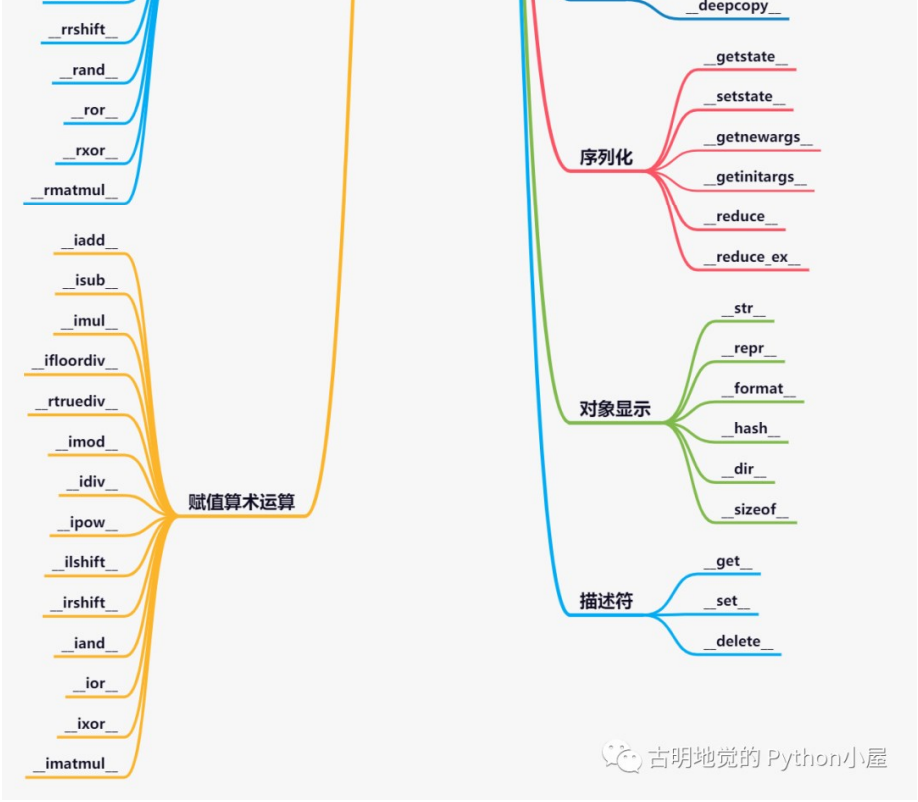
收录于合集
#CPython
97个 >

楔子

下面来看一下Python的魔法方法，我们知道Python将每个操作符都抽象成了魔法方法(magic method)，实例对象进行操作时，实际上会调用魔法方法。也正因为如此，numpy 才得以很好的实现。

那么魔法方法都有哪些呢？我们按照特征分成了几类，先大致浏览一下，然后再举例说明它们的用法。






注意：有一些方法是 Python2 中的，但是在 Python3 里面依然存在，但是不推荐使用了。比如：__cmp__、__coerce__、__delslice__等等，我们就没有画在图中。

下面我们就来介绍一下这些魔法方法的实际用途，当然啦，里面的一部分方法在前面已经说过了，这里就不再说了。

## 构建以及初始化

这里面只有一个 __del__ 没有说，它是做什么的呢？

```
class Girl:

    def __del__(self):
        print("对象被销毁了")

g = Girl()
del g  # 对象被销毁了
```

__del__被称为析构函数，当一个实例对象被销毁时会调用该函数。如果没有销毁，那么程序结束时也会调用。

## 比较操作

Python的比较操作符也抽象成了魔法方法。

```
class Girl:

    def __init__(self, name):
        self.name = name
```

```
5
6     def __eq__(self, other):
7         return "==", self.name, other.name
8
9     def __ne__(self, other):
10        return "!=", self.name, other.name
11
12    def __le__(self, other):
13        return "<=", self.name, other.name
14
15    def __lt__(self, other):
16        return "<", self.name, other.name
17
18    def __ge__(self, other):
19        return ">=", self.name, other.name
20
21    def __gt__(self, other):
22        return ">", self.name, other.name
23
24 girl1 = Girl("girl1")
25 girl2 = Girl("girl2")
26
27 print(girl1 == girl2)  # ('==', 'girl1', 'girl2')
28 print(girl1 != girl2)  # ('!=', 'girl1', 'girl2')
29 print(girl1 < girl2)  # ('<', 'girl1', 'girl2')
30 print(girl2 <= girl1)  # ('<=', 'girl2', 'girl1')
31 print(girl2 > girl1)  # ('>', 'girl2', 'girl1')
32 print(girl2 >= girl1)  # ('>=', 'girl2', 'girl1')
```

