klass-registry Documentation

Igor Quintanilha

Apr 23, 2019

Contents:

CHAPTER 1

Getting Started

As you saw in the *introduction*, you can create a new registry using the `klass_registry.ClassRegistry` class.

`ClassRegistry` defines a `register` method that you can use as a decorator to add classes to the registry:

```
from klass_registry import ClassRegistry

pokedex = ClassRegistry()

@pokedex.register('fire')
class Charizard(object):
  pass
```

Once you've registered a class, you can then create a new instance using the corresponding registry key:

```
sparky = pokedex['fire']
assert isinstance(sparky, Charizard)
```

Note in the above example that `sparky` is an *instance* of `Charizard`.

If you try to access a registry key that has no classes registered, it will raise a `klass_registry.RegistryKeyError`:

```
from klass_registry import RegistryKeyError

try:
  tex = pokedex['spicy']
except RegistryKeyError:
  pass
```

1.1 Registry Keys

By default, you have to provide the registry key whenever you register a new class. But, there's an easier way to do it!

When you initialize your `ClassRegistry`, provide an `attr_name` parameter. When you register new classes, your registry will automatically extract the registry key using that attribute:

```
pokedex = ClassRegistry('element')

@pokedex.register
class Squirtle(object):
  element = 'water'

beauregard = pokedex['water']
assert isinstance(beauregard, Squirtle)
```

Note in the above example that the registry automatically extracted the registry key for the `Squirtle` class using its `element` attribute.

1.2 Collisions

What happens if two classes have the same registry key?

```
pokedex = ClassRegistry('element')

@pokedex.register
class Bulbasaur(object):
  element = 'grass'

@pokedex.register
class Ivysaur(object):
  element = 'grass'

janet = pokedex['grass']
assert isinstance(janet, Ivysaur)
```

As you can see, if two (or more) classes have the same registry key, whichever one is registered last will override any the other(s).

Note: It is not always easy to predict the order in which classes will be registered, especially when they are spread across different modules!

If you don't want this behavior, you can pass `unique=True` to the `ClassRegistry` initializer to raise an exception whenever a collision occurs:

```
from klass_registry import RegistryKeyError

pokedex = ClassRegistry('element', unique=True)

@pokedex.register
class Bulbasaur(object):
  element = 'grass'

try:
  @pokedex.register
  class Ivysaur(object):
    element = 'grass'
except RegistryKeyError:
```

(continues on next page)

(continued from previous page)

```
  pass

janet = pokedex['grass']
assert isinstance(janet, Bulbasaur)
```

Attempting to register `Ivysaur` with the same registry key as `Bulbasaur` raised a `RegistryKeyError`, so it didn't override `Bulbasaur`.

1.3 Init Params

Every time you access a registry key in a `ClassRegistry`, it creates a new instance:

```
marlene = pokedex['grass']
charlene = pokedex['grass']

assert marlene is not charlene
```

Since you're creating a new instance every time, you also have the option of providing args and kwargs to the class initializer using the registry's `get()` method:

```
pokedex = ClassRegistry('element')

@pokedex.register
class Caterpie(object):
  element = 'bug'

  def __init__(self, level=1):
    super(Caterpie, self).__init__()
    self.level = level

timmy = pokedex.get('bug')
assert timmy.level == 1

tommy = pokedex.get('bug', 16)
assert tommy.level == 16

tammy = pokedex.get('bug', level=42)
assert tammy.level == 42
```

Any arguments that you provide to `get()` will be passed directly to the corresponding class' initializer.

Hint: You can create a registry that always returns the same instance per registry key by wrapping it in a `ClassRegistryInstanceCache`. See *Factories vs. Registries* for more information.

CHAPTER 2

Factories vs. Registries

Despite its name, `ClassRegistry` also has aspects in common with the Factory pattern.

Most notably, accessing a registry key automatically creates a new instance of the corresponding class.

But, what if you want a `ClassRegistry` to behave more strictly like a registry — always returning the the *same* instance each time the same key is accessed?

This is where `ClassRegistryInstanceCache` comes into play. It wraps a `ClassRegistry` and provides a caching mechanism, so that each time you access a particular key, it always returns the same instance for that key.

