2.5 面向对象编程

::: details INFO
译者：CIQi6

来源：2.5 Object-Oriented Programming

对应：Disc 05、HW 04、Lab 06、Ants
:::

面向对象编程（OOP）是一种组织程序的方法，它将本章介绍的许多思想结合在一起。与数据抽象中的函数一样，类创建了在使用和实现数据之间的抽象屏障。与调度字典（dispatch dictionaries）一样，对象响应行为请求。与可变数据结构一样，对象具有无法从全局环境直接访问的本地状态。Python 对象系统提供了方便的语法来促进使用这些技术来组织程序。这种语法的大部分在其他面向对象的编程语言之间共享。

对象系统提供的不仅仅是便利。它为设计程序提供了一个新的隐喻，其中几个独立的代理在计算机内交互。每个对象都以抽象两者的复杂性的方式将本地状态和行为捆绑在一起。对象相互通信，并且由于它们的交互而计算有用的结果。对象不仅传递消息，而且还在相同类型的其他对象之间共享行为，并从相关类型继承特征。

面向对象编程（OOP）的范式有自己的词汇来支持对象隐喻。我们已经看到，对象（object）是具有方法和属性的数据值，可通过点表达式（dot notation）访问。每个对象（object）也有一个类型，称为其类（class）。为了创建新类型的数据，我们实现了新类。

2.5.1 对象和类

类就像一个模板，对象是按照模板（类）生成的实例。到目前为止我们使用的对象都有内置类，但也可以创建新的用户定义类。类定义指定在该类的对象之间共享的属性和方法。我们将通过重新访问银行账户的例子来介绍类语句。

在引入本地状态时，我们看到银行账户要具有 `balance` 的可变值。银行帐户对象应具有 `withdraw` 方法，用于更新帐户余额并返回请求的金额（如果可用）。要完成抽象：一个银行账户应该能够返回其当前的 `balance`，返回账户 `holder` 的名称，以及 `deposit` 的金额。

`Account` 类允许我们创建多个银行账户实例。创建新对象实例的操作称为实例化类。Python 中用于实例化类的语法与调用函数的语法相同。在这种情况下，我们用参数 `Kirk` 调用 `Account`，即帐户持有人的姓名。

```
>>> a = Account('Kirk')
```

对象的属性是与对象关联的名称 - 值对，可通过点表达式访问。对于特定对象，其有特定值的属性，（而不是类的所有对象）称为实例属性。每个 `Account` 都有自己的余额和账户持有人姓名，这是实例属性的示例。在更广泛的编程社区中，实例属性也可以称为字段、属性或实例变量。

```
>>> a.holder
'Kirk'
>>> a.balance
0
```

对对象进行操作或执行特定于对象的计算的函数称为方法。方法的返回值和副作用可以依赖于并更改对象的其他属性。例如， `deposit` 是我们 `Account` 对象 `a` 的方法。它需要一个参数，即要存入的金额，更改对象的 `balance` 属性，并返回结果余额。

```
>>> a.deposit(15)
15
```

我们说方法是在特定对象上调用的。调用 `withdraw` 方法的结果是，要么批准提款并扣除金额，要么拒绝请求并返回错误消息。

```
>>> a.withdraw(10)  # withdraw 方法返回扣除后的金额
5
>>> a.balance       # 金额属性发生改变
5
>>> a.withdraw(10)
'Insufficient funds'
```

如上所示，方法的行为可能取决于对象不断变化的属性，方法也可以改变对象的属性。具有相同参数的两次对 `withdraw` 的调用将返回不同的结果。

2.5.2 类的定义

`class` 语句可以创建自定义类，类体里面又包含多条子语句。类语句定义类名，类体包含一组语句来定义类的属性。

```
class <name>:
    <suite>
```

执行类语句，将创建一个新类，并在当前环境的第一帧中绑定 `<name>` 。然后执行类体里面的语句。在 `class` 的 `<suite>` 中 `def` 或赋值语句中绑定的任何名称都会创建或修改类的属性。