我们看到如果是 a > b，那么会调用 a 的 __gt__ 方法，self 就是 a、other 就是 b；如果是 b > a，那么调用 b 的 __gt__ 方法，self 就是 b、other 就是a。也就是说谁在前面，就调用谁的魔法方法。


下面再来看看单目运算。

```
1  class Girl:
2
3      # +self 的时候调用
4      def __pos__(self):
5          return "__pos__"
6
7      # -self 的时候调用
8      def __neg__(self):
9          return "__neg__"
10
11     # abs(self) 的时候会调用, 也可以是np.abs(self), 但不推荐numpy调用
12     def __abs__(self):
13         return "__abs__"
14
15     # ~self 的时候调用
16     def __invert__(self):
17         return "__invert__"
18
19     # round(self, n) 的时候调用
20     def __round__(self, n=None):
21         return f"__round__, {n}"
```

```
22
23      # math.floor(self)的时候调用
24      # 也可以是np.floor(self), 但不推荐numpy调用
25      def __floor__(self):
26          return "__floor__"
27
28      # math.ceil(self)的时候调用
29      # 也可以是np.ceil(self), 但不推荐numpy调用
30      def __ceil__(self):
31          return "__ceil__"
32
33      # math.trunc(self)的时候调用
34      # 也可以是np.trunc(self), 或者int(self)
35      # 但不推荐numpy调用
36      def __trunc__(self):
37          return "__trunc__"
38
39  girl = Girl()
40
41  # 1. +girl 触发 __pos__
42  print(+girl)  # __pos__
43
44  # 2. -girl触发 __neg__
45  print(-girl)  # __neg__
46  #注意: 不可以写成 0 + girl 和 0 - girl
47  # 尽管我们知道在数学上这与 girl 和 -girl 是等价的
48
49  # 3. abs(girl)或者np.abs(girl)触发 __abs__
50  print(abs(girl))  # __abs__
51
52  # 4. ~girl 触发 __invert__
53  print(~girl)  # __invert__
54
55  # 5. round(girl) 触发 __round__
56  print(round(girl))  # __round__, None
57  print(round(girl, 2))  # __round__, 2
58
59  # 6. math.floor(girl) np.floor(girl) 触发 __floor__
60  import math, numpy as np
61  print(math.floor(girl))  # __floor__
62  print(np.floor(girl))  # __floor__
63
64  # 7. math.ceil(girl), np.ceil(girl)触发 __ceil__
65  print(math.ceil(girl))  # __ceil__
66  print(np.ceil(girl))  # __ceil__
67
68  # 8. math.trunc(girl), np.trunc(girl)触发 __trunc__
69  print(math.trunc(girl))  # __trunc__
70  print(np.trunc(girl))  # __trunc__
71  # __trunc__表示截断, 只保留整数位, 所以int(girl)也是可以触发的
72  # 但如果是int(girl)这种方式, 它要求__trunc__必须返回一个数值
73  try:
74      int(girl)
75  except Exception as e:
76      print(e)  # __trunc__ returned non-Integral (type str)
77  Girl.__trunc__ = lambda self: 666
78  print(int(girl))  # 666
```

以上便是单目运算的一些魔法方法，但说实话除了 __pos__、__neg__、__invert__ 之外，其它不太常用。因为我们可能希望一些操作的调用方式尽可能简单，所以会通过重写 +、-、~ 操作符对应的魔法方法，来赋予实例对象一些特殊的含义。

