dataclass-builder Documentation

Release 1.2.0

Michael R. Shannon

Aug 21, 2019

CONTENTS

CHAPTER

ONE

USER GUIDE

Work in progress.

CHAPTER

TWO

API DOCUMENTATION

If you are looking for information on a specific function, class, or method, this part of the documentation is for you, in particular the *Public API*.

2.1 Public API

Work in progress, see the README for now.

CHAPTER

THREE

CONTRIBUTOR GUIDE

3.1 Private API

The documentation for the private API is automatically generated by *sphinx-apidoc* and is only to be used for debug and development purposes. None of the features documented here are intended for the end user. Only features documented in the *Public API* are considered stable and suitable for use outside of **dataclass-builder**.

3.1.1 dataclass_builder package

Submodules

dataclass_builder.__version__ module

Project information, specifically the version.

dataclass_builder._common module

Common utilities.

dataclass_builder.exceptions module

Exceptions for the package.

exception dataclass_builder.exceptions.**DataclassBuilderError**

Bases: `Exception`

Base class of errors raised by `DataclassBuilder`.

exception dataclass_builder.exceptions.**UndefinedFieldError**(*message: str*, *dataclass: Any*, *field: str*)

Bases: *dataclass_builder.exceptions.DataclassBuilderError*

Exception thrown when attempting to assign to an invalid field.

Parameters

- **message** – Human readable error message
- **dataclass** – `dataclasses.dataclass()` the `DataclassBuilder` was made for.
- **field** – Name of the invalid field that the calling code tried to assign to.

dataclass
dataclasses.dataclass() the DataclassBuilder was made for.

field
Name of the invalid field that the calling code tried to assign to.

exception dataclass_builder.exceptions.**MissingFieldError**(*message: str, dataclass: Any, field: Field[Any]*)
Bases: *dataclass_builder.exceptions.DataclassBuilderError*

Thrown when fields are missing when building a dataclasses.dataclass().

Parameters

- **message** – Human readable error message
- **dataclass** – dataclasses.dataclass() the DataclassBuilder was made for.
- **field** – The dataclasses.Field representing the missing field that needs to be assigned.

dataclass
dataclasses.dataclass() the DataclassBuilder was made for.

field
The dataclasses.Field representing the missing field that needs to be assigned.

dataclass_builder.factory module

Create dataclasses.dataclass() builders for specific dataclasses.

This module uses a factory to build builder classes that build a specific dataclass. These builder classes implement the builder pattern and allow constructing dataclasses over a period of time instead of all at once.

Examples

Using specialized builders allows for better documentation than the DataclassBuilder wrapper and allows for type checking because annotations are dynamically generated.

```
from dataclasses import dataclass
from dataclass_builder import (dataclass_builder, build, fields,
                               REQUIRED, OPTIONAL)

@dataclass
class Point:
    x: float
    y: float
    w: float = 1.0

PointBuilder = dataclass_builder(Point)
```

Now we can build a point.

```
>>> builder = PointBuilder()
>>> builder.x = 5.8
>>> builder.y = 8.1
>>> builder.w = 2.0
```

(continues on next page)

(continued from previous page)

```
>>> build(builder)
Point(x=5.8, y=8.1, w=2.0)
```

As long as the dataclass the builder was constructed for does not have a *build* field then a *build* method will be generated as well.

```
>>> builder.build()
Point(x=5.8, y=8.1, w=2.0)
```

Field values can also be provided in the constructor.

```
>>> builder = PointBuilder(x=5.8, w=100)
>>> builder.y = 8.1
>>> builder.build()
Point(x=5.8, y=8.1, w=100)
```

Note: Positional arguments are not allowed.

Fields with default values in the dataclass are optional in the builder.

```
>>> builder = PointBuilder()
>>> builder.x = 5.8
>>> builder.y = 8.1
>>> builder.build()
Point(x=5.8, y=8.1, w=1.0)
```

Fields that don't have default values in the dataclass are not optional.

```
>>> builder = PointBuilder()
>>> builder.y = 8.1
>>> builder.build()
Traceback (most recent call last):
...
MissingFieldError: field 'x' of dataclass 'Point' is not optional
```

Fields not defined in the dataclass cannot be set in the builder.

```
>>> builder.z = 3.0
Traceback (most recent call last):
...
UndefinedFieldError: dataclass 'Point' does not define field 'z'
```

Note: No exception will be raised for fields beginning with an underscore as they are reserved for use by subclasses.