Let's see what this looks like in action:

```
from klass_registry import ClassRegistry, ClassRegistryInstanceCache

pokedex = ClassRegistry('element')

@pokedex.register
class Pikachu(object):
  element = 'electric'

@pokedex.register
class Alakazam(object):
  element = 'psychic'

fighters = ClassRegistryInstanceCache(pokedex)

# Accessing the ClassRegistry yields a different instance every
# time.
pika_1 = pokedex['electric']
assert isinstance(pika_1, Pikachu)
pika_2 = pokedex['electric']
assert isinstance(pika_2, Pikachu)
assert pika_1 is not pika_2

# ClassRegistryInstanceCache works just like ClassRegistry, except
# it returns the same instance per key.
```

(continues on next page)

(continued from previous page)

```
pika_3 = fighters['electric']
assert isinstance(pika_3, Pikachu)
pika_4 = fighters['electric']
assert isinstance(pika_4, Pikachu)
assert pika_3 is pika_4

darth_vader = fighters['psychic']
assert isinstance(darth_vader, Alakazam)
anakin_skywalker = fighters['psychic']
assert isinstance(anakin_skywalker, Alakazam)
assert darth_vader is anakin_skywalker
```

CHAPTER 3

Iterating Over Registries

Sometimes, you want to iterate over all of the classes registered in a `ClassRegistry`. There are three methods included to help you do this:

- `items()` iterates over the registry keys and corresponding classes as tuples.
- `keys()` iterates over the registry keys.
- `values()` iterates over the registered classes.

Note: Regardless of which version of Python you are using, all three of these methods return generators.

Here's an example:

```
from klass_registry import ClassRegistry

pokedex = ClassRegistry('element')

@pokedex.register
class Geodude(object):
  element = 'rock'

@pokedex.register
class Machop(object):
  element = 'fighting'

@pokedex.register
class Bellsprout(object):
  element = 'grass'

assert list(pokedex.items()) == \
  [('rock', Geodude), ('fighting', Machop), ('grass', Bellsprout)]

assert list(pokedex.keys()) == ['rock', 'fighting', 'grass']
```

(continues on next page)

(continued from previous page)

```
assert list(pokedex.values()) == [Geodude, Machop, Bellsprout]
```

Tip: Tired of having to add the `register()` decorator to every class?

You can use the `AutoRegister()` metaclass to automatically register all non-abstract subclasses of a particular base class. See *Advanced Topics* for more information.

3.1 Changing the Sort Order

As you probably noticed, these functions iterate over classes in the order that they are registered.

If you'd like to customize this ordering, use `SortedClassRegistry`:

```
from klass_registry import SortedClassRegistry

pokedex =\
  SortedClassRegistry(attr_name='element', sort_key='weight')

@pokedex.register
class Geodude(object):
  element = 'rock'
  weight = 1000

@pokedex.register
class Machop(object):
  element = 'fighting'
  weight = 75

@pokedex.register
class Bellsprout(object):
  element = 'grass'
  weight = 15

assert list(pokedex.items()) == \
  [('grass', Bellsprout), ('fighting', Machop), ('rock', Geodude)]

assert list(pokedex.keys()) == ['grass', 'fighting', 'rock']

assert list(pokedex.values()) == [Bellsprout, Machop, Geodude]
```

In the above example, the code iterates over registered classes in ascending order by their `weight` attributes.

You can provide a sorting function instead, if you need more control over how the items are sorted:

```
from functools import cmp_to_key

def sorter(a, b):
  """
  Sorts items by weight, using registry key as a tiebreaker.

  ``a`` and ``b`` are tuples of (registry key, class)``.
  """
```

(continues on next page)

(continued from previous page)

```
  # Sort descending by weight first.
  weight_cmp = (
      (a[1].weight < b[1].weight)
    - (a[1].weight > b[1].weight)
  )

  if weight_cmp != 0:
    return weight_cmp

  # Use registry key as a fallback.
  return ((a[0] > b[0]) - (a[0] < b[0]))

pokedex =\
  SortedClassRegistry(
    attr_name = 'element',

    # Note that we pass ``sorter`` through ``cmp_to_key`` first!
    sort_key = cmp_to_key(sorter),
  )

@pokedex.register
class Horsea(object):
  element = 'water'
  weight = 5

@pokedex.register
class Koffing(object):
  element = 'poison'
  weight = 20

@pokedex.register
class Voltorb(object):
  element = 'electric'
  weight = 5

assert list(pokedex.items()) == \
  [('poison', Koffing), ('electric', Voltorb), ('water', Horsea)]

assert list(pokedex.keys()) == ['poison', 'electric', 'water']

assert list(pokedex.values()) == [Koffing, Voltorb, Horsea]
```

This time, the `SortedClassRegistry` used our custom sorter function, so that the classes were sorted descending by weight, with the registry key used as a tiebreaker.