# 算术运算

算术运算是比较常用的了，我们来看看算数运算对应的魔法方法。

```
class Girl:

    # a + b 的时候调用, self就是a、other就是b
    def __add__(self, other):
        return "__add__"

    # a - b 的时候调用, self就是a、other就是b
    def __sub__(self, other):
        return "__sub__"

    # a * b 的时候调用, self就是a、other就是b
    def __mul__(self, other):
        return "__mul__"

    # a // b 的时候调用, self就是a、other就是b
    def __floordiv__(self, other):
        return "__floordiv__"

    # a / b 的时候调用, self就是a、other就是b
    def __truediv__(self, other):
        return "__truediv__"

    # a % b 的时候调用, self就是a、other就是b
    def __mod__(self, other):
        return "__mod__"

    # divmod(a, b) 的时候调用, self就是a、other就是b
    def __divmod__(self, other):
        return "__divmod__"

    # a ** b 的时候调用, self就是a、other就是b
    # pow(a, b) 的时候也会调用
    def __pow__(self, power, modulo=None):
        return "__pow__"

    # a << b 的时候调用, self就是a、other就是b
    def __lshift__(self, other):
        return "__lshift__"

    # a >> b 的时候调用, self就是a、other就是b
    def __rshift__(self, other):
        return "__rshift__"

    # a & b 的时候调用, self就是a、other就是b
    def __and__(self, other):
        return "__and__"

    # a | b 的时候调用, self就是a、other就是b
    def __or__(self, other):
        return "__or__"

    # a ^ b 的时候调用, self就是a、other就是b
    def __xor__(self, other):
        return "__xor__"

```

```
    # a @ b 的时候调用, self就是a、other就是b
    def __matmul__(self, other):
        # 这个方法是用于矩阵运算
        # Python在3.5版本的时候将@抽象成了这个方法
        # 比如numpy的两个数组如果想进行矩阵之间的相乘
        # 除了np.dot(arr1, arr2)之外, 还可以直接arr1 @ arr2
        return "__matmul__"

girl1 = Girl()
girl2 = Girl()

print(girl1 + girl2)  # __add__
print(girl1 - girl2)  # __sub__
print(girl1 * girl2)  # __mul__
print(girl1 // girl2)  # __floordiv__
print(girl1 / girl2)  # __truediv__
print(girl1 % girl2)  # __mod__
print(divmod(girl1, girl2))  # __divmod__
print(girl1 ** girl2)  # __pow__
print(pow(girl1, girl2))  # __pow__
print(girl1 << girl2)  # __lshift__
print(girl1 >> girl2)  # __rshift__
print(girl1 & girl2)  # __and__
print(girl1 | girl2)  # __or__
print(girl1 ^ girl2)  # __xor__
print(girl1 @ girl2)  # __matmul__
```

常见的算术运算大概就是上面这些，还是很简单的。

## 反射算术运算

反射算术运算指的是什么呢？比如: a + b，我们知道会调用 a 的 __add__。但如果 type(a) 中没有定义 __add__，那么会尝试寻找 b 的 __radd__。

```
class Girl1:

    def __add__(self, other):
        print(self)
        print(other)
        return "Girl1: __add__"

class Girl2:

    def __radd__(self, other):
        print(self)
        print(other)
        return "Girl2: __radd__"

girl1 = Girl1()
girl2 = Girl2()

# girl1 + girl2 会调用 Girl1 的 __add__
# 里面的 self 就是 girl1、other 就是 girl2
print(girl1 + girl2)
"""
<__main__.Girl1 object at 0x000002188CCBD040>
<__main__.Girl2 object at 0x000002188CCBD070>
Girl1: __add__
```

```
"""

# 但如果 Girl1 里面没有 __add__ 怎么办？
# 显然这个时候会去找 Girl2 的 __radd__
# 里面的 self 就是 girl2、other 就是 girl1
# 虽然 girl2 位于 + 后面，但调用的是它的魔法方法，所以self就是它
delattr(Girl1, "__add__")
print(girl1 + girl2)
"""
<__main__.Girl2 object at 0x0000027A062EE070>
<__main__.Girl1 object at 0x0000027A062EE040>
Girl2: __radd__
"""

# 再比如内置实例对象也是可以的
# 此时 other 就变成了 123
print(123 + girl2)
"""
<__main__.Girl2 object at 0x00000219065EE070>
123
Girl2: __radd__
"""

# 但是注意：123 + girl1 不可以
# 因为 girl1 没有 __radd__，它只有 __add__
# 所以，如果想使表达式合法，那么应该写成 girl1 + 123
```

