Lesson 12 - Interfaces and protocols

July 9, 2020

1 Agenda

- Private and “protected” attributes
- Interfaces
- Protocols

2 Private and “protected” attributes

[7]:
```
# %load vehicle.py
class Vehicle:

    __id_type = 'vin'


class Car(Vehicle):

    __id_type = 'license_plate'


new_car = Car()
print(new_car._Vehicle__id_type)
print(new_car._Car__id_type)

print(new_car.__id_type)
```

```
vin
license_plate


       ␣
 ↪---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call␣
 ↪last)
```

```
    <ipython-input-7-300e9f2120ad> in <module>
     14 print(new_car._Car__id_type)
     15
---> 16 print(new_car.__id_type)


    AttributeError: 'Car' object has no attribute '__id_type'
```

Private attribute names are mangled by prefixing the _ and the class name.

Name mangling is about safety, not security: it's designed to prevent accidental access and not intentional wrongdoing.

[4]:
```python
new_car._Vehicle__id_type = 'nothing'
print(new_car._Vehicle__id_type)
```

```
nothing
```

Attributes with a single _ prefix are called "protected", but the single underscore prefix has no special meaning to the Python interpreter when used in attribute names.

It is a strong convention among Python programmers that you should not access such attributes from outside the class

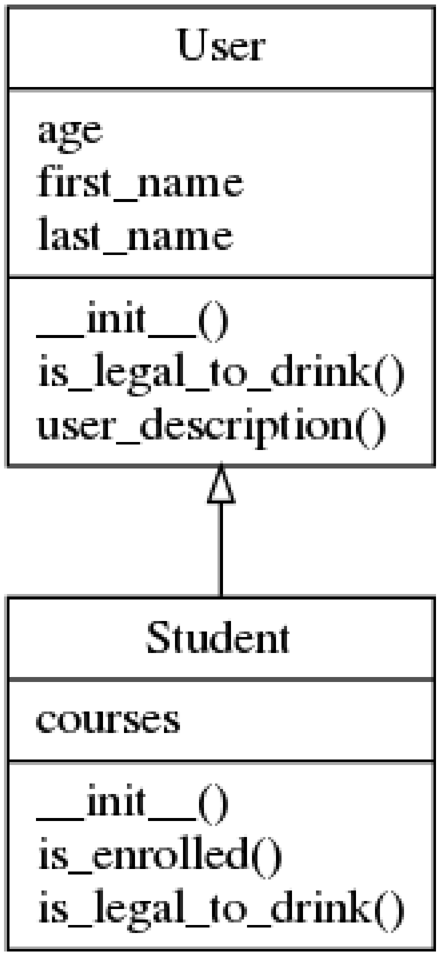
[9]:
```python
# %load product.py
class Product:

    def __init__(self, name, label, qty):
        self.name = name
        self.label = label
        self._qty = qty

    def take_one_item(self):
        self._qty -= 1


chocolate = Product('chocolate', 'Mars', 10)

print(chocolate.name)
print(chocolate.label)
print(chocolate._qty)

chocolate._qty = 2
print(chocolate._qty)
```

```
chocolate
Mars
10
2
```

3 Interfaces

An interface is the set public attributes (methods or data attributes) implemented or inherited by a class.

[12]:
```
# %load user.py
class User(object):
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def user_description(self):
        return f'The user {self.first_name} {self.last_name[0]}. age: {self.
 ↪age}'

    def is_legal_to_drink(self):
        return self.age >= 21


class Student(User):
    def __init__(self, first_name, last_name, age, courses):
        self.courses = courses
        super().__init__(first_name, last_name, age)

    def is_enrolled(self, course):
        return course in self.courses

    def is_legal_to_drink(self):
        return self.age >= 18
```



[14]:
```
dir(Student)
```

[14]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'is_enrolled',
 'is_legal_to_drink',
 'user_description']
```

[15]:
```
help(Student)
```

```
Help on class Student in module __main__:

class Student(User)
 |  Student(first_name, last_name, age, courses)
 |
 |  Method resolution order:
 |      Student
 |      User
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, first_name, last_name, age, courses)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  is_enrolled(self, course)
 |
 |  is_legal_to_drink(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from User:
 |
 |  user_description(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from User:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

