py-class-pool Documentation

Release 0.1

The ZHR Matrix Team

Jan 09, 2019

Basics:

py-class-pool is a simple but powerful class registry for Python with additional Django support. It allows to build dynamic registry of classes in a similar way how Django handles registry of models. Classes are distinguished by ID.

Warning: This library is used on production environment however please note that it has been not used widely in different use cases. Please consider this before using it on production environment. Any issue reports and improvement suggestions are welcomed.

CHAPTER 1

Installation

```
pip install class-pool
```

CHAPTER 2

Features

- register classes in four ways:
  - on-demand using register() method
  - using register() decorator
  - by metaclass
  - by scanning list of provided packages/modules
- access classes by ID
- **each registry is a singleton**
- define custom ID generator
- specify if abstract classes should be registered or not
- listen for on register signal to run additional logic when class is registered
- Django support which includes loading classes from applications, choice field which allows to select registered classes and corresponding form field
- Django REST Framework support with serializer field and dynamic serializers loaded from a pool
- and many more

CHAPTER 3

Requirements

- Python 3
- Django to use django related features (optional)
- Django REST Framework to use DRF related features (optional)

CHAPTER 4

Contribution

Library is developed and maintained by The Matrix Team, IT team of The Scouts Organisation of the Republic of Poland.

Source code is available at GitLab. Everyone can contribute to this library by making a PR, reporting a bug or suggesting an improvement.

4.1 Class registration

Class can be registered in the pool (registry) in multiple ways. It's allowed to register different classes in the same pool using different methods. Below, base use cases for all four methods are discussed. For more details check Advanced Topics.

4.1.1 register() method

This method allows to register class in the pool on demand. You need only pool instance and class, mechanism is very similar to the way how admin pages are registered in Django Admin Panel, but even you are not familiar with Django snippet below is self explainable.

```
>>> from class_pool import Pool
>>>
>>> pool = Pool()
>>>
>>> class FooClass:
>>>     pass
>>>
>>> pool.register(FooClass)
>>>
```

That's all! Now, `FooClass` is registered in the global `Pool`.

Note: All pool instances shares common state, so each pool class can be considered as singleton.

By default class identifier is generated from class name, so now you can get class from the pool using following call:

```
>>> pool.get('FooClass')
```

More advanced examples can be found at *register() decorator*.

See *Getting registered class* for more information about accessing classes in the pool.

4.1.2 register() decorator

`register()` method can be also used as a class level decorator. If class definition and registration in the pool are done at the same module it's much easier and readable to use the decorator.

```
>>> from class_pool import Pool
>>>
>>> pool = Pool()
>>> # Note: all pool instances created from the same class shares common
>>> # registry (see Singleton pattern chapter for more details)
>>>
>>> @pool.register
>>> class FooClass:
```

More advanced examples can be found at *register() decorator*.

4.1.3 Metaclass

Many times there is a need to register all classes which derives from one base class. For example core project has implemented base plugins class which can be used by 3rd parties to build community plugins. It'll be great if all plugins can be registered automatically without explicit `register()` method call. Using pool's metaclass it's possible to register all classes created by this metaclass automatically. Let's go through next example:

```
>>> # somewhere in core project...
>>>
>>> from class_pool import Pool
>>> pool = Pool()
>>>
>>> class BasePlugin(metaclass=pool.metaclass()):
>>>     pass
>>>
>>> # and 3rd party code
>>>
>>> class FooPlugin(BasePlugin):
>>>     pass
>>>
```

Firstly we need to specify metaclass for our example `BasePlugin` class. To do that we are using `metaclass()` method from the `pool`. This method produce and return a metaclass connected with the pool. Don't worry! Metaclasses are cached locally, so this method is doing the job only once per pool class (in this case `Pool`).

Note: All pool instances shares common state, so each pool class can be considered as singleton.

Now, all classes which derives from `BasePlugin` will be automatically registered. By default base class is **not registered**. If you need to register it, set `register_base` argument in `metaclass` method to `True`:

```
>>> class BasePluginAlsoRegistered(metaclass=pool.metaclass(register_base=True)):
>>>     pass
```

More information about pool's metaclass usage can be found in *Auto-registration by metaclass*.