类通常通过操作类属性来进行设计，这些属性是与该类的每个实例关联的名称 - 值对。类通过定义一个初始化对象的方法来指定特定对象的实例属性。例如，初始化 `Account` 类的对象的一部分是为它分配一个 0 的起始余额。

`class` 语句中的 `<suite>` 包含 `def` 语句， `def` 语句为类的对象定义新方法。初始化对象的方法在 `Python` 中有一个特殊的名称 `__init__` (“init”的每一侧都有两个下划线)，称为类的构造函数（constructor）。

```
>>> class Account:
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
```

`Account` 的 `__init__` 方法有两个形式参数。第一个 `self` 绑定到新创建的 `Account` 对象。第二个参数 `account_holder` 绑定到调用类进行实例化时传递给类的参数。

构造函数将实例属性名称 `balance` 绑定到 0。它还将属性名称 `holder` 绑定到名称 `account_holder` 的值。形式参数 `account_holder` 是 `__init__` 方法中的本地名称。另一方面，通过最终赋值语句绑定的名称 `holder` 仍然存在，因为它使用点表达式存储为 `self` 的属性。

定义 `Account` 类后，我们可以实例化它。

```
>>> a = Account('Kirk')
```

上面的语句调用 `Account` 类创建一个新对象，这个对象是 `Account` 的一个实例，然后使用两个参数调用构造函数 `__init__`：新创建的对象和字符串“Kirk”。一般来说，我们使用参数名称 `self` 作为构造函数的第一个参数，它会自动绑定到正在实例化的对象。几乎所有的 Python 代码都遵守这个规定。

现在，我们可以使用符号点来访问对象的 `balance` 和 `holder`。

```
>>> a.balance
0
>>> a.holder
'Kirk'
```

身份标识：每一个账号实例都有自己的余额属性，它的值是独立的。

```
>>> b = Account('Spock')
>>> b.balance = 200
>>> [acc.balance for acc in (a, b)]
[0, 200]
```

为了强调这种独立性，每一个实例对象都具有唯一的身份标识。使用 `is` 和 `is not` 运算符可以比较对象的标识。

```
>>> a is a
True
>>> a is not b
True
```

尽管是从相同的调用构造的，但绑定到 `a` 和 `b` 的对象并不相同。像前面的一样，使用赋值将对象绑定到新名称不会创建新对象。

```
>>> c = a
>>> c is a
True
```

仅当使用调用表达式语法实例化类（如 `Account`）时，才会创建具有用户定义类的新对象。

方法：对象方法也由 `class` 语句内的 `def` 语句定义。下面， `deposit` 和 `withdraw` 都定义为 `Account` 类对象上的方法。

```
>>> class Account:
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
            if amount > self.balance:
                return 'Insufficient funds'
            self.balance = self.balance - amount
            return self.balance
```

虽然方法定义在声明方式上与函数定义没有区别，但方法定义在执行时确实具有不同的效果。由 `class` 语句中的 `def` 语句创建的函数值绑定到声明的名称，作为属性在类中本地绑定。该值可以使用类实例中的点表达式的方法调用。

每个方法都包含着一个特殊的首参 `self` ，该参数绑定调用该方法的对象。例如，假设在特定的 `Account` 对象上调用 `deposit` 并传递单个参数：存入的金额。对象本身就被绑定到 `self` ，而传入的参数绑定到 `amount` 。所有调用的方法都可以通过 `self` 参数来访问对象，因此它们都可以访问和操作对象的状态。

为了调用这些方法，我们再次使用点表达式，如下图所示。

```
>>> spock_account = Account('Spock')
>>> spock_account.deposit(100)
100
>>> spock_account.withdraw(90)
10
>>> spock_account.withdraw(90)
'Insufficient funds'
>>> spock_account.holder
'Spock'
```

当通过点表达式调用方法时，对象本身（在本例中绑定为 `spock_account` ）扮演双重角色。首先，它确定名称 `withdraw` 的含义; `withdraw` 不是环境中的名称，而是 `Account` 类的本地名称。其次，当调用 `withdraw` 方法时，它绑定到第一个参数 `self` 。

