autoregistry

Release 0.0.0

Brian Pugh

Nov 27, 2023

CONTENTS:

Invoking functions and class-constructors from a string is a common design pattern that AutoRegistry aims to solve. For example, a user might specify a backend of type `"sqlite"` in a yaml configuration file, for which our program needs to construct the `SQLite` subclass of our `Database` class. Classically, you would need to manually create a lookup, mapping the string `"sqlite"` to the `SQLite` constructor. With AutoRegistry, the lookup is automatically created for you.

AutoRegistry has a single powerful class `Registry` that can do the following:

- Be inherited to automatically register subclasses by their name.
- Be directly invoked `my_registry = Registry()` to create a decorator for registering callables like functions.
- Traverse and automatically create registries for other python libraries.

CHAPTER

ONE

INSTALLATION

AutoRegistry requires Python >=3.8 and can be installed from pypi via:

```
python -m pip install autoregistry
```

To install directly from github, you can run:

```
python -m pip install git+https://github.com/BrianPugh/autoregistry.git
```

For development, its recommended to use Poetry:

```
git clone https://github.com/BrianPugh/autoregistry.git
cd autoregistry
poetry install
```

CHAPTER

TWO

OVERVIEW

All of AutoRegistry's functionality comes from the `Registry` class.

```
from autoregistry import Registry
```

To use the `Registry` class, we can either inherit it, or directly invoke it to create a `Registry` object.

2.1 Inheritance

Generally, when inheriting `Registry`, we are defining an interface, and thusly an abstract base class. The `Registry` class is an instance of `ABCMeta`, so the `abc` decorator `@abstractmethod` will work with subclasses of `Registry`.

```
from abc import abstractmethod
from autoregistry import Registry


class Pokemon(Registry):
    @abstractmethod
    def attack(self, target) -> int:
        pass


class Pikachu(Pokemon):
    def attack(self, target):
        return 5
```

The interface `Pokemon` is defined and currently has one subclass, `Pikachu`. The `Pokemon` **class** can be treated like a dictionary, mapping strings to class-constructors. The keys are derived from the subclasses' names.

```
>>> len(Pokemon)
1
>>> Pokemon
<Pokemon: ['pikachu']>
>>> list(Pokemon)
['pikachu']
>>> pikachu = Pokemon["pikachu"]()
>>> pikachu
<__main__.Pikachu object at 0x10689fb20>
```

Unlike a dictionary, the queries are, by default, case-insensitive:

```
>>> pikachu = Pokemon["pIkAcHU"]()  # Case insensitive works, too.
>>> "pikachu" in Pokemon
True
>>> "PIKACHU" in Pokemon
True
```

If an unregistered string is queried, a `KeyError` will be raised. You can also use the `get` method to handle missing-key queries. If the provided `default` argument is a string, a lookup will be performed.

```
>>> Pokemon["ash"]
KeyError: 'ash'
>>> pikachu = Pokemon.get("ash", "pikachu")()
>>> pikachu = Pokemon.get("ash", Pikachu)()  # The default could also be the constructor.
>>> pikachu = Pokemon.get("ash")()  # If default is not specified, its None.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

The ruleset for deriving keys and valid classnames is configurable. See *Configuration*.

2.2 Decorator

Instead of using classes, you can also use `Registry` to explicitly create a dictionary-like object and use it to decorate functions.

```
from autoregistry import Registry

my_registry = Registry()


@my_registry
def foo(x):
    return x


@my_registry()  # This also works.
def bar(x):
    return 2 * x


# Assigning as you would a dictionary also works
def baz(x):
    return 3 * x


my_registry["baz"] = baz  # The key could be any string.


# You can also register classes this way.
@my_registry
class Baz:
    pass
```

The `my_registry` **object** can be treated like a dictionary, mapping strings to registered functions. The keys are derived from the function names.

```
>>> len(my_registry)
3
>>> my_registry
<Registry: ['foo', 'bar', "baz"]>
>>> list(my_registry)
['foo', 'bar', 'baz']
>>> my_registry["foo"](7)
7
```

You can also pass in an object or a list of objects at registry creation:

```
def foo():
    pass


def bar():
    pass


my_registry = Registry([foo, bar])


@my_registry
def baz():
    pass
```

2.3 Module

Another use of AutoRegistry is to automatically create a registry of an external module. For example, in pytorch, the `torch.optim` submodule contains many optimizers that we may want to configure via a yaml file.