相比算术运算，反射算术运算的名称只是前面多一个字母 r，至于区别也是我们说的那样。优先寻找操作符左边的对象的算术运算操作，如果没有，退而求其次，寻找操作符右边的对象的反射算术运算操作。并且调用的是谁的操作，self 就是谁，与它们是在操作符的左边还是右边没有关系。

以上就是 __radd__，其它的魔法方法也是类似的。

## 赋值算术运算

赋值算术运算适用于类似于 += 这种形式，比如：

```
class Girl:

    def __iadd__(self, other):
        print(self, other)
        return ("ping", "pong")

girl = Girl()
girl += 123
print(girl)
"""
<__main__.Girl object at 0x000...> 123
('ping', 'pong')
"""
```

比较简单，其它的也与此类似。

## 序列操作

下面我们看看序列操作。

```
class Girl:

    def __len__(self):
        # 必须返回整数
        return 123

girl = Girl()
print(len(girl))  # 123

# 此外，__len__还有充当布尔值的作用
print(bool(girl))  # True
Girl.__len__ = lambda self: 0
print(bool(girl))  # False

# 当然真正起到决定性作用的是__bool__方法，
# 如果定义了__bool__，那么以__bool__的返回值为准
# 没有__bool__，那么解释器会退化，寻找__len__
Girl.__bool__ = lambda self: True
print(bool(girl))  # True
```

所以解释器具有退化功能，会优先寻找某个方法，但如果没有，那么会退化寻找替代方法。在后面，我们还会看到类似的实现。

```
class Girl:

    def __getitem__(self, item):
        print(item)

    def __setitem__(self, key, value):
        print(key, value)

    def __delitem__(self, key):
        print(key)

girl = Girl()

# 上面三个操作符可以让我们像操作字典一样，操作实例对象
girl["xxx"]  # xxx
girl["xxx"] = "yyy"  # xxx yyy
del girl["aaa"]  # aaa

# 不仅如此，它们还可以作用于切片
girl[3: 4]  # slice(3, 4, None)
girl["嘿": "蛤": "哼"]  # slice('嘿', '蛤', '哼')
girl["□": "□": "□"] = "□"  # slice('□', '□', '□') □
del girl["□", "青", "回"]  # ('□', '青', '回')
```

另外，__getitem__ 还可以让实例支持 for 循环。我们知道 for 循环的时候，会去找 __iter__，但如果找不到，会退化寻找 __getitem__。

```
class Girl:

    def __reversed__(self):
        return "__reversed__"

    def __contains__(self, item):
        # a in b 本质上是 b.__contains__(a)
        # 这里返回的如果不是布尔值，那么会转成布尔值再返回
        return item
```

```
10
11 girl = Girl()
12 print(reversed(girl))  # __reversed__
13 print("xxx" in girl)  # xxx
14 print("" in girl)  # False
```

最后一个__missing__比较特殊，它是针对于字典的，我们来看一下。

```
1  class Girl(dict):
2
3      def __missing__(self, key):
4          return str(key).upper()
5
6  girl = Girl({"name": "古明地觉"})
7  print(girl["name"])  # 古明地觉
8  print(girl["Name"])  # NAME
9
10 # 当我们获取元素时，首先调用__getitem__
11 # 由于我们没有重写，显然调用父类的__getitem__
12 # 如果获取到结果，那么直接返回
13 # 获取不到，那么会去寻找调用__missing__
14 # 如果没有__missing__则报错，有的话则是__missing__的返回值
```

因此这个 __miss__ 一定要定义在继承字典的子类里面才有意义。


很简单的内容了，我们直接来看一下。

```
1  class Girl:
2
3      def __int__(self):
4          return 123
5
6      def __index__(self):
7          return 789
8
9      def __float__(self):
10         return 3.14
11
12     def __complex__(self):
13         return 1 + 3j
14
15 girl = Girl()
16
17 # __int__ 和 __index__ 的作用类似
18 # 都是在执行 int(self) 时候调用
19 # 但是存在一个优先级，默认是__int__
20 print(int(girl))  # 123
21 # 如果没有__init__，执行__index__
22 del Girl.__int__
23 print(int(girl))  # 789
24
25 # 调用 float(girl) 会执行 __float__
26 # 必须返回浮点数
27 print(float(girl))  # 3.14
28
29 # 调用 complex(girl) 会执行 __complex__
```

```
# 必须返回复数
print(complex(girl))  # (1+3j)
```