2.5.3 消息传递和点表达式

在类中定义的方法和在构造函数中分配的实例属性是面向对象编程的基本元素。这两个概念在传递数据值的消息实现中复制了调度字典的大部分行为。对象使用点表达式获取消息，但这些消息不是任意字符串值键，而是类的本地名称。对象还具有命名的本地状态值（实例属性），但可以使用点表达式访问和操作该状态，而无需在实现中使用 `nonlocal` 语句。

消息传递的主要思想是，数据值应该通过响应与其表示的抽象类型相关的消息来具有行为。点表示式是 Python 的一个语法特征，它形式化了消息传递隐喻。将语言与内置对象系统一起使用的优点是，消息传递可以与其他语言功能（如赋值语句）无缝交互。我们不需要不同的消息来“获取”或“设置”与本地属性名称关联的值; 语言语法允许我们直接使用消息名称。

点表达式：代码片段 `spock_account.deposit` 称为点表达式。点表达式由表达式、点和名称组成：

```
<expression>.<name>
```

`<expression>` 可以是任何有效的 Python 表达式，但 `<name>` 必须是简单名称（而不是计算结果为名称的表达式）。点表达式的计算结果为作为 `<expression>` 值的对象的 `<name>` 的属性值。

内置函数 `getattr` 也可以按名称返回对象的属性。它是点表示法的函数等效物。使用 `getattr` ，我们可以使用字符串查找属性，就像我们对调度字典所做的那样。

```
>>> getattr(spock_account, 'balance')
10
```

我们还可以使用 `hasattr` 来测试对象是否具有指定的属性。

```
>>> hasattr(spock_account, 'deposit')
True
```

对象的属性包括其所有实例属性，以及其类中定义的所有属性（包括方法）。方法是需要特殊处理的类的属性。

方法和函数：在对象上调用方法时，该对象将作为第一个参数隐式传递给该方法。也就是说，点左侧的 `<expression>` 值的对象将自动作为第一个参数传递给点表达式右侧命名的方法。因此，对象绑定到参数 `self`。

为了实现自动 `self` 绑定，Python 区分了我们从文本开头就一直在创建的函数和绑定方法，它们将函数和将调用该方法的对象耦合在一起。绑定方法值已与其第一个参数（调用它的实例）相关联，在调用该方法时将命名为 `self`。

我们可以通过对点表达式的返回值调用 `type` 来查看交互式解释器的差异。作为类的属性，方法只是一个函数，但作为实例的属性，它是一个绑定方法：

```
>>> type(Account.deposit)
<class 'Function'>
>>> type(spock_account.deposit)
<class 'method'>
```

这两个结果的区别仅在于第一个是参数为 `self` 和 `amount` 的标准双参数函数。第二种是单参数方法，调用方法时，名称 `self` 将自动绑定到名为 `spock_account` 的对象，而参数 `amount` 将绑定到传递给方法的参数。这两个值（无论是函数值还是绑定方法值）都与相同的 `deposit` 函数体相关联。

我们可以通过两种方式调用 `deposit`：作为函数和作为绑定方法。在前一种情况下，我们必须显式地为 `self` 参数提供一个参数。在后一种情况下， `self` 参数会自动绑定。

```
>>> Account.deposit(spock_account, 1001)     # 函数 deposit 接受两个参数
1011
>>> spock_account.deposit(1000)              # 方法 deposit 接受一个参数
2011
```

函数 `getattr` 的行为与点表示法完全相同：如果它的第一个参数是一个对象，但名称是类中定义的方法，则 `getattr` 返回一个绑定方法值。另一方面，如果第一个参数是一个类，则 `getattr` 直接返回属性值，这是一个普通函数。

命名约定：类名通常使用 CapWords 约定（也称为 CamelCase，因为名称中间的大写字母看起来像驼峰）编写。方法名称遵循使用下划线分隔的小写单词命名函数的标准约定。