Important: Note that we had to pass the sorter function through `functools.cmp_to_key()` before providing it to the `SortedClassRegistry` initializer.

This is necessary because of how sorting works in Python. See Sorting HOW TO for more information.

CHAPTER 4

Entry Points Integration

A serially-underused feature of setuptools is its entry points. This feature allows you to expose a pluggable interface in your project. Other libraries can then declare entry points and inject their own classes into your class registries!

Let's see what that might look like in practice.

First, we'll create a package with its own `setup.py` file:

```
# generation_2/setup.py

from setuptools import setup

setup(
  name = 'pokemon-generation-2',
  description = 'Extends the pokédex with generation 2 pokémon!',

  entry_points = {
    'pokemon': [
      'grass=gen2.pokemon:Chikorita',
      'fire=gen2.pokemon:Cyndaquil',
      'water=gen2.pokemon:Totodile',
    ],
  },
)
```

Note that we declared some `pokemon` entry points.

Let's see what happens once the `pokemon-generation-2` package is installed:

```
% pip install pokemon-generation-2
% ipython

In [1]: from klass_registry import EntryPointClassRegistry

In [2]: pokedex = EntryPointClassRegistry('pokemon')
```

(continues on next page)

(continued from previous page)

```
In [3]: list(pokedex.items())
Out[3]:
[('grass', <class 'gen2.pokemon.Chikorita'>),
 ('fire', <class 'gen2.pokemon.Cyndaquil'>),
 ('water', <class 'gen2.pokemon.Totodile'>)]
```

Simply declare an `EntryPointClassRegistry` instance, and it will automatically find any classes registered to that entry point group across every single installed project in your virtualenv!

4.1 Reverse Lookups

From time to time, you may need to perform a "reverse lookup": Given a class or instance, you want to determine which registry key is associated with it.

For `ClassRegistry`, performing a reverse lookup is simple because the registry key is (usually) defined by an attribute on the class itself.

However, `EntryPointClassRegistry` uses an external source to define the registry keys, so it's a bit tricky to go back and find the registry key for a given class.

If you would like to enable reverse lookups in your application, you can provide an optional `attr_name` argument to the registry's initializer, which will cause the registry to "brand" every object it returns with the corresponding registry key.

```
In [1]: from klass_registry import EntryPointClassRegistry

In [2]: pokedex = EntryPointClassRegistry('pokemon', attr_name='element')

In [3]: fire_pokemon = pokedex['fire']

In [4]: fire_pokemon.element
Out[4]: 'fire'

In [5]: water_pokemon_class = pokedex.get_class('water')

In [6]: water_pokemon_class.element
Out[6]: 'water'
```

We set `attr_name='element'` when initializing the `EntryPointClassRegistry`, so it set the `element` attribute on every class and instance that it returned.

> **Caution:** If a class already has an attribute with the same name, the registry will overwrite it.

CHAPTER 5

Advanced Topics

This section covers more advanced or esoteric uses of ClassRegistry features.

5.1 Registering Classes Automatically

Tired of having to add the `register` decorator to every class that you want to add to a class registry? Surely there's a better way!

ClassRegistry also provides an `AutoRegister()` metaclass that you can apply to a base class. Any non-abstract subclass that extends that base class will be registered automatically.

Here's an example:

```
from abc import abstractmethod
from klass_registry import AutoRegister, ClassRegistry
from six import with_metaclass

pokedex = ClassRegistry('element')

# Note ``AutoRegister(pokedex)`` used as the metaclass here.
class Pokemon(with_metaclass(AutoRegister(pokedex))):
   @abstractmethod
   def get_abilities(self):
     raise NotImplementedError()

# Define some non-abstract subclasses.
class Butterfree(Pokemon):
  element = 'bug'

  def get_abilities(self):
    return ['compound_eyes']

class Spearow(Pokemon):
  element = 'flying'
```

(continues on next page)

(continued from previous page)

```
  def get_abilities(self):
    return ['keen_eye']

# Any non-abstract class that extends ``Pokemon`` will automatically
# get registered in our Pokédex!
assert list(pokedex.items()) == \
  [('bug', Butterfree), ('flying', Spearow)]
```

In the above example, note that `Butterfree` and `Spearow` were added to `pokedex` automatically. However, the `Pokemon` base class was not added, because it is abstract.