## 上下文管理

上下文管理器(context manager)负责管理一个代码块中的资源，会在进入代码块时创建资源，然后在退出代码块时清理资源。比如：文件就支持上下文管理器API，可以确保文件读写后关闭文件。

那么 Python 里面如何实现上下文管理呢?

```
class Girl:
    name = "古明地觉"

    def __enter__(self):
        print("有了 __enter__ 就可以使用 with 了")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("with 结束后会执行 __exit__")

with Girl() as g:
    print(g.name)
"""
有了 __enter__ 就可以使用 with 了
古明地觉
with 结束后会执行 __exit__
"""
```

我们看到 **with Girl() as g:** 的流程就是，先创建一个 Girl 的实例对象，然后通过实例对象来调用 __enter__ 方法，将其返回值赋值给 with 语句中的 g；然后执行 with 语句块中的代码，最后执行 __exit__ 方法。

需要注意的是，**with Girl() as g:** 中的 g 是什么，取决于 __enter__ 返回了什么。

```
class Girl:
    name = "古明地觉"

    def __enter__(self):
        return "古明地恋"

    def __exit__(self, exc_type, exc_val, exc_tb):
        pass

with Girl() as g:
    print(g)
"""
古明地恋
"""
```

我们看到 print(g) 打印的是一个字符串，因为__enter__中返回的就是一个字符串。**with Girl() as g:** 这一行代码所做的事情就是先创建一个 Girl 实例，只不过这个实例对象我们没有用变量进行接收，但它确实存在。

然后该实例对象再调用__enter__，将__enter__的返回值赋值给 g，因此__enter__返回了什么，这个 g 就是什么，所以在 with 代码块中打印 g 得到的是一个字符串。

所以要记住: as 后面的变量 g 是由 __enter__ 的返回值决定的，只不过大多数情况下，__enter__ 里面返回的都是 self，所以相应的 g 指向的也是该类的实例对象。

```
# with Girl() as g 这种方式是将实例化和调用 __enter__ 合为一步了
# 我们也可以先实例化一个对象，然后再使用with
g = Girl()
# 这里会将 g.__enter__() 赋值给 g1
with g as g1:
    print(g1)  # 古明地恋
    print(g1.__class__)  # <class 'str'>

# 当然 with 语句中也可以不出现 as
with g:
    pass
```

因此一个对象究竟能否使用with语句，取决于实例化该对象的类(或者继承的基类)中是否同时实现了 __enter__ 和 __exit__，两者缺一不可。所以with语句的流程我们就很清晰了，以**with XXX() as xx:**为例，总共分为三步：

- 创建 XXX 的实例对象，然后调用它 __enter__ 方法，将其返回值交给 as 后面的 xx；
- 执行 with 语句块的代码；
- 最后由该实例对象再调用 __exit__ 进行一些收尾工作；

__enter__ 我们清楚了，但是我们发现 __exit__ 里面除了 self 之外，还有三个参数分别是 exc_type, exc_val, exc_tb，它们是做什么的呢？显然这三个参数分别是异常类型、异常值、异常的回溯栈， 从名字上也能看出来。

```
class Girl:

    def __enter__(self):
        return "古明地恋"

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(exc_type)
        print(exc_val)
        print(exc_tb)
        return True

with Girl() as g:
    print(g)
"""
古明地恋
None
None
None
"""

# 我们看到exc_type, exc_val, exc_tb三者全部为None
# 由于它们是和异常相关，而我们这里没有出现异常，所以为None
# 但如果出现异常了呢？
with Girl() as g:
    print(g)
    1 / 0
    print(123)
    print(456)
    print(789)
print("你猜我会被执行吗？")
"""
古明地恋
<class 'ZeroDivisionError'>
division by zero
<traceback object at 0x00000226EC1FC7C0>
你猜我会被执行吗？
```

```
37 """
```

