EPAM Python Software Engineer Training

Lesson 2: Python classes

Contents

1 Course

1.1 Classes

In Python there are old-style and new-style classes. Since Python version `2.4` all buildin classes are new-style and a usage of only new style classes is strongly encouraged as since Python version `3.0` a old-style are not supported. The simplest new-style class is `class A(object): pass`. Since Python version `3.0` an explicit inheritance from *object* is optional.

Classes can be inherited. Multiple inheritance is allowed but not encouraged.

Python classes (expectedly) can have instances which behave as defined by a class. An object contains a reference to its class that is used by an interpreter to access class runtime information.

1.2 Attributes

In Python objects can have attributes... and classes can have attributes (as they are objects too).

Attributes can be read from and written too, they also can be deleted using a `del` statement. It is possible to read class attributes' values from a class instance, but it is allowed to write and delete instance attributes only. This is a convenient way to access all class attributes from their class instances and override them when necessary but leave a class intact. Consider the following interactive Python session for explanation:

```
>>> class A(object):
...     class_attr = 1
...
>>> b = A()
>>> b.obj_attr = 2
>>> b.class_attr
1
>>> b.obj_attr
2
>>> A.class_attr = 3
>>> b.class_attr
3
>>> b.class_attr = 4
>>> A.class_attr
3
>>> b.class_attr
4
```

It is possible to access not only attributes of a class within an class instance, but also attributes of its base classes. In case if the same attribute is declared in several classes (especially in case of a multiple inheritance), interpreter will read an attribute of the first class inside a magic attribute `__mro__` that have it. An `__mro__` is a list of all base classes with a multiple inheritance graph resolved into a linear structure. It is possible to access a class attribute from a class instance level through a magic `__class__` object attribute and an attribute of a base class using a `super` function.

All object attributes are stored in a magic attribute `__dict__` and can even be assigned through it. However, keeping an entire dictionary of all attributes in a class is quite inefficient from a memory usage point of view. Hence, if some kind of objects need to be created in large amounts it makes sense to declare a class with `__slots__` attribute in which case attributes will be accessed by a memory alignment just like in C structs and a `__dict__` attribute will not be available in this class instances. Most primitive Python types are declared with a `__slots__` attribute for efficiency.

1.3 Methods

Methods are functions assigned to class attributes i.e. each function assigned to a class as an attribute is wrapped into a method. A method can be either *bound* or *unbound*. When accessed through a class instance - a method is *bound* to that instance and translates a function call in such a way that a class instance is passed into an underlying function as the first argument. When accessed through a class - a method is *unbound* and all arguments to it shall be passed explicitly.

As all methods are class attributes - they undergo the same rules applicable to class attributes (e.g. `__mro__`, `super` etc). Note that if a function is assigned to a class instance directly - it does not become a method (as methods are class attributes, not instance attributes) and it will not receive a class instance as the first argument implicitly.

Besides regular methods a class can also have *class* and *static* methods using an appropriate decorator (`classmethod` and `staticmethod` accordingly). A *class* method is a method which receives a class itself as the first argument instead of a class instance. It is accessible through derived classes and class instances. A *static* method is a method that receives neither a class nor a class instance as the first argument. It is accessible through a declaring class only.

Based on the above one could state that all methods in Python are *virtual* except for *static* methods which are not. Although, a term *virtual* is not applicable to Python.

1.4 Properties

In Python a *property* is a syntax sugar allowing to invoke a method as an attribute. A property is declared using a decorator and it can consist of a *getter*, a *setter*, and a *deleter* all of which are optional and are used to get, set, and delete a property value accordingly.

There are quite limited set of use case when properties are really necessary in Python as it encourages a direct attribute access and encapsulation of a logic rather than a data.

1.5 Magic Methods

In Python there are a set of *magic* methods that may be declared in classes to define some performance critical common low-level behavior. All *magic* methods start and end in two underscores (e.g. __new__) and unlike regular methods they are JIT-compiled *statically* rather than *dynamically* to speed up their invokation. One side effect of this is that it is not possible to assign a *magic* method to a class instance to override its behavior (it will not be called).

Some *magic* methods can be declared to allow a specific class's instances behave like numbers, collections, strings, generators and so on. The others allow to alter different Python object model semantics like class or class instance creation, attribute access etc.

1.6 Metaclasses

Metaclasses in Python are class builders which allow to tweak class creation procedure. For example, a metaclass might generate class methods on the fly. They are rarely used are are out of scope for an introduction Python course.

2 Tasks

2.1 Shapes

Build a class hierarchy for a primitive graphic editor figures data model. Two basic entities of a graphic editor are a *Color* and a *Coordinates* which are building blocks for all other entities. *Coordinates* can be defined in several ways (Linear, Cyllindric, Spheric) through *static* methods. A conversion logic between them is out of scope for this task, for simplicity just store a coordinates type in a field.

There are several basic shapes: a *Point*, a *Line*, a *Circle*, a *Rectangle*, and a *Triangle* - each defined by a different combination of *Color* and *Coordinates*. A line can have a *Pattern* consisting of a list of (*Color*, length) tuples; more complex shapes can be filled with a *Color* or not (be transparent) and each their border can still have a *Pattern*.

Within a course of this task no other methods than are necessary to create objects are required.

2.2 Truth Table

Write a Mixin class that checks if a given class instance instance is True or False based on a truth table. A truth table is a list of object hashes that would evaluate to True (or False) specified as a class attribute. For example, the following code snippet shall print `True True`:

```python
class TrueTest(int, TruthTable):

    __true_values__ = (0, 1, 2, 3)

class FalseTest(str, TruthTable):

    __false_values__ = ('false', hash('no'))

print bool(TrueTest(0)), bool(FalseTest(''))
```

2.3 Memento

Write a context manager class that takes an object, its attribute name and value and sets that attribute, but later restores an original attribute value in a way suitable for a *with* statement. For example, the following code snippet shall print `Did you want to exit?`:

```
with memento(sys, 'exit', lambda x: 'Did you want to exit?'):
    print sys.exit(1)
```

Compare a performance and readability with the same solution using a *contextlib* library.

2.4 Custom Dictionary

Write a function which returns a custom dictionary class which allows to set a predefined set of custom attributes (but not an arbitrary attribute). E.g. the following code snippet shall work just fine:

```
Test = dict_with_attrs('test', 'other')
d = Test({'a': 1}, test='test')
d.other = 'Hey!'
d[10] = 11

# This shall fails:
d.unknown = 42
```

2.5 Proxy

Write a universal transparent proxy that is able to provide read/write access to attributes of any object instance being proxied. For example, the following code snippet shall print `Hello World!`:

```
class A(object):

    phrase = 'Test'

    def test(self):
        print self.phrase

proxy = Proxy(A())
proxy.phrase = 'Hello World!'
proxy.test()
```

In addition, a proxy shall count how many times a proxied object methods were called (separately for each method).

Note

A method can be accessed but not called, hence, you need to proxy method objects as well to fulfill this task. At the same moment, any read/write operation on method proxy shall be delegated to an original method as well.