4.1.4 Modules scanning

All registration ways shown above has one disadvantage. Registration is done when Python executes module where class is declared. In many cases that's enough, but sometimes it will be great if all classes can be registered with doing explicit imports somewhere only to register them. To solve this problem, pools comes with simple packages scanning mechanism based on Django's models scanning mechanism. By creating new pool programmer can define name of module which holds classes to register. Later, pool can be instantiated once and list of packages can be passed to perform automatic imports and registration.

```
>>> pool = Pool.new(base_class=object, module_lookup='plugins')
>>> pool.populate([
>>>     'package_foo',
>>>     'package_bar'
>>> ])
>>>
```

Following example creates *Custom pool* which has auto-registration enabled by defining `module_lookup` attribute. When `populate()` method is called, each package is imported and checked if contains given module, if so all members are retrieved and tried to be registered in the pool.

Note: To get know how to filter classes which can be registered in the pool see *Filter classes to register*.

4.2 Getting registered class

Classes in the pool can be accessed by `__getitem__` operator which raises `KeyError` exception if requested class does not exist:

```
>>> cls = pool['FooClass']
>>> # will raise KeyError exception if FooClass is not registered
```

To make safe retrieval use `get()` method, which in opposite to standard `get()` in dictionaries does not take `default` argument but returns default class specified in the pool. If pool does not have default class, `None` is returned. `Pool` class has no default set, but you can create *Custom pool* to define default class. `get()` method never raises `KeyError` exception.

```
>>> cls = pool.get('FooClass')
>>> # will return None if FooClass is not registered and there is no
>>> # default class set in the pool
```

To determine if given identifier is registered in the pool use `__contains__` operator:

```
>>> if 'FooClass' in pool:
>>>     # do something...
```

4.3 Iterating over the pool

Pools are iterable and returns tuples of two elements, where first one is an ID, second one is class. Internal class registry represented as a regular Python dictionary can be accessed by `classes` property, but you should never make any changes in this object.

Deprecated since version 0.5: `classes` property is deprecated and will be removed in future releases.

4.4 Default class

Sometimes pool has a set of classes which specialises some operations, but in general one common class can be defined for all standard cases. To avoid registering same class multiple times using different IDs to cover all cases `Pool` allows to define default class which is returned every time when class of requested ID does not exist in the pool.

Warning: Default class is working only with `get()` method. `__getitem__` operator raises `KeyError` even if default class is defined!

Because all pool instances shares common state, we need to create *Custom pool* class which derives from `Pool`. Pools come with a `new()` helper function which allows to create custom pools with common customizations without declaring new classes explicitly.

```
>>> from class_pool import Pool
>>>
>>> class DefaultClass:
>>>     pass
>>>
>>> pool = Pool.new(base_class=object, default=DefaultClass)
>>>
>>> pool.get('NotExistingClass')
>>> # returns DefaultClass
```

Note: `new()` helper returns already **instance** of the custom pool, not class. Each `new()` method call creates new custom pool which **does not** shares common state each other. If you need *Singleton pattern* behaviour you have to define custom class pool explicitly. See *Custom pool* for more details.

4.5 Custom pool

To create more pools or make some common customizations declaring custom pools is needed. Because all pool instances shares common state, creating new instance of `Pool` class does not create new pool. This is indeed behaviour explained in more details in *Singleton pattern*.

The simplest way to create custom pool is using `new()` helper as in *Default class* example. This way allows to create pools fast, but is limited. `new()` returns pool instance of dynamically created class which is not accessible anymore. The instance needs to be passed to all places where is needed. It's not possible to get the same custom pool instance again using `new()` helper.

More complex and powerful way to declare custom pools is subclassing the `Pool` class. It allows to customize not only basics but also more advanced options including class filtering described in *Filter classes to register*.

4.5.1 new() helper

The `new()` helper creates dynamically custom pool class and returns its **instance**. There is only one mandatory argument which is a `base_class`. Pool will allow to register only classes which are subclasses of given base. Usually it will be `object` but you can pass any Python class as a base.

There is also couple of optional arguments:

- `module_lookup` which allows to specify module which should be scanned when `populate()` method is called. See *Modules scanning* for more details.
- `default` which allows to specify default class which should be returned if class of requested ID is not registered. See *Default class* for more details.
- `abstract` to filter abstract classes. By default this value is `False` which means that abstract classes cannot be registered in the pool. To allows abstract classes registration set this to `True`. See *Filter classes to register* for more details.

