8.11 简化数据结构的初始化¶

问题¶

你写了很多仅仅用作数据结构的类，不想写太多烦人的 `__init__()` 函数

解决方案¶

可以在一个基类中写一个公用的 `__init__()` 函数：

```
import math

class Structure1:
    # Class variable that specifies expected fields
    _fields = []

    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))
        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)
```

然后使你的类继承自这个基类:

```
# Example class definitions
class Stock(Structure1):
    _fields = ['name', 'shares', 'price']

class Point(Structure1):
    _fields = ['x', 'y']

class Circle(Structure1):
    _fields = ['radius']

    def area(self):
        return math.pi * self.radius ** 2
```

使用这些类的示例：

```
>>> s = Stock('ACME', 50, 91.1)
>>> p = Point(2, 3)
>>> c = Circle(4.5)
>>> s2 = Stock('ACME', 50)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "structure.py", line 6, in __init__
        raise TypeError('Expected {} arguments'.format(len(self._fields)))
TypeError: Expected 3 arguments
```

如果还想支持关键字参数，可以将关键字参数设置为实例属性：

```
class Structure2:
    _fields = []

    def __init__(self, *args, **kwargs):
        if len(args) > len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set all of the positional arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the remaining keyword arguments
        for name in self._fields[len(args):]:
            setattr(self, name, kwargs.pop(name))
```

```
        # Check for any remaining unknown arguments
        if kwargs:
            raise TypeError('Invalid argument(s): {}'.format(','.join(kwargs)))
# Example use
if __name__ == '__main__':
    class Stock(Structure2):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, price=91.1)
    s3 = Stock('ACME', shares=50, price=91.1)
    # s3 = Stock('ACME', shares=50, price=91.1, aa=1)
```

你还能将不在 `_fields` 中的名称加入到属性中去：

```
class Structure3:
    # Class variable that specifies expected fields
    _fields = []

    def __init__(self, *args, **kwargs):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the additional arguments (if any)
        extra_args = kwargs.keys() - self._fields
        for name in extra_args:
            setattr(self, name, kwargs.pop(name))

        if kwargs:
            raise TypeError('Duplicate values for {}'.format(','.join(kwargs)))

# Example use
if __name__ == '__main__':
    class Stock(Structure3):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, 91.1, date='8/2/2012')
```

讨论¶

当你需要使用大量很小的数据结构类的时候， 相比手工一个个定义 `__init__()` 方法而已，使用这种方式可以大大简化代码。

在上面的实现中我们使用了 `setattr()` 函数类设置属性值， 你可能不想用这种方式，而是想直接更新实例字典，就像下面这样：

```
class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments (alternate)
        self.__dict__.update(zip(self._fields,args))
```

尽管这也可以正常工作，但是当定义子类的时候问题就来了。 当一个子类定义了 `__slots__` 或者通过property(或描述器)来包装某个属性， 那么直接访问实例字典就不起作用了。我们上面使用 `setattr()` 会显得更通用些，因为它也适用于子类情况。

这种方法唯一不好的地方就是对某些IDE而言，在显示帮助函数时可能不太友好。比如：

```
>>> help(Stock)
Help on class Stock in module __main__:
class Stock(Structure)
...
| Methods inherited from Structure:
|
| __init__(self, *args, **kwargs)
|
...
>>>
```

可以参考9.16小节来强制在 `__init__()` 方法中指定参数的类型签名。