```
{'b': 2}

>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []):
        print(attr, '=>', getattr(X, attr))

b => 2                                          # Other superclass slots missed!
a => 1
__dict__ => {'b': 2}

>>> dir(X)                                      # But dir() includes all slot names
[...many names omitted... 'a', 'b', 'c', 'd']
```

换句话说，就通用地列举实例属性而言，一个 `__slots__` 总是不够的——实例的完整属性可能受制于完整继承搜索的过程。参阅前面的 *mapattrs-slots.py* 文件来看看另一个 `slot` 出现在多重父类中的例子。如果一棵类树中的多个类都存在自己的 `__slots__` 属性，那么通用的程序必须为列举属性建立其他的策略，这在下一小节中会给出解释。

## 一般性地处理 slot 和其他“虚拟”属性

目前为止，你可能希望回顾上一章末尾的 *lister.py* 显示 mixin 类示例中对 `slot` 策略选项的讨论。那是一个为什么通用程序可能需要关注 `slot` 的重要例子。在这样一个试图通用地列出属性的工具中必须考虑到 `slot`，以及其他像这样的潜在“虚拟”实例属性，例如后面会讨论到的 property 和描述符。这些名称也相似地出现在类中但可以按需为实例提供属性值。`slot` 是它们之中最以数据为核心的，不过也代表着一个更大的类别。

这样的属性需要包容性的方式、特殊的处理或是通用的回避，后者在任何程序员向主体代码中引入 `slot` 的时候就失效了。事实上，像 `slot` 这样类一级的实例属性可能需要我们重新定义“实例数据”一词——这个新的定义包括局部存储属性、所有被继承属性的并集、或是这个并集的一个子集。

例如，一些程序可能将 `slot` 名称归为类的属性而非实例的属性；毕竟，这些属性并不存在于实例的命名空间字典中。另一方面，正如前面所述，程序可以通过依赖 `dir` 来获取所有被继承的属性名称、或是用 `getattr` 来获取实例中它们对应的值，从而变得更加有包容性，并因此无需考虑它们物理上的位置或是实现。如果你必须支持 `slot` 作为实例数据，这种方式是可能是最健壮的：

```
>>> class Slotful:
        __slots__ = ['a', 'b', '__dict__']
        def __init__(self, data):
            self.c = data

>>> I = Slotful(3)
>>> I.a, I.b = 1, 2
>>> I.a, I.b, I.c                               # Normal attribute fetch
(1, 2, 3)

>>> I.__dict__                                  # Both __dict__ and slots storage
```