Accessing a field of the builder before it is set gives either the *REQUIRED* or *OPTIONAL* constant

```
>>> builder = PointBuilder()
>>> builder.x
REQUIRED
>>> builder.w
OPTIONAL
```

The *fields* method can be used to retrieve a dictionary of settable fields for the builder. This is a mapping of field names to `dataclasses.Field` objects from which extra data can be retrieved such as the type of the data stored in the field.

```
>>> list(builder.fields().keys())
['x', 'y', 'w']
>>> [f.type.__name__ for f in builder.fields().values()]
['float', 'float', 'float']
```

A subset of the fields can be also be retrieved, for instance, to only get required fields:

```
>>> list(builder.fields(optional=False).keys())
['x', 'y']
```

or only the optional fields.

```
>>> list(builder.fields(required=False).keys())
['w']
```

Note: If the underlying dataclass has a field named *fields* this method will not be generated and instead the `fields()` function should be used instead.

dataclass_builder.factory.**dataclass_builder**(*dataclass: Type[Any], *, name: Optional[str] = None*) → Type[Any]

Create a new builder class specialized to a given dataclass.

Parameters

- **dataclass** – The `dataclasses.dataclass()` to create the builder for.
- **name** – Override the name of the builder, by default it will be '<dataclass>Builder' where <dataclass> is replaced by the name of the dataclass.

Return object A new dataclass builder class that is specialized to the given *dataclass*. If the given `dataclasses.dataclass()` does not contain the fields *build* or *fields* these will be exposed as public methods with the same signature as the *dataclass_builder.utility.build()* and *dataclass_builder.utility.fields()* functions respectively.

Raises **TypeError** – If *dataclass* is not a `dataclasses.dataclass()`. This is decided via `dataclasses.is_dataclass()`.

dataclass_builder.utility module

Utility functions for the package.

dataclass_builder.utility.**build**(*builder: dataclass_builder.wrapper.DataclassBuilder*) → Any

Use the given `DataclassBuilder` to initialize a *dataclass*.

This will use the values assigned to the given *builder* to construct a `dataclasses.dataclass()` of the type the *builder* was created for.

Note: This is not a method of `DataclassBuilder` in order to not interfere with possible field names. This function will use special private methods of `DataclassBuilder` which are excepted from field assignment.

Parameters **builder** – The dataclass builder to build from.

Raises ***dataclass_builder.exceptions.MissingFieldError*** – If not all of the required fields have been assigned to this builder.

dataclass_builder.utility.**fields**(*builder: dataclass_builder.wrapper.DataclassBuilder*, *, *required: bool = True*, *optional: bool = True*) → Mapping[str, Field[Any]]

Get a dictionary of the given `DataclassBuilder`'s fields.

Note: This is not a method of `DataclassBuilder` in order to not interfere with possible field names. This function will use special private methods of `DataclassBuilder` which are excepted from field assignment.

Parameters

- **builder** – The dataclass builder to get the fields for.
- **required** – Set to False to not report required fields.
- **optional** – Set to False to not report optional fields.

Returns A mapping from field names to actual `dataclasses.Field`'s in the same order as the *builder*'s underlying `dataclasses.dataclass()`.

dataclass_builder.utility.**update**(*dataclass: Any*, *builder: dataclass_builder.wrapper.DataclassBuilder*) → None

Update a dataclass or dataclass builder from a partial dataclass builder.

Parameters

- **dataclass** – :func‘dataclasses.dataclass‘ or dataclass builder to update.

  Note: Technically this can be any object that supports `__setattr__()`.

- **builder** – The datalcass builder to update *dataclass* with. All fields that are not missing in the *builder* will be set (overridden) on the given *dataclass*.

dataclass_builder.wrapper module

Create instances of `dataclasses.dataclass()` with the builder pattern.

This module uses a generic wrapper that becomes specialized at initialization into a builder instance that can build a given dataclass.