4 Protocols

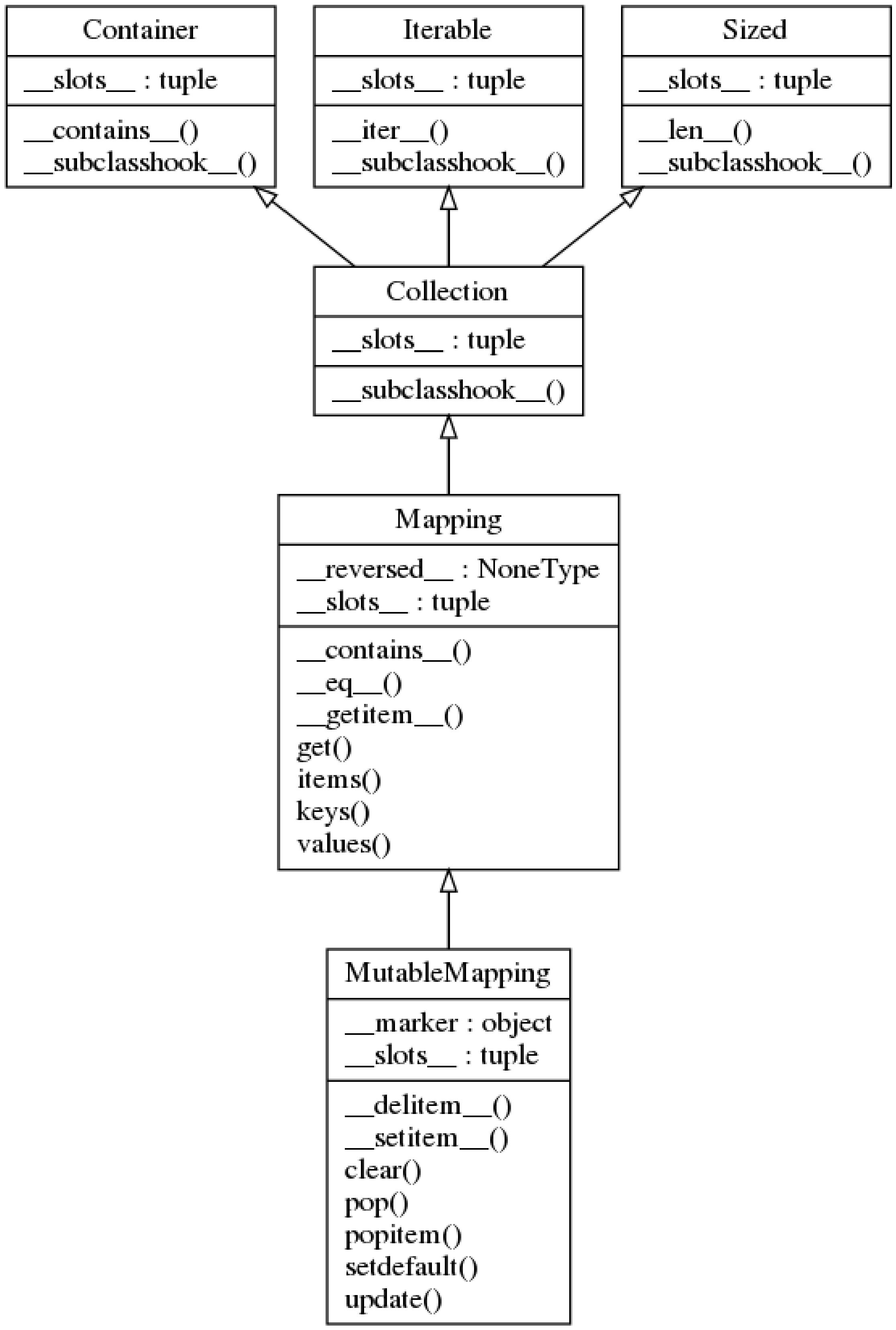
> A procotol is an interface seen as a set of methods to fulfill a role.
>
> Protocols are independent of inheritance; a class may implement several protocols, enabling its instances to fulfill several roles.