```
import torch
from autoregistry import Registry

optims = Registry(torch.optim)

assert list(optims) == [
    "asgd",
    "adadelta",
    "adagrad",
    "adam",
    "adamw",
    "adamax",
    "lbfgs",
    "nadam",
    "optimizer",
    "radam",
    "rmsprop",
    "rprop",
```

(continues on next page)

(continued from previous page)

```
    "sgd",
    "sparseadam",
    "lr_scheduler",
    "swa_utils",
]
```

CHAPTER

THREE

KEY SPLITTING

Consider the following code example:

```
class Pokemon(Registry, recursive=False):
    pass


class Pikachu(Pokemon):
    pass


class SurfingPikachu(Pokemon):
    pass
```

We can naively access the `SurfingPikachu` constructor via `Pokemon["pikachu"]["surfingpikachu"]`. We can also access the same constructor using dot or slash notation from a single string. The query string will be split on dots and slashes, then iteratively queried:

```
assert SurfingPikachu == Pokemon["pikachu"]["surfingpikachu"]
assert SurfingPikachu == Pokemon["pikachu.surfingpikachu"]
assert SurfingPikachu == Pokemon["pikachu/surfingpikachu"]
```

CHAPTER

FOUR

REVERSE LOOKUP

Consider the following class hierarchy:

```
class Pokemon(Registry, case_sensitive=False):
    pass


class Pikachu(Pokemon):
    pass


class SurfingPikachu(Pokemon):
    pass
```

Subclasses can be accessed via the standard AutoRegistry indexing, i.e:

```
assert Pokemon["pikachu"] == Pikachu
```

To perform the reverse-lookup, i.e. obtain the string `"pikachu"` from the class `Pikachu`, access the `__registry__.name` attribute:

```
assert Pikachu.__registry__.name == "pikachu"
```

CHAPTER

FIVE

CONFIGURATION

5.1 Configuring Inheritance

When inheriting from the `Registry` class, keyword configuration values can be passed along side it when defining the subclass. For example:

```
class Pokemon(Registry, case_sensitive=True):
    pass
```

Each subclass registry will copy the configuration of its parent, and update it with newly passed in values. For example:

```
class Pokemon(Registry, suffix="Type", recursive=False):
    pass


class RockType(Pokemon, suffix=""):
    pass


class Geodude(RockType):
    pass


# it's just "rock" instead of "rocktype" because we strip the suffix by default.
geodude = Pokemon["rock"]["geodude"]()
```

All direct children of `Pokemon` MUST end with `"Type"`. Children of `RockType` will NOT be registered with `RockType`'s parent, `Pokemon` because `recursive=False` is set. For `RockType`, setting `suffix=""` overrides its parent's `suffix` setting, allowing the definition of the subclass `Geodude`, despite it not ending with `"Type"`.

5.2 Configuring Decorator

When directly declaring a `Registry`, configurations are passed as keyword arguments when instantiating the `Registry` object:

```
readers = Registry(suffix="_read")


@readers
```

(continues on next page)

(continued from previous page)

```
def yaml_read(fn):
    pass


@readers()  # This also works.
def json_read(fn):
    pass


# it's just "json" instead of "json_read" because we strip the suffix by default.
data = readers["json"]("my_file.json")
```

5.3 Name Override and Aliases

There are two special configuration values: `name` and `aliases`. `name` overrides the auto-derived string to register the class/function under, while `aliases` registers *additional* string(s) to the class/function, but doesn't impact the auto-derived registration key. `aliases` may be a single string, or a list of strings.

`name` and `aliases` values are **not** subject to configured naming rules and will **not** be modified by configurations like `strip_suffix`. Similarly, directly setting a registry element `my_registry["myfunction"] = myfunction` is not subject to naming rules. However, values are still subject to the `overwrite` configuration and will raise `KeyCollisionError` if `name` or `aliases` attempts to overwrite an existing entry while `overwrite=False`. Additionally, `name` and `aliases` may **not** contain a `.` or a `/` due to *Key Splitting*.

These parameters are intended to aid developers maintain backwards compatibility as their codebase changes.

5.3.1 Inheritance

Name and aliases are provided as additional class keyword arguments.