Examples

Using a builder instance is the fastest way to get started with the *dataclass-builder* package.

```
from dataclasses import dataclass
from dataclass_builder import (DataclassBuilder, build, fields,
                               REQUIRED, OPTIONAL)

@dataclass
class Point:
    x: float
```

(continues on next page)

(continued from previous page)

```
    y: float
    w: float = 1.0
```

Now we can build a point.

```
>>> builder = DataclassBuilder(Point)
>>> builder.x = 5.8
>>> builder.y = 8.1
>>> builder.w = 2.0
>>> build(builder)
Point(x=5.8, y=8.1, w=2.0)
```

Field values can also be provided in the constructor.

```
>>> builder = DataclassBuilder(Point, x=5.8, w=100)
>>> builder.y = 8.1
>>> build(builder)
Point(x=5.8, y=8.1, w=100)
```

Note: Positional arguments are not allowed, except for the dataclass itself.

Fields with default values in the dataclass are optional in the builder.

```
>>> builder = DataclassBuilder(Point)
>>> builder.x = 5.8
>>> builder.y = 8.1
>>> build(builder)
Point(x=5.8, y=8.1, w=1.0)
```

Fields that don't have default values in the dataclass are not optional.

```
>>> builder = DataclassBuilder(Point)
>>> builder.y = 8.1
>>> build(builder)
Traceback (most recent call last):
...
MissingFieldError: field 'x' of dataclass 'Point' is not optional
```

Fields not defined in the dataclass cannot be set in the builder.

```
>>> builder.z = 3.0
Traceback (most recent call last):
...
UndefinedFieldError: dataclass 'Point' does not define field 'z'
```

Note: No exception will be raised for fields beginning with an underscore as they are reserved for use by subclasses.

Accessing a field of the builder before it is set gives either the *REQUIRED* or *OPTIONAL* constant

```
>>> builder = DataclassBuilder(Point)
>>> builder.x
REQUIRED
```

(continues on next page)

(continued from previous page)

```
>>> builder.w
OPTIONAL
```

The `fields()` function can be used to retrieve a dictionary of settable fields for the builder. This is a mapping of field names to `dataclasses.Field` objects from which extra data can be retrieved such as the type of the data stored in the field.

```
>>> list(fields(builder).keys())
['x', 'y', 'w']
>>> [f.type.__name__ for f in fields(builder).values()]
['float', 'float', 'float']
```

A subset of the fields can be also be retrieved, for instance, to only get required fields:

```
>>> list(fields(builder, optional=False).keys())
['x', 'y']
```

or only the optional fields.

```
>>> list(fields(builder, required=False).keys())
['w']
```

class dataclass_builder.wrapper.**DataclassBuilder**(*dataclass: Any*, ***kwargs: Any*)

Bases: `object`

Wrap a dataclass with an object implementing the builder pattern.

This class, via wrapping, allows dataclasses to be constructed with the builder pattern. Once an instance is constructed simply assign to it's attributes, which are identical to the dataclass it was constructed with. When done use the *`dataclass_builder.utility.build()`* function to attempt to build the underlying dataclass.

> **Warning:** Because this class overrides attribute assignment when extending it care must be taken to only use private or "dunder" attributes and methods.

Parameters

- **dataclass** – The dataclass_that should be built by the builder instance
- ****kwargs** – Optionally initialize fields during initialization of the builder. These can be changed later and will raise UndefinedFieldError if they are not part of the *dataclass*'s *__init__* method.

Raises

- **TypeError** – If *dataclass* is not a dataclass. This is decided via `dataclasses.is_dataclass()`.
- ***dataclass_builder.exceptions.UndefinedFieldError*** – If you try to assign to a field that is not part of the *dataclass*'s *__init__*.
- ***dataclass_builder.exceptions.MissingFieldError*** – If `build()` is called on this builder before all non default fields of the *dataclass* are assigned.

__setattr__(*item: str*, *value: Any*) → None

Set a field value, or an object attribute if it is private.

Note: This will pass through all attributes beginning with an underscore. If this is a valid field of the dataclass it will still be built correctly but UndefinedFieldError will not be thrown for attributes beginning with an underscore.

If you need the exception to be thrown then set the field in the constructor.

Parameters

- **item** – Name of the dataclass field or private/"dunder" attribute to set.
- **value** – Value to assign to the dataclass field or private/"dunder" attribute.

Raises *dataclass_builder.exceptions.UndefinedFieldError* – If *item* is not initialisable in the underlying dataclass. If *item* is private (begins with an underscore) or is a "dunder" then this exception will not be raised.

__repr__() → str

Print a representation of the builder.

```
from dataclasses import dataclass
from dataclass_builder import DataclassBuilder, build, fields

@dataclass
class Point:
    x: float
    y: float
    w: float = 1.0
```

```
>>> DataclassBuilder(Point, x=4.0, w=2.0)
DataclassBuilder(Point, x=4.0, w=2.0)
```

Returns String representation that can be used to construct this builder instance.