我们看到在没有出现异常的时候，exc_type, exc_val, exc_tb打印的值全部是None。然而一旦with语句里面出现了异常，那么会立即执行__exit__，并将 异常的类型，异常的值，异常的回溯栈传入到__exit__中。

因此：当with语句正常结束之后会调用__exit__，如果with语句里面出现了异常则会立即调用__exit__。但是__exit__返回了个True是什么意思呢？

当with语句里面出现了异常，理论上是会报错的，但是由于要执行__exit__，所以相当于暂时把异常塞进了嘴里。如果__exit__最后返回了一个布尔类型为True的值，那么会把塞进嘴里的异常吞下去，程序不报错正常执行。如果返回布尔类型为False的值，会在执行完__exit__之后再把嘴里的异常吐出来，引发程序崩溃。

这里我们返回了True，因此程序正常执行，最后一句话被打印了出来。但是 **1 / 0** 这句代码后面的几个print却没有打印，为什么呢？

因为我们说上下文管理执行是有顺序的：1) 先创建Girl的实例对象，调用__enter__方法，将__enter__的返回值交给 g；2) 执行with语句块的代码；3) 最后调用__exit__。

只要__exit__执行结束，那么这个with语句就算结束了。而with语句里面如果有异常，那么会立即进入__exit__，因此异常语句后面的代码是无论如何都不会被执行的。

## 属性访问

主要是 __getattr__、__setattr__、__delattr__ 这几个方法。

```python
class Girl:

    def __getattr__(self, item):
        # 当访问一个不存在的属性时
        # 会触发 __getattr__
        print(f"你访问了 {item} 属性")

    def __setattr__(self, key, value):
        print(f"你将属性 {key} 设置为 {value}")

    def __delattr__(self, item):
        print(f"你删除了 {item} 属性")

g = Girl()
g.name
g.name = "satori"
del g.name
"""
你访问了 name 属性
你将属性 name 设置为 satori
你删除了 name 属性
"""
```

getattr、setattr、delattr这几个内置函数本质上也是调用这几个魔法方法，只不过它额外做了一些其它的工作。以getattr为例：

```python
class Girl:

    def __init__(self):
        self.name = "古明地觉"
```

```
    def __getattr__(self, item):
        return f"你访问了 {item} 属性"

g = Girl()
# 会触发 __getattr__
print(getattr(g, "name"))  # 古明地觉
print(getattr(g, "age"))  # 你访问了 age 属性

# getattr 会先从属性字典中获取
# 属性字典如果没有，那么会执行 __getattr__
```

当然啦， getattr 还可以指定第三个参数，也就是默认值。

```
# 在找不到属性时，会返回默认值
print(getattr(123, "xxx", "哼哼"))  # 哼哼

class Girl:

    def __init__(self):
        self.name = "古明地觉"

    def __getattr__(self, item):
        print(f"你访问了不存在的 {item} 属性")
        raise AttributeError

g = Girl()

# getattr 会从属性字典中查找属性，找到了就返回
# 如果找不到，那么会执行 __getattr__
# 如果在 __getattr__ 中 raise AttributeError
# 那么会返回 getattr 中的默认值
print(getattr(g, "xxx", "yyy"))
"""
你访问了不存在的 xxx 属性
yyy
"""
```

总结一下，属性访问和 getattr 函数本质上是类似的，只是当属性以字符串的形式出现时（事先并不知道要访问的属性叫什么），我们需要使用 getattr。而不管哪一种方式，都是从属性字典中查找属性，但如果访问了一个不存在的属性，会那么触发 __getattr__。

然后 getattr 还可以接收第三个参数，如果调用 __getattr__ 时出现了 AttributeError，那么会返回默认值。

最后是 __getattribute__，我们说当访问一个不存在的属性时，会触发 __getattr__。而 __getattribute__，不管属性存不存在都会先经过它。

```
class Girl:

    def __init__(self):
        self.name = "古明地觉"

    def __getattr__(self, item):
        print(f"__getattr__")

    def __getattribute__(self, item):
        print(f"__getattribute__")

g = Girl()
g.name
g.age
"""
```

```
__getattribute__
__getattribute__
"""
```