Note: In default `Pool` implementation class abstraction is determined using `isabstract()` method from `inspect` module.

4.5.2 `Pool` subclasses

By subclassing the `Pool` class it's possible to customize any pool behaviour. It's advanced topic and usually you don't need to override pool methods. However, sometimes it's needed to make reusable pool which can be instantiated many times to take power of *Singleton pattern*. Fortunately, custom pools creation by subclassing as simple as creating a subclass in Python.

```
>>> from class_pool import Pool
>>>
>>> class AwesomePool(Pool):
>>>     abstract = True
>>>
```

In the example above, new `AwesomePool` is declared which allows to hold abstract classes. All arguments described in *new() helper* are applicable in subclass customization as attributes.

Note: In opposite to *new() helper* it's not needed to define `base_class` in custom subclasses. It's already set to `object`.

Remember, all instances of particular custom pool shares the same state, but multiple custom pool classes **does not** shares state between each other and/or global `Pool` class. Common state is shared only between instances of exactly the same class. *Singleton pattern* covers this topic in more details.

4.6 Django support

Warning: Following features are available only if Django is installed.

4.6.1 Model field

To add field which allows to select class from given pool use `PoolChoiceField` from `class_pool.db` module. This field extends standard `CharField` and dynamically build `choices` by iterating the pool.

Warning: Because `PoolChoiceField` builds list of choices in `__init__()` method, so pool **has to be** fully populated before field is created (which means before Django imports models). See Django AppConfig documentation to get know how django loads applications and models.

Usually importing module with classes to be registered in `apps.py` file is a good place.

```
>>> from django.db import models
>>> from class_pool.db import PoolChoiceField
>>> from myapp.pool import plugins_pool
>>>
>>> class CustomModel(models.Model):
>>>     plugin = PoolChoiceField(pool=plugins_pool)
>>>
```

`PoolChoiceField` requires only one mandatory argument `pool` which takes pool instance or path to pool instance to be imported lazily. It a good practice to pass pool always as a path to avoid circular imports. Remember! Choice field takes always an instance of pool not a class!

```
>>> plugin = PoolChoiceField(pool='myapp.pool.plugins_pool')
>>>
>>> # ... and in myapp.pool module:
>>> from class_pool import Pool
>>> plugins_pool = Pool()
>>>
```

Note: By default `PoolChoiceField` works with IDs not longer than 40 chars. If you have longer IDs you must set `max_length` appropriately.

Django migrations are supported.

4.6.2 Form field

Currently there is no specialised field for Django forms, but if you need to play with model forms `PoolChoiceField` should be mapped in the same way as regular Django `CharField` with `choices` attribute defined.

4.7 Django REST Framework support

Warning: Following features are available onl if Django REST Framework is installed.

4.7.1 Serializer field

Using `PoolChoiceField` from `class_pool.drf` it's possible to declare serializer field which allows to select one of class identifier from given pool. Field behaviour is very similar to *Model field* and the only difference is that serializer field extends DRF `ChoiceField`. There is also only one mandatory argument `pool` which accepts pool **instance** or path to it. Like model field, serializer field also imports pool lazily if path is provided instead of pool instance.

Warning: Because `PoolChoiceField` builds list of choices in `__init__()` method, so pool **has to be** fully populated before field is created. Make sure that all modules with classes to be registered are imported before serializer is imported.

```
>>> from django_rest_framework import serializers
>>> from class_pool.drf import PoolChoiceField
>>>
>>> class CustomSerializer(serializers.Serializer):
>>>     plugin = PoolChoiceField(pool='myapp.pool.plugins_pool')
```

4.7.2 Serializers

Sometimes serializer (or embedded serializer) structure depends on *type-like* field. Let's analyse JSON below which represents a subset of available unit definitions in a game. `abilities` field structure depends on `type` field value.

```
[{
  "type": "soldier",
  "hp": 100,
  "abilities": {
    "attack": 10,
    "kind": "sword"
  }
}, {
  "type": "worker",
  "hp": 50,
  "abilities": {
    "mining": 5
  }
}]
```

Of course there could be one serializer for `abilities` field which contains all possible subfields for all types but this approach won't work if not all units have the same abilities. Usually this limitation is workaround by converting `abilities` object to a list which has a name, value pairs, for instance:

```
...
"abilities": [
    {
        "name": "attack",
        "value": 10,
    }, {
        "name": "kind",
        "value": "sword"
    }
]
...
```

This approach allows to validate list of abilities depending on `type` field in quite generic field but in many cases it's over-engineered approach which increases serializes logic complexity and decrease API readability.