在某些情况下，有一些实例变量和方法与对象的维护和一致性相关，我们不希望对象的用户看到或使用。它们不是类定义的抽象的一部分，而是实现的一部分。Python 的约定规定，如果属性名称以下划线开头，则只能在类本身的方法中访问它，而不是用户访问。

2.5.4 类属性

某些属性值在给定类的所有对象之间共享。此类属性与类本身相关联，而不是与类的任何单个实例相关联。例如，假设银行以固定利率支付账户余额的利息。该利率可能会发生变化，但它是所有账户共享的单一价值。

类属性由 `class` 语句套件中的赋值语句创建，位于任何方法定义之外。在更广泛的开发人员社区中，类属性也可以称为类变量或静态变量。以下类语句为 `Account` 创建名称为 `interest` 的类属性。

```
>>> class Account:
        interest = 0.02            # 类属性
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        # 在这里定义更多的方法
```

仍然可以从类的任何实例访问此属性。

```
>>> spock_account = Account('Spock')
>>> kirk_account = Account('Kirk')
>>> spock_account.interest
0.02
>>> kirk_account.interest
0.02
```

但是，类属性的赋值会改变类的所有实例的属性值。

```
>>> Account.interest = 0.04
>>> spock_account.interest
0.04
>>> kirk_account.interest
0.04
```

属性名称：我们已经在对象系统中引入了足够的复杂性，以至于我们必须指定如何将名称解析为特定属性。毕竟，我们可以很容易地拥有一个同名的类属性和一个实例属性。

正如我们所看到的，点表达式由表达式、点和名称组成：

```
<expression> . <name>
```

计算点表达式：

1. 点表达式左侧的 `<expression>`，生成点表达式的对象。
2. `<name>` 与该对象的实例属性匹配；如果存在具有该名称的属性，则返回属性值。
3. 如果实例属性中没有 `<name>`，则在类中查找 `<name>`，生成类属性。
4. 除非它是函数，否则返回属性值。如果是函数，则返回该名称绑定的方法。

在这个过程中，实例属性在类属性之前，就像本地名称在环境中优先于全局名称一样。在类中定义的方法与点表达式的对象相结合，以在此计算过程的第四步中形成绑定方法。在类中查找名称的过程具有其他细微差别，一旦我们引入类继承，很快就会出现这些细微差别。

属性赋值：所有左侧包含点表达式的赋值语句都会影响该点表达式对象的属性。如果对象是实例，则赋值将设置实例属性。如果对象是类，则赋值将设置类属性。由于此规则，对对象的属性的赋值不会影响其类的属性。下面的示例说明了这种区别。

如果我们分配给帐户实例的命名属性 `interest`，我们将创建一个与现有类属性同名的新实例属性。

```
>>> kirk_account.interest = 0.08
```

并且该属性值将从点表达式返回。

```
>>> kirk_account.interest
0.08
```

但是，class 属性的 `interest` 仍保留其初始值，该值将针对其他账号（实例）返回。

```
>>> spock_account.interest
0.04
```

对类属性 `interest` 的更改将影响到 `spock_account` ，但 `kirk_account` 的实例属性将不受影响。

```
>>> Account.interest = 0.05      # 改变类属性
>>> spock_account.interest       # 实例属性发生变化（该实例中没有和类属性同名称的实例属性）
0.05
>>> kirk_account.interest        # 如果实例中存在和类属性同名的实例属性，则改变类属性，不会影响实例属性
0.08
```

2.5.5 继承

在面向对象编程范式中，我们经常会发现不同类型之间存在关联，尤其是在类的专业化程度上。即使两个类具有相似的属性，它们的特殊性也可能不同。

例如，我们可能需要实现一个支票账户，与标准账户不同，支票账户每次取款需额外收取 1 美元手续费，并且利率较低。下面我们展示了期望的行为。

```
>>> ch = CheckingAccount('Spock')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # withdrawals decrease balance by an extra charge
14
```

