Subclass Register Documentation

Release 1.0.1

Yngve Mardal Moe

Aug 16, 2019

Contents

CHAPTER 1

Motivation

This library implements a simple class decorator that you can apply to a base class. This decorator then hooks into the way the decorated class is subclassed, adding all new subclasses to a dictionary whose keys are class names and values are the classes themselves.

The motivation for this project was to autogenerate deep learning models from pure JSON files, thus ensuring reproducibility of the results. I do, however, think that it is ideal for any kind of codebase where we want to generate safe code from configuration files.

CHAPTER 2

Installation instructions

The subclass register can be installed with `pip`:

```
pip install subclass-register
```

by cloning this repo and running `setup.py`

```
git clone https://github.com/yngvem/subclass-register
cd subclass-register
python setup.py
```

or by simply downloading the `src\subclass_register\subclass_register.py` file and the `LISENCE` file into your project.

CHAPTER 3

Documentation

class subclass_register.**SubclassRegister**(*class_type='class'*)

Creates a register instance used to register all subclasses of some base class.

Use the *SubclassRegister.link* decorator to link a base class with the register.

Examples

We create the register as any other class and link it to a base class using the `link_base` decorator.

```
>>> register = SubclassRegister('car')
>>> @register.link_base
... class BaseCar:
...     pass
>>> class SUV(BaseCar):
...     def __init__(self, num_seats):
...         self.num_seats = num_seats
>>> class Sedan(BaseCar):
...     def __init__(self, num_seats):
...         self.num_seats = num_seats
```

The `available_classes` attribute returns a tuple with the class-names in the register

```
>>> register.available_classes
('SUV', 'Sedan')
```

We can also ommit adding a class from the register, using the skip decorator.

```
>>> @register.skip
... class SportsCar(BaseCar):
...     def __init__(self, horse_powers):
...         self.horse_powers = horse_powers
```

We see thawt the `SportsCar` class is not added to the register.

```
>>> register.available_classes
('SUV', 'Sedan')
```

Indexing works as if the register was a dictionary

```
>>> register['SUV']
<class 'subclass_register.subclass_register.SUV'>
```

We can also check if elements are in the register

```
>>> 'SUV' in register
True
```

And delete them from the register

```
>>> del register['SUV']
>>> 'SUV' in register
False
>>> register.available_classes
('Sedan',)
```

We can also manually add classes to the register

```
>>> register['SUV'] = SUV
>>> 'SUV' in register
True
>>> register.available_classes
('Sedan', 'SUV')
```

But we can not overwrite already existing classes in the register

```
>>> register['SUV'] = SUV
Traceback (most recent call last):
  ...
ValueError: Cannot register two classes with the same name
```

If we use a name that is not in the register, we get an error and a list of the available classes sorted by similarity (using difflib)

```
>>> register['sedan'] # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
  ...
NotInRegisterError: sedan is not a valid name for a car.
Available cars are (in decreasing similarity):
   * Sedan
   * SUV
```

Similarly, if we try to access a class that we skipped, we get the same error.

```
>>> register['SportsCar'] # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
  ...
NotInRegisterError: SportsCar is not a valid name for a car.
Available cars are (in decreasing similarity):
   * Sedan
   * SUV
```

When we iterate over the register, we iterate over the class names

```
>>> for car in register:
...     print(car)
Sedan
SUV
```

We can also iterate over the register using dictionary-style methods

```
>>> for car, Car in register.items():
...     print(car, Car)
Sedan <class 'subclass_register.subclass_register.Sedan'>
SUV <class 'subclass_register.subclass_register.SUV'>
>>> for Car in register.keys():
...     print(Car)
Sedan
SUV
>>> for Car in register.values():
...     print(Car)
<class 'subclass_register.subclass_register.Sedan'>
<class 'subclass_register.subclass_register.SUV'>
```

`__init__`(*class_type='class'*)
Initiate a class register.

Parameters **`class_type`** (`str`) – The name of the classes we register, e.g. layer or model if used for neural networks. It is used for pretty error messages.

`link_base`(*cls*)
Link a base class to the register. Can be used as a decorator.

`skip`(*cls*)
Decorator used to signal that the class shouldn't be added to the register.

`available_classes`
Tuple of the classes in the register.

Type tuple[str]

`linked`
Whether the register is linked to a base class or not.

Type bool

`items`()
Iterate over class names and classes.

`values`()
Iterate over classes (not names)

`keys`()
Iterate over class names

`__contains__`(*class_name*)
Check if a class name is in the register.

`__iter__`()
Iterate over class names.

`__getitem__`(*class_name*)
Get a class from the register.

`__setitem__`(*name*, *class_name*)
Add a new class to the register. It is impossible to change existing classes.

`__delitem__`(*class_name*)

Delete a class from the register.

Index