To solve such cases, `pool_class` library provides two serializers which dynamically validate given data using serializer taken from a pool and pointed by *type-like* field.:

- *TypedSerializer*, for cases when type field is at the same level as dynamic object

```
{
  "type": "foo",
  "dynamic": {
    ... structure depends on type field value ...
  }
}
```

- *PoolSerializer*, when type field is a part of dynamic object

```
{
  "type": "foo",
  ... the rest of fields depends on type field value ...
}
```

TypedSerializer

```
>>> from class_pool.drf import TypedSerializer
>>>
>>> class UnitSerializer(serializers.Serializer):
>>>     type = serializers.ChoiceField()      # or PoolChoiceField
>>>     name = serializers.CharField()
>>>     abilities = TypedSerializer(pool='pool.abilities_serializer')
>>>
>>> # { // UnitSerializer
>>> #   "type": "",
>>> #   "name": "",
>>> #   "abilities": {  // TypedSerializer for abilities field,
>>> #                      which structure depends on type field value
>>> #       ...
>>> #   }
>>> # }
```

When DRF is going to serialize or validate `abilities` field, `TypedSerializer` is getting value from `type` field and using it to get serializer class from `pool.abilities_serializer` pool using `get()` method. If there is matching serializer class, it's instantiated and used to serialize `abilities` field data.

`TypedSerializer` takes one mandatory argument `pool` which has to be a pool instance or path to it. Optional `type_field` can be used to specify custom name of type field. By default the name is set to *type*.

```
>>> unit = serializer.ChoiceField()
>>> abilities = TypedSerializer(pool='pool.abilities_serializer', type_name='unit')
```

PoolSerializer

```
>>> from class_pool.drf import PoolSerializer
>>>
>>> class UnitSerializer(PoolSerializer):
```

(continues on next page)

(continued from previous page)

```
>>>     def __init__(self, *args, **kwargs):
>>>         super().__init__(pool='pool.abilities_pool', *args, **kwargs)
>>>
>>> # { // UnitSerializer with structure depends on type field
>>> #   "type": "",
>>> #   ...
>>> # }
```

or

```
>>> from class_pool.drf import PoolSerializer
>>>
>>> class UnitWrapper(serializers.Serializer):
>>>     unit = PoolSerializer(pool='pool.units_pool')
>>>
>>> # {     // UnitWrapper serializer
>>> #   "unit": {   // PoolSerializer which structure depends on type field value
>>> #     "type": "",
>>> #     ...
>>> #   }
>>> # }
```

When DRF is going to serialize or validate `PoolSerializer` it's getting `type` field value from given object and then serializer class from `pool.abilities_serializer` pool using `get()` method. If there is matching serializer class, it's instantiated and used to process the data.

`PoolSerializer` takes one mandatory argument `pool` which has to be a pool instance or path to it. Currently type field name is hardcoded to *type* and it's not possible to customize it.

`PoolSerializer` works also with lists:

```
>>> class EmbarkedUnits(serializers.Serializer):
>>>     units = PoolSerializer(pool='pool.units_pool', many=True)
>>>
>>> # {     // EmbarkedUnits serializer
>>> #   "units": [{   // PoolSerializer which structure depends on type field value
>>> #     "type": "",
>>> #     ...
>>> #   }, {
>>> #     "type": "",
>>> #     ...
>>> #   }
>>> # }
```

4.8 register() decorator

4.9 Custom class ID generator

4.10 Filter classes to register

4.11 Auto-registration by metaclass

4.12 Auto-registration by importing

4.13 “on register” signal

4.14 Singleton pattern

CHAPTER 5

Indices and tables

- genindex
- modindex
- search