不管属性存不存在，都会经过 __getattribute__，于是我们可以实现属性拦截器。

```python
class Girl:

    def __init__(self):
        self.name = "古明地觉"
        self.age = 16
        self.gender = "female"

    def __getattr__(self, item):
        return f"不存在的属性 {item}"

    def __getattribute__(self, item):
        # 获取属性字典，注意：不能写 self.__dict__
        # 因为不管什么属性，都会先经过此方法
        # 所以写 self.__dict__ 会引发无限递归
        # 因此需要执行父类的 __getattribute__，其逻辑也很简单
        # 如果实例存在该属性，那么直接返回，否则执行 __getattr__
        # 由于 self 存在 __dict__ 属性，因此直接返回
        __dict__ = super().__getattribute__("__dict__")

        if item == "age":
            return "女人的芳龄不可泄露"
        elif item in __dict__:
            return __dict__[item]
        else:
        # 执行到这里，说明属性一定不存在了
        # 那么应该执行 __getattr__
        # 但是能够直接调用 self.__getattr__ 吗？
        # 显然是不能的，因为会引发无限递归
        # 所以应该通过父类的 __getattribute__ 去获取
        # 即：super().__getattribute__(item)，然后 return
        # 如果实例存在该属性就返回，实例不存在则会执行 __getattr__
            raise AttributeError
        # 但这里除了调用父类的 __getattribute__ 之外还有一种方式
        # 如果 raise AttributeError，那么会直接执行 __getattr__
        # 要是能够确定实例一定不存在该属性，那么这种方式也是可以的
        # 但是上面获取 __dict__ 的时候，我们没有 raise AttributeError
        # 这是因为我们需要拿到返回值，然后代码继续往下执行
        # 而这里后续已经没有逻辑了，所以直接 raise AttributeError 即可

g = Girl()
print(g.name)  # 古明地觉
print(g.gender)  # female
print(g.age)  # 女人的芳龄不可泄露

print(g.address)  # 不存在的属性 address
```

以上就是 __getattribute__，值得好好体会一下。

最后 __getattr__ 还有一个神奇的用法，就是它除了可以定义在类里面，还可以作为模块的全局变量而存在，两者的功能是类似的。我们举个栗子：

```python
# attr.py
def __getattr__(name):
    return f"不存在的属性：{name}"

name = "古明地觉"
```

看到示例代码，相信你已经猜到接下来的测试用例了。

```python
import attr
print(attr.name)  # 古明地觉
print(attr.age)  # 不存在的属性：age

from attr import gender
print(gender)  # 不存在的属性：gender
```

## 反射

__instancecheck__ 专门用于 isinstance 函数，检测一个实例对象是否属于某个类的实例。但是注意：这个方法一定要定义在元类当中，我们举个例子。

```python
class Girl:

    def __instancecheck__(self, instance):
        print("__instancecheck__被调用")
        return True


print(isinstance(123, Girl))  # False
# 上面打印了False，很正常，因为123显然不是Girl的实例
# 虽然定义了__instancecheck__、里面返回了 True，但是没用
# 因为调用的是type(Girl)的__instancecheck__

# 于是你可能想到了
print(isinstance(123, Girl()))
"""
__instancecheck__被调用
True
"""
```

我们将 Girl 改成 Girl() 不就行了吗，这样的话会调用type(Girl())、也就是Girl的__instancecheck__。确实可以，但这没有什么意义，而且事实上isinstance的第二个参数不可以是实例对象，否则报错。

```python
try:
    isinstance(123, object())
except Exception as e:
    # 告诉我们isinstance的第二个参数必须是一个类
    # 或者是一个包含多个类的元组
    print(e)
"""
isinstance() arg 2 must be a type or tuple of types
"""
# 而刚才的 isinstance(123, Girl()) 之所以没有报错
# 就是因为 Girl 的内部定义了 __instancecheck__
# 如果没有定义，那么也会报出同样的错误
```

因此__instancecheck__要定义在元类当中，尽管定义在普通的类里面也可以使用，但是没有什么意义。

```python
class MyType(type):

    def __instancecheck__(self, instance):
        # 当我们调用 isinstance(obj, cls) 的时候
        # 那么 obj 就会传递给这里的 instance 参数
        # 前提是 cls 这个类是由这里的 MyType 实例化得到的
```

```
        if hasattr(instance, "ɾ(ﾉ^ﾚ)ｧ"):
            return True
        return False

class Test(metaclass=MyType):
    pass


from fastapi import FastAPI
# FastAPI 不是 Test 的实例
print(isinstance(FastAPI, Test))  # False
# 给它设置一个属性
setattr(FastAPI, "ɾ(ﾉ^ﾚ)ｧ", "(•ㅅ•〃)")
# 就变成 Test 的实例了
print(isinstance(FastAPI, Test))  # True
```