`CheckingAccount` 是 `Account` 的特化。在 OOP 术语中，通用帐户将用作 `CheckingAccount` 的基类，而 `CheckingAccount` 将用作 `Account` 的子类。术语基类（base class）也常叫父类（parent class）和超类（superclass），而子类（subclass）也叫孩子类（child class）。

子类继承其父类的属性，但可以重写某些属性，包括某些方法。对于继承，我们只指定子类和父类之间的区别。我们在子类中未指定的任何内容都会被自动假定为与父类的行为一样。

继承在我们的对象隐喻中也起着作用，除了是一个有用的组织特征。继承旨在表示类之间的 is-a 关系，这与 has-a 关系形成对比。活期账户是一种特定类型的账户，因此从 `Account` 继承 `CheckingAccount` 是继承的适当使用。另一方面，银行有它管理的银行账户清单，所以任何一方都不应该从另一方继承。相反，帐户对象列表自然地表示为银行对象的实例属性。

2.5.6 使用继承

首先，我们给出了 `Account` 类的完整实现，其中包括该类及其方法的文档字符串。

```
>>> class Account:
        """一个余额非零的账户。"""
        interest = 0.02
```

```
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
            """存入账户 amount，并返回变化后的余额"""
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
            """从账号中取出 amount，并返回变化后的余额"""
            if amount > self.balance:
                return 'Insufficient funds'
            self.balance = self.balance - amount
            return self.balance
```

下面显示了 `CheckingAccount` 的完整实现。我们通过将计算结果为基类的表达式放在类名后面的括号中来指定继承。

```
>>> class CheckingAccount(Account):
          """从账号取钱会扣出手续费的账号"""
          withdraw_charge = 1
          interest = 0.01
          def withdraw(self, amount):
                return Account.withdraw(self, amount + self.withdraw_charge)
```

在这里，我们介绍一个 `CheckingAccount` 类的类属性 `withdraw_charge` 。我们为 `CheckingAccount` 的 `interest` 属性分配一个较低的值。我们还定义了一个新的 `withdraw` 方法来覆盖 `Account` 类中定义的行为。由于类套件中没有其他语句，所有其他行为都继承自基类 `Account` 。

```
>>> checking = CheckingAccount('Sam')
>>> checking.deposit(10)
10
>>> checking.withdraw(5)
4
>>> checking.interest
0.01
```

表达式 `checking.deposit` 的计算结果是用于存款的绑定方法，该方法在 `Account` 类中定义。当 Python 解析点表达式中不是实例属性的名称时，它会在类中查找该名称。事实上，在类中“查找”名称的行为试图在原始对象的类的继承链中的每个父类中找到该名称。我们可以递归地定义此过程。

在类中查找名称。

1. 如果它命名在指定类中的属性，则返回属性值。
2. 否则，在该类的父类中查找该名称的属性。

在 `deposit` 的情况下，Python 将首先在实例上查找名称，然后在 `CheckingAccount` 类中查找名称。最后，它将在定义了 `deposit` 的 `Account` 类中查找。根据我们对点表达式的计算规则，由于 `deposit` 是在类中查找的 `checking` 实例的函数，因此点表达式的计算结果为绑定方法值。该方法使用参数 10 调用，该参数调用 deposit 方法，其中 `self` 绑定到 `checking` 对象， `amount` 绑定到 10。

对象的类始终保持不变。尽管在 `Account` 类中找到了 `deposit` 方法，但调用 `deposit` 时， `3` 绑定到 `CheckingAccount` 的实例，而不是 `Account` 的实例。

调用父类。重写的属性可以通过类对象来访问。例如，我们通过调用 `CheckingAccount` 中包含 `withdraw_charge` 参数的方法 `withdraw` 。该方法的实现是通过调用 `Account` 中的 `withdraw` 方法来实现的。

请注意，我们调用了 `self.withdraw_charge` 而不是等效的 `CheckingAccount.withdraw_charge` 。前者相对于后者的好处是，从 `CheckingAccount` 继承的类可能会覆盖 `withdraw_charge` 。如果是这种情况，我们希望我们的实现的 `withdraw` 找到新值而不是旧值。