```
class Pokemon(Registry):
    pass


class Ekans(name="snake"):
    pass


class Pikachu(aliases=["electricmouse"]):
    pass


my_pokemon = []
# Pokemon["ekans"] will raise a KeyError
my_pokemon.append(Pokemon["snake"]())
my_pokemon.append(Pokemon["pikachu"]())
my_pokemon.append(Pokemon["electricmouse"]())
```

To not register a subclass to the appropriate registry(s), set the parameter `skip=True`.

```
class Sensor(Registry):
    pass


class Oxygen(Sensor, skip=True):
    pass


class Temperature(Sensor):
    pass


assert list(Sensor.keys()) == ["temperature"]
```

5.3.2 Decorator

Name and aliases are provided as additional decorator keyword arguments.

```
registry = Registry()


@registry(name="foo")
def foo2():
    pass


@registry(aliases=["baz", "bop"])
def bar():
    pass


assert list(registry) == ["foo", "bar", "baz", "bop"]
```

5.4 Parameters

This section describes and provides examples for all of the configurable options in `autoregistry`.

5.4.1 case_sensitive: bool = False

If `True`, all lookups are case-sensitive. Otherwise, all lookups are case-insensitive. A failed lookup will result in a `KeyError`.

```
class Pokemon(Registry, case_sensitive=False):
    pass


class Pikachu(Pokemon):
    pass
```

(continues on next page)

(continued from previous page)

```
class SurfingPikachu(Pokemon):
    pass


assert list(Pokemon) == ["pikachu", "surfingpikachu"]
assert list(Pikachu) == ["surfingpikachu"]
pikachu = Pokemon["piKaCHu"]()
```

```
class Pokemon(Registry, case_sensitive=True):
    pass


class Pikachu(Pokemon):
    pass


class SurfingPikachu(Pokemon):
    pass


assert list(Pokemon) == ["Pikachu", "SurfingPikachu"]
assert list(Pikachu) == ["SurfingPikachu"]
pikachu = Pokemon["Pikachu"]()

# This will raise a KeyError due to the lowercase "p".
pikachu = Pokemon["pikachu"]()
```

5.4.2 regex: str = ""

Registered items **MUST** match this regular expression. If a registered item does **NOT** match this regex, `InvalidNameError` will be raised.

```
# Capital letters only
registry = Registry(regex="[A-Z]+", case_sensitive=True)


@registry
def FOO():
    pass


# This will raise an InvalidNameError, because the supplied regex only allows for␣
↪capital letters.
@registry
def bar():
    pass


assert list(registry) == ["FOO"]
```

5.4.3 prefix: str = ""

Registered items **MUST** start with this prefix. If a registered item does **NOT** start with this prefix, `InvalidNameError` will be raised.

```
class Sensor(Registry, prefix="Sensor"):
    pass


# This will raise an InvalidNameError because the class name doesn't start with "Sensor"
class Temperature(Sensor):
    pass


class SensorTemperature(Sensor):
    pass
```

5.4.4 suffix: str = ""

Registered items **MUST** end with this suffix. If a registered item does **NOT** end with this suffix, `InvalidNameError` will be raised.

```
class Sensor(Registry, suffix="Sensor"):
    pass


# This will raise an InvalidNameError because the class name doesn't end with "Sensor"
class Temperature(Sensor):
    pass


class TemperatureSensor(Sensor):
    pass
```

5.4.5 strip_prefix: bool = True

If `True`, the `prefix` will be removed from registered items. This generally allows for a more natural lookup.

```
class Sensor(Registry, prefix="Sensor", strip_prefix=True):
    pass


class SensorTemperature(Sensor):
    pass


class SensorHumidity(Sensor):
    pass
```

(continues on next page)

(continued from previous page)

```
assert list(Sensor) == ["temperature", "humidity"]
my_temperature_sensor = Sensor["temperature"]()
```

5.4.6 strip_suffix: bool = True

If `True`, the `suffix` will be removed from registered items. This generally allows for a more natural lookup.

```
class Sensor(Registry, suffix="Sensor", strip_suffix=True):
    pass


class TemperatureSensor(Sensor):
    pass


class HumiditySensor(Sensor):
    pass


assert list(Sensor) == ["temperature", "humidity"]
my_temperature_sensor = Sensor["temperature"]()
```

5.4.7 register_self: bool = False

If `True`, each registry class is registered in its own registry.

```
class Pokeball(Registry, register_self=True):
    def probability(self, target):
        return 0.2


class Masterball(Pokeball):
    def probability(self, target):
        return 1.0


assert list(Pokeball) == ["pokeball", "masterball"]
```

5.4.8 recursive: bool = True

If `True`, all subclasses will be recursively registered to their parents. If registering a `module`, this means all submodules will be recursively traversed.

