Chapter 1

类与对象

1.1 修改实例的字符串表示

特殊方法`__repr__()`返回的是实例的代码表示（code representation），通常可以用它返回的字符串文本来重新创建这个实例。内建的 repr()函数可以用来返回这个字符串，当缺少交互式解释环境时可用它来检查实例的值。特殊方法`__str__()`将实例转换为一个字符串，这也是由 str()和 print()函数所产生的输出。

对于`__repr__()`，标准的做法是让它产生的字符串文本能够满足 `eval(repr(x)) == x`。如果不可能办到或者说不希望有这种行为，那么通常就让它产生一段有帮助意义的文本，并且以<和>括起来。

1.2 自定义字符串的输出格式

`__format__()`方法在 Python 的字符串格式化功能中提供了一个钩子。

1.3 让对象支持上下文管理协议

要让对象能够兼容 with 语句，需要实现`__enter__()`和`__exit__()`方法。

要编写一个上下文管理器，其背后的主要原则就是我们编写的代码需要包含在由 with 语句定义的代码块中。当遇到 with 语句时，`__enter__()`方法首先被触发执行。`__enter__()`的返回值（如果有的话）被放置在由 as 限定的变量当中。之后开始执行 with 代码块中的语句。最后，`__exit__()`方法被触发来执行清理工作。

这种形式的控制流与 with 语句块中发生了什么情况是没有关联的，出现异常时也是如此。`__exit__()`方法可以选择以某种方式来使用异常信息，或者什么也不干直接忽略它并返回 None 作为结果。如果`__exit__()`返回 True，异常就会被清理干净，好像什么都没发生过一样，而程序也会立刻继续执行 with 语句块之后的代码。

1.4 当创建大量实例时如何节省内存

当定义了`__slots__`属性时，Python 就会针对实例采用一种更加紧凑的内部表示。不再让每个实例都创建一个`__dict__`字典，现在的实例是围绕着一个固定长度的小型数组来构建的，这和一

个元组或者列表很相似。在__slots__中列出的属性名会在内部映射到这个数组的特定索引上。使用__slots__带来的副作用是我们没法再对实例添加任何新的属性了——我们被限制为只允许使用__slots__中列出的那些属性名。

尽管__slots__看起来似乎是一个非常有用的特性，但是在大部分代码中都应该尽量别使用它。Python 中有许多部分都依赖于传统的基于字典的实现。此外，定义了__slots__属性的类不支持某些特定的功能，比如多重继承。就大部分情况而言，我们应该只针对那些在程序中被当做数据结构而频繁使用的类上采用__slots__技法（例如，如果你的程序创建了上百万个特定的类实例）。

关于__slots__有一个常见的误解，那就是这是一种封装工具，可以阻止用户为实例添加新的属性。尽管这的确是使用__slots__所带来的副作用，但这绝不是使用__slots__的原本意图。相反，人们一直以来都把__slots__当做一种优化工具。

1.5 将名称封装到类中

第一个规则是任何以单下划线（_）开头的名字应该总是被认为只属于内部实现。Python 本身并不会阻止其他人访问内部名称。但是如果有人这么做了，则被认为是粗鲁的，而且可能导致产生出脆弱不堪的代码。

以双下划线打头的名称会导致出现名称重整（name mangling）的行为。此时你可能会问，类似这样的名称重整其目的何在？答案就是为了继承——这样的属性不能通过继承而覆盖。

对于大部分代码而言，我们应该让非公有名称以单下划线开头。但是，如果我们知道代码中会涉及子类化处理，而且有些内部属性应该对子类进行隐藏，那么此时就应该使用双下划线开头。

此外还应该指出的是，有时候可能想定义一个变量，但是名称可能会和保留字产生冲突。基于此，应该在名称最后加上一个单下划线以示区别。

1.6 创建可管理的属性

property 的重要特性就是它看起来就像一个普通的属性，但是根据访问它的不同方式，会自动触发 getter、setter 以及 deleter 方法。

property 属性实际上就是把一系列的方法绑定到一起。如果检查类的 property 属性，就会发现 property 自身所持有的属性 fget、fset 和 fdel 所代表的原始方法。一般来说我们不会直接去调用 fget 或者 fset，但是当我们访问 property 属性时会自动触发对这些方法的调用。

只有当确实需要在访问属性时完成一些额外的处理任务时，才应该使用 property。

property 也可以用来定义需要计算的属性。这类属性并不会实际保存起来，而是根据需要完成计算。（**读者注：似乎有点类似于 SQL 中的视图，只是定义了一种计算或者操作**）

1.7 调用父类中的方法

要调用父类（或称超类）中的方法，可以使用 super()函数完成。super()函数的一种常见用途是调用父类的__init__()方法，确保父类被正确地初始化了。另一种常见用途是当覆盖了 Python 中的特殊方法时.。

针对每一个定义的类，Python 都会计算出一个称为方法解析顺序（MRO）的元组。MRO 元组只是简单地对所有的基类进行线性排列。要实现继承，Python 从 MRO 列表中最左边的类开始，从左到右依次查找，直到找到待查的属性时为止。

而 MRO 列表本身又是如何确定的呢？这里用到了一种称为 C3 线性化处理（C3 Linearization）的技术。为了不陷入到艰深的数学理论中，简单来说这就是针对父类的一种归并排序，它需要满足 3 个约束：

- 先检查子类再检查父类；
- 有多个父类时，按照 MRO 列表的顺序依次检查；
- 如果下一个待选的类出现了两个合法的选择，那么就从第一个父类中选取。

老实说，所有需要的知道的就是 MRO 列表中对类的排序几乎适用于任何定义的类层次结构（class hierarchy）。

当使用 super()函数时，Python 会继续从 MRO 中的下一个类开始搜索。只要每一个重新定义过的方法（也就是覆盖方法）都使用了 super()，并且只调用了它一次（**读者注：第三条约束吧**），那么控制流最终就可以遍历整个 MRO 列表，并且让每个方法只会被调用一次。

由于 super()可能会调用到我们不希望调用的方法，那么这里有一些应该遵守的基本准则。首先，确保在继承体系中所有同名的方法都有可兼容的调用签名（即，参数数量相同，参数名称也相同）。如果 super()尝试去调用非直接父类的方法，那么这就可以确保不会遇到麻烦。其次，确保最顶层的类实现了这个方法通常是个好主意。这样沿着 MRO 列表展开的查询链会因为最终找到了实际的方法而终止。

1.8 在子类中扩展属性

1.9 创建一种新形式的类属性或实例属性

如果想创建一个新形式的实例属性，可以以描述符类的形式定义其功能。所谓的描述符就是以特殊方法`__get__()`、`__set__()`和`__delete__()`的形式实现了三个核心的属性访问操作（对应于 get、set 和 delete）的类。这些方法通过接受类实例作为输入来工作。之后，底层的实例字典会根据需要适当地进行调整。当这么做时，所有针对描述符属性（即，这里的 x 或 y）的访问都会被`__get__()`、`__set__()`和`__delete__()`方法所捕获。每个描述符方法都会接受被操纵的实例作为输入。要执行所请求的操作，底层的实例字典（即`__dict__`属性）会根据需要适当地进行调整。描述符的 self.name 属性会保存字典的键，通过这些键可以找到存储在实例字典中的实际数据。

应该强调的是，如果只是想访问某个特定的类中的一种属性，并对此做定制化处理，那么最好不要编写描述符来实现。对于这个任务，用 property 属性方法来完成会更加简单。在需要大量重用代码的情况下，描述符会更加有用（例如，我们希望在自己的代码中大量使用描述符提供的功能，或者将其作为库来使用）。