接口。在面向对象的程序中，不同类型的对象将共享相同的属性名称是极其常见的。对象接口是这些属性的属性和条件的集合。例如，所有帐户都必须具有采用数值参数的 `deposit` 和 `withdraw` 方法，以及 `balance` 属性。类 `Account` 和类 `CheckingAccount` 都实现此接口。继承（Inheritance）专门以这种方式促进名称共享。在某些编程语言（如 Java）中，必须显式声明接口实现。在其他对象（如 Python、Ruby 和 Go）中，任何具有适当名称的对象都实现了接口。

在使用对象（不是实现对象）的时候，我们只假设它们的属性，而不假设对象类型，则对将来的更改最可靠。例如耳朵，我们不先考虑它是猫的耳朵，还是狗的耳朵，我们只知道它是耳朵。我们只说耳朵是有形状的，它可以听见某种声音然后产生反馈的。等到以后，我们需要它是狗的耳朵时，我们在具体说它是漏斗状的，它的听力范围大概在 24 米左右。也就是说，他们使用对象抽象，而不是对其实现进行任何假设。

例如，假设我们运行彩票，我们希望将 5 美元存入每个帐户列表。以下实现不假定有关这些帐户类型的任何内容，因此同样适用于具有 `deposit` 方法的任何类型的对象：

```
>>> def deposit_all(winners, amount=5):
        for account in winners:
            account.deposit(amount)          # 这里调用的是实例 account 的 deposit 方法
            # 对于不同实例来说，它们的 deposit 方法可能不同。这个例子相对于下面来讲，更加具有健壮性
```

上面的函数 `deposit_all` 仅假设每个 `account` 满足帐户对象抽象，因此它将与也实现此接口的任何其他帐户类一起使用。假设特定的帐户类将违反帐户对象抽象的抽象屏障。例如，以下实现不一定适用于新类型的帐户：

```
>>> def deposit_all(winners, amount=5):
        for account in winners:
            Account.deposit(account, amount)    # 这里调用的是类 Account 中的 deposit 方法
```

我们将在本章后面更详细地讨论这个主题。

2.5.7 多继承

Python 支持子类从多个基类继承属性的概念，这种语言功能称为多重继承（multiple inheritance）。

假设我们有一个从 `Account` 继承的 `SavingsAccount` ，但每次客户存款时都会向他们收取少量费用。

```
>>> class SavingsAccount(Account):
        deposit_charge = 2
        def deposit(self, amount):
            return Account.deposit(self, amount - self.deposit_charge)
```

然后，一位聪明的高管设想了一个具有 `CheckingAccount` 和 `SavingsAccount` 最佳功能的 `AsSeenOnTVAccount` 账户：提款费、存款费和低利率。它既是活期账户，又是储蓄账户！“如果我们建造它，”这位高管解释说，“有人会注册并支付所有这些费用。我们甚至会给他们一美元。

```
>>> class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
        def __init__(self, account_holder):
            self.holder = account_holder
            self.balance = 1           # 赠送的 1 $!
```

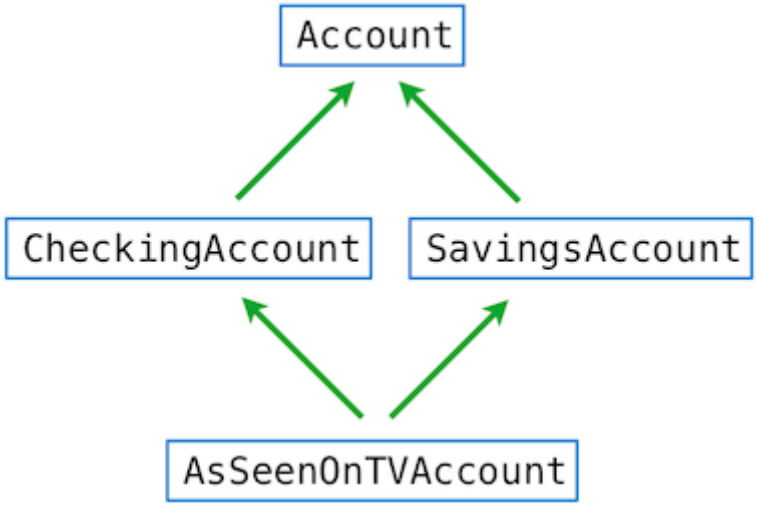
事实上，上面这段简短的代码已经实现了我们想要的功能了。取款和存款都将产生费用，分别使用 `CheckingAccount` 和 `SavingsAccount` 中的函数定义。

```
>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)             # 调用 SavingsAccount 的 deposit 方法，会产生
2 $的存储费用
19
>>> such_a_deal.withdraw(5)             # 调用 CheckingAccount 的 withdraw 方法，产生
1 $的取款费用。
13
```