```
class Pokemon(Registry, recursive=True):
    pass


class Pikachu(Pokemon):
```

(continues on next page)

(continued from previous page)

```
    pass


class SurfingPikachu(Pokemon):
    pass


assert list(Pokemon) == ["pikachu", "surfingpikachu"]
assert list(Pikachu) == ["surfingpikachu"]
```

```
class Pokemon(Registry, recursive=False):
    pass


class Pikachu(Pokemon):
    pass


class SurfingPikachu(Pokemon):
    pass


assert list(Pokemon) == ["pikachu"]
assert list(Pikachu) == ["surfingpikachu"]
```

Consider the following more complicated situation:

```
class ClassA(Registry, recursive=False):
    pass


class ClassB(ClassA):
    pass


class ClassC(ClassB, recursive=True):
    pass


class ClassD(ClassC):
    pass


class ClassE(ClassD):
    pass
```

The registries and configurations are as follows:

- `ClassA` has `recursive=False`, and contains `["classb"]`, its only direct child.
- `ClassB` inherits `recursive=False`, and contains `["classc"]`, its only direct child.
- `ClassC` overrides `recursive=True`, and contains all of its children `["classd", "classe"]`
- `ClassD` inherits `recursive=True`, and contains its child `["classe"]`.

- `ClassE` inherits `recursive=True`, and is empty since it has no children.

5.4.9 snake_case: bool = False

By default, for case-insensitive queries, the key is derived by taking the all-lowercase version of the class name. If `snake_case=True`, the PascalCase class names will be instead converted to snake_case.

Snake case conversion is performed *after* name validation (like `prefix` and `suffix`) checks are performed.

```
class Tools(Registry, snake_case=True):
    pass


class Hammer(Tools):
    pass


class SocketWrench(Tools):
    pass


assert list(Tools) == ["hammer", "socket_wrench"]
```

5.4.10 overwrite: bool = False

If `overwrite=False`, attempting to register an object that would overwrite an existing registered item would result in a `KeyCollisionError`. If `overwrite=True`, then the previous entry will be overwritten and no exception will be raised.

```
registry = Registry()


@registry
def foo():
    pass


# This will raise a ``KeyCollisionError``
@registry
def foo():
    pass
```

```
registry = Registry(overwrite=True)


@registry
def foo():
    return 1


@registry
def foo():
```

(continues on next page)

(continued from previous page)

```
    return 2


assert registry["foo"]() == 2
```

5.4.11 hyphen: bool = False

Converts all underscores to hyphens.

```
tools = Registry(hyphen=True)


@registry
def ballpeen_hammer():
    pass


@registry
def socket_wrench():
    pass


assert list(Tools) == ["ballpeen-hammer", "socket-wrench"]
```

Can be used in conjunction with `snake_case`.

```
class Tools(Registry, snake_case=True, hyphen=True):
    pass


class Hammer(Tools):
    pass


class SocketWrench(Tools):
    pass


assert list(Tools) == ["hammer", "socket-wrench"]
```

5.4.12 transform: Optional[Callable] = None

Provide a custom function to modify the registry for a given function/class name. Must that in a single string argument, and return a string. The `transform` is called as the **final** name processing step, after all other transforms like `snake_case` and `hyphen`.

```
def transform(name: str) -> str:
    return f"shiny_{name}"
```

(continues on next page)

(continued from previous page)

```
class Pokemon(Registry, transform=transform, snake_case=True):
    pass


class Pikachu(Pokemon):
    pass


class SurfingPikachu(Pokemon):
    pass


assert list(Pokemon) == [
    "shiny_pikachu",
    "shiny_surfing_pikachu",
]
```

5.4.13 redirect: bool = True

If `redirect=True`, then methods that would have collided with the dict-like registry interface are wrapped in a redirect object. The redirect object will invoke registry methods if called from the class, e.g. `MyClass.keys()`, but will call the user-defined method if called from an instantiated object, e.g. `my_class.keys()`. Methods decorated with `@classmethod` or `@staticmethod` will not be wrapped; they will override the dict-like registry interface.

```
class Foo(Registry):
    def keys(self):
        return 0


class Bar(Foo):
    pass


foo = Foo()
assert list(Foo.keys()) == ["bar"]
assert foo.keys() == 0
```