Important: Python defines an abstract class as a class with at least one unimplemented abstract method. You can't just add `metaclass=ABCMeta`!

```
from abc import ABCMeta

# Declare an "abstract" class.
class ElectricPokemon(with_metaclass(ABCMeta, Pokemon)):
  element = 'electric'

  def get_abilities(self):
    return ['shock']

assert list(pokedex.items()) == \
  [('bug', Butterfree), \
   ('flying', Spearow), \
   ('electric', ElectricPokemon)]
```

Note in the above example that `ElectricPokemon` was added to `pokedex`, even though its metaclass is `ABCMeta`.

Because `ElectricPokemon` doesn't have any unimplemented abstract methods, Python does **not** consider it to be abstract.

We can verify this by using `inspect.isabstract()`:

```
from inspect import isabstract
assert not isabstract(ElectricPokemon)
```

5.2 Patching

From time to time, you might need to register classes temporarily. For example, you might need to patch a global class registry in a unit test, ensuring that the extra classes are removed when the test finishes.

ClassRegistry provides a `RegistryPatcher` that you can use for just such a purpose:

```
from klass_registry import ClassRegistry, RegistryKeyError, \
  RegistryPatcher

pokedex = ClassRegistry('element')

# Create a couple of new classes, but don't register them yet!
class Oddish(object):
```

(continues on next page)

(continued from previous page)

```
  element = 'grass'

class Meowth(object):
  element = 'normal'

# As expected, neither of these classes are registered.
try:
  pokedex['grass']
except RegistryKeyError:
  pass

# Use a patcher to temporarily register these classes.
with RegistryPatcher(pokedex, Oddish, Meowth):
  abbot = pokedex['grass']
  assert isinstance(abbot, Oddish)

  costello = pokedex['normal']
  assert isinstance(costello, Meowth)

# Outside the context, the classes are no longer registered!
try:
  pokedex['grass']
except RegistryKeyError:
  pass
```

If desired, you can also change the registry key, or even replace a class that is already registered.

```
@pokedex.register
class Squirtle(object):
  element = 'water'

# Get your diving suit Meowth; we're going to Atlantis!
with RegistryPatcher(pokedex, water=Meowth):
  nemo = pokedex['water']
  assert isinstance(nemo, Meowth)

# After the context exits, the previously-registered class is
# restored.
ponsonby = pokedex['water']
assert isinstance(ponsonby, Squirtle)
```

Important: Only mutable registries can be patched (any class that extends `MutableRegistry`).

In particular, this means that `EntryPointClassRegistry` can **not** be patched using `RegistryPatcher`.

CHAPTER 6

ClassRegistry

At the intersection of the Registry and Factory patterns lies the `ClassRegistry`:

- Define global factories that generate new class instances based on configurable keys.
- Seamlessly create powerful service registries.
- Integrate with setuptools's `entry_points` system to make your registries infinitely extensible by 3rd-party libraries!
- And more!

6.1 Getting Started

Create a registry using the `klass_registry.ClassRegistry` class, then decorate any classes that you wish to register with its `register` method:

```
from klass_registry import ClassRegistry

pokedex = ClassRegistry()

@pokedex.register('fire')
class Charizard(Pokemon):
  ...

@pokedex.register('grass')
class Bulbasaur(Pokemon):
  ...

@pokedex.register('water')
class Squirtle(Pokemon):
  ...
```

To create a class instance from a registry, use the subscript operator:

```
# Charizard, I choose you!
fighter1 = pokedex['fire']

# CHARIZARD fainted!
# How come my rival always picks the type that my pokémon is weak against??
fighter2 = pokedex['grass']
```

6.1.1 Advanced Usage

There's a whole lot more you can do with ClassRegistry, including:

- Provide args and kwargs to new class instances.
- Automatically register non-abstract classes.
- Integrate with setuptools's `entry_points` system so that 3rd-party libraries can add their own classes to your registries.
- Wrap your registry in an instance cache to create a service registry.
- And more!

To learn more about what you can do with ClassRegistry, *keep reading!*

6.2 Requirements

ClassRegistry is compatible with Python versions 3.6, 3.5 and 2.7.

6.3 Installation

Install the latest stable version via pip:

```
pip install klass-registry
```