如果没有非歧义引用，就按预期正确解析：

```
>>> such_a_deal.deposit_charge
2
>>> such_a_deal.withdraw_charge
1
```

但是，当引用不明确时，例如对 `Account` 和 `CheckingAccount` 中定义的 `withdraw` 方法的引用，该怎么办？下图描述了 `AsSeenOnTVAccount` 个类的继承图。每个箭头都指向从子类到基类。



对于像这样的简单“菱形”形状，Python 会从左到右解析名称，然后向上解析名称。在此示例中，Python 按顺序检查以下类中的属性名称，直到找到具有该名称的属性：

```
AsSeenOnTVAccount, CheckingAccount, SavingsAccount, Account, object
```

继承排序问题没有正确的解决方案，因为在某些情况下，我们可能更愿意将某些继承类置于其他类之上。但是，任何支持多重继承的编程语言都必须以一致的方式选择某些排序，以便该语言的用户可以预测其程序的行为。

进一步阅读。Python 使用称为 C3 方法解析排序的递归算法解析此名称。可以在所有类上使用 `mro` 方法查询任何类的方法解析顺序。

```
>>> [c.__name__ for c in AsSeenOnTVAccount.mro()]
['AsSeenOnTVAccount', 'CheckingAccount', 'SavingsAccount', 'Account', 'object']
```

找到方法解析顺序的具体算法不是本文的主题，但 Python 的主要作者已经提供了描述该算法的参考文献。

2.5.8 对象的作用

Python 的对象系统旨在同时方便和灵活地实现数据抽象和消息传递。类、方法、继承和点表达式的特殊语法都使我们能够在程序中形式化对象的概念，从而提高我们组织大型程序的能力。换句话说，Python 的对象系统提供了一种方便而灵活的方法来创建和操作对象，使程序员能够更好地组织和管理复杂的程序。

特别是，我们希望我们的对象系统能够促进程序不同方面之间的关注点分离。程序中的每个对象封装和管理程序状态的某些部分，每个类语句定义实现程序整体逻辑的某些部分的函数。抽象障碍强制实施大型程序不同方面之间的边界。

面向对象编程非常适合用于模拟由独立但相互作用部分构成的系统。例如，不同用户在社交网络中进行交互，不同角色在游戏中进行交互，不同形状在物理模拟中进行交互。在表示这样的系统时，程序中的对象通常可以自然地映射到被建模系统中的对象，而类则代表它们的类型和关系。

另一方面，类可能不是实现某些抽象的最佳机制。函数式抽象提供了一个更自然的隐喻来表示输入和输出之间的关系。我们不应该觉得必须将程序中的每一点逻辑都塞进一个类中，尤其是在定义独立函数来操作数据更自然的情况下。函数还可以强制实现关注点的分离。换句话说，函数式编程提供了另一种有效地组织程序逻辑的方法，使得程序员能够更好地处理和维护程序。在某些情况下，使用函数式编程方法可能比使用面向对象编程更自然和有效。

多范式语言，如 Python，允许程序员将组织范式与适当的问题相匹配。学会识别何时引入新类，而不是新函数，以简化或模块化程序，是软件工程中一项重要的设计技能，值得认真关注。