然后是__subclasscheck__，它用于issubclass，判断一个类是不是另一个类的子类。这个方法同样需要定义在元类里面才有意义。

```
class MyType(type):

    def __subclasscheck__(self, subclass):
        # 当我们调用 issubclass(cls1, cls2) 的时候
        # 这个 cls1 就会传递给这里的 subclass 参数
        # 前提是 cls2 这个类是由这里的 MyType 实例化得到的
        if hasattr(subclass, "ɾ(ﾉ^ﾚ)ｧ"):
            return True
        return False

class Test(metaclass=MyType):
    pass


from fastapi import FastAPI
# FastAPI 不是 Test 的子类
print(issubclass(FastAPI, Test))  # False
# 给它设置一个属性
setattr(FastAPI, "ɾ(ﾉ^ﾚ)ｧ", "(•ㅅ•〃)")
# 就变成 Test 的子类
print(issubclass(FastAPI, Test))  # True
```

如果我们不把它定义在元类中，看看会怎么样？

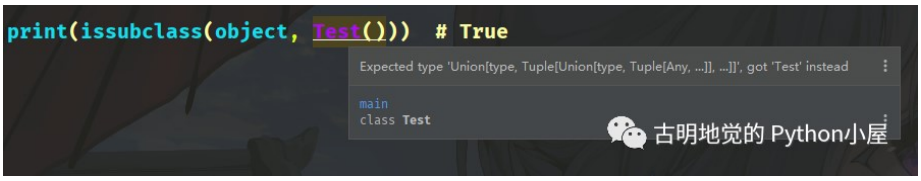
```
class Test:

    def __subclasscheck__(self, subclass):
        return True

# Test 是 object 的子类，这没有问题
# 因此 object 是所有类的基类
print(issubclass(Test, object))  # True

# 但是问题来了
print(issubclass(object, Test()))  # True
# object 居然又是 Test 实例对象的子类
# 首先 Test 的实例对象压根就不是一个类
# 它居然摇身一变，成为了Python世界中万物之父object的父类
# 究其原因就是因为 Test 内部定义了__subclasscheck__
```

所以无论是 __instancecheck__，还是 __subclasscheck__，它们都应该定义在元类里面，而不是类里面。如果定义在类里面，那么要想使这两个魔法方法生效，就必须使用该类的实例对象。

而 isinstance 和 issubclass 的第二个参数接收的都是类(或者包含多个类的元组)，我们传入实例

对象理论上是会报错的，只不过生成该实例对象的类里面定义了相应的魔法方法，所以才不会报错。

但即便如此，我们也不要这么做，因为这样没有什么意义。而且，如果你用的是PyCharm这种智能的IDE的话，也会给你标黄。




所以我们不要传入一个实例对象，也就是不要将这两个魔法方法定义在普通的类中。而是要定义在继承自 type 的类中，也就是元类。

除了上面两个魔法方法之外，还有一个与之类似的方法，叫 __subclasshook__。但它不是定义在元类中的，而是要定义在抽象基类中。可以去模块_collections_abc中看一下，里面定义了大量的抽象基类，比如Iterable、Sized、Container、Collection等等一大堆 。

随便挑一两个看一下，你就知道 __subclasshook__ 怎么用了。

## 小结

魔法方法就介绍到这，剩余的一部分就不演示了，可以自己测试一下。总的来说，Python 的花活蛮多的，特别是魔法方法，值得好好体会。


收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》79. 模块是如何导入的

下一篇 ›
《源码探秘 CPython》77. 实例对象的属性访问（下）



喜欢此内容的人还喜欢

Python第一阶段第3课————与计算机的沟通
慧心学社

python第一阶段第二课《与世界打招呼》
慧心学社

linux系统利用awk统计nginx日志
SZMSJSZ