Many Python native protocols are now encapsulated by abstract base classes:

ABC	Inherits from	Abstract Methods	Mixin Methods
`Container`		`__contains__`	
`Hashable`		`__hash__`	
`Iterable`		`__iter__`	
`Iterator`	`Iterable`	`__next__`	`__iter__`
`Reversible`	`Iterable`	`__reversed__`	
`Generator`	`Iterator`	`send`, `throw`	`close`, `__iter__`, `__next__`
`Sized`		`__len__`	
`Callable`		`__call__`	
`Collection`	`Sized`, `Iterable`, `Container`	`__contains__`, `__iter__`, `__len__`	
`Sequence`	`Reversible`, `Collection`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
`MutableSequence`	`Sequence`	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited `Sequence` methods and `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
`ByteString`	`Sequence`	`__getitem__`, `__len__`	Inherited `Sequence` methods
`Set`	`Collection`	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, and `isdisjoint`
`MutableSet`	`Set`	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Inherited `Set` methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
`Mapping`	`Collection`	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
`MutableMapping`	`Mapping`	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Inherited `Mapping` methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
`MappingView`	`Sized`		`__len__`
`ItemsView`	`MappingView`, `Set`		`__contains__`, `__iter__`
`KeysView`	`MappingView`, `Set`		`__contains__`, `__iter__`
`ValuesView`	`MappingView`		`__contains__`, `__iter__`
`Awaitable`		`__await__`	
`Coroutine`	`Awaitable`	`send`, `throw`	`close`
`AsyncIterable`		`__aiter__`	
`AsyncIterator`	`AsyncIterable`	`__anext__`	`__aiter__`
`AsyncGenerator`	`AsyncIterator`	`asend`, `athrow`	`aclose`, `__aiter__`, `__anext__`

Source: https://docs.python.org/3.6/library/collections.abc.html

5 What's actually a dictionary?



[27]:
```
# %load dict.py
from collections.abc import MutableMapping, Mapping, Iterable, Collection,␣
↪Container, Sized

empty_dict = {}

print(type(empty_dict))

abcs = [MutableMapping, Mapping, Iterable, Collection, Container, Sized]
for abc in abcs:
    print(
        'empty_dict is a {}: {}'.format(
            abc.__name__, isinstance(empty_dict, abc))
    )
```

```
<class 'dict'>
empty_dict is a MutableMapping: True
empty_dict is a Mapping: True
empty_dict is a Iterable: True
empty_dict is a Collection: True
empty_dict is a Container: True
empty_dict is a Sized: True
```

[29]:
```
# %load stud_collection.py
from collections.abc import MutableMapping, Mapping, Iterable, Collection,␣
↪Container, Sized


class Student(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __repr__(self):
        class_name = type(self).__name__
        return '{} ({} {} - {}) [{}]'.format(
            class_name, self.first_name,
            self.last_name, self.age, id(self)
        )

    def __str__(self):
        return '{} {}, {}'.format(
            self.first_name, self.last_name, self.age)
```

```python
class StudentsCollection:

    def __init__(self, students_list=None):
        self._students = list(students_list) if students_list else []

    def __iter__(self):
        return iter(self._students)

    def __getitem__(self, index):
        return self._students[index]


students = [
    Student('John', 'Doe', 19),
    Student('Jack', 'Fluffy', 18),
    Student('Matthew', 'Wu', 19),
    Student('Heather', 'Rafferty', 19),
    Student('Randall', 'Blackdall', 20),
    Student('Marissa', 'Raynaud', 19),
    Student('Marlo', 'Ranbot', 19)
]

stud_collection = StudentsCollection(students)

print(type(stud_collection))

abcs = [MutableMapping, Mapping, Iterable, Collection, Container, Sized]
for abc in abcs:
    print(
        'stud_collection is a {}: {}'.format(
            abc.__name__, isinstance(stud_collection, abc))
    )
```

```
<class '__main__.StudentsCollection'>
stud_collection is a MutableMapping: False
stud_collection is a Mapping: False
stud_collection is a Iterable: True
stud_collection is a Collection: False
stud_collection is a Container: False
stud_collection is a Sized: False
```

6 Challenge!

Make StudentsCollection a sized collection.