Module contents

Create instances of dataclasses with the builder pattern.

exception dataclass_builder.**DataclassBuilderError**

Bases: Exception

Base class of errors raised by *DataclassBuilder*.

exception dataclass_builder.**MissingFieldError**(*message: str, dataclass: Any, field: Field[Any]*)

Bases: *dataclass_builder.exceptions.DataclassBuilderError*

Thrown when fields are missing when building a dataclasses.dataclass().

Parameters

- **message** – Human readable error message
- **dataclass** – dataclasses.dataclass() the *DataclassBuilder* was made for.
- **field** – The dataclasses.Field representing the missing field that needs to be assigned.

dataclass
dataclasses.dataclass() the *DataclassBuilder* was made for.

field
The dataclasses.Field representing the missing field that needs to be assigned.

exception dataclass_builder.**UndefinedFieldError**(*message: str*, *dataclass: Any*, *field: str*)
Bases: *dataclass_builder.exceptions.DataclassBuilderError*

Exception thrown when attempting to assign to an invalid field.

Parameters

- **message** – Human readable error message
- **dataclass** – dataclasses.dataclass() the *DataclassBuilder* was made for.
- **field** – Name of the invalid field that the calling code tried to assign to.

dataclass
dataclasses.dataclass() the *DataclassBuilder* was made for.

field
Name of the invalid field that the calling code tried to assign to.

dataclass_builder.**dataclass_builder**(*dataclass: Type[Any], *, name: Optional[str] = None*) → Type[Any]
Create a new builder class specialized to a given dataclass.

Parameters

- **dataclass** – The dataclasses.dataclass() to create the builder for.
- **name** – Override the name of the builder, by default it will be '<dataclass>Builder' where <dataclass> is replaced by the name of the dataclass.

Return object A new dataclass builder class that is specialized to the given *dataclass*. If the given dataclasses.dataclass() does not contain the fields *build* or *fields* these will be exposed as public methods with the same signature as the *dataclass_builder.utility.build()* and *dataclass_builder.utility.fields()* functions respectively.

Raises **TypeError** – If *dataclass* is not a dataclasses.dataclass(). This is decided via dataclasses.is_dataclass().

dataclass_builder.**build**(*builder: dataclass_builder.wrapper.DataclassBuilder*) → Any
Use the given *DataclassBuilder* to initialize a *dataclass*.

This will use the values assigned to the given *builder* to construct a dataclasses.dataclass() of the type the *builder* was created for.

Note: This is not a method of *DataclassBuilder* in order to not interfere with possible field names. This function will use special private methods of *DataclassBuilder* which are excepted from field assignment.

Parameters **builder** – The dataclass builder to build from.

Raises ***dataclass_builder.exceptions.MissingFieldError*** – If not all of the required fields have been assigned to this builder.

dataclass_builder.**fields**(*builder: dataclass_builder.wrapper.DataclassBuilder*, *, *required: bool = True*, *optional: bool = True*) → Mapping[str, Field[Any]]

Get a dictionary of the given *DataclassBuilder*'s fields.

Note: This is not a method of *DataclassBuilder* in order to not interfere with possible field names. This function will use special private methods of *DataclassBuilder* which are excepted from field assignment.

Parameters

- **builder** – The dataclass builder to get the fields for.
- **required** – Set to False to not report required fields.
- **optional** – Set to False to not report optional fields.

Returns A mapping from field names to actual dataclasses.Field's in the same order as the *builder*'s underlying dataclasses.dataclass().

dataclass_builder.**update**(*dataclass: Any*, *builder: dataclass_builder.wrapper.DataclassBuilder*) → None

Update a dataclass or dataclass builder from a partial dataclass builder.

Parameters

- **dataclass** – :func‘dataclasses.dataclass‘ or dataclass builder to update.

  Note: Technically this can be any object that supports __setattr__().

- **builder** – The datalcass builder to update *dataclass* with. All fields that are not missing in the *builder* will be set (overridden) on the given *dataclass*.

class dataclass_builder.**DataclassBuilder**(*dataclass: Any*, ***kwargs: Any*)

Bases: object

Wrap a dataclass with an object implementing the builder pattern.

This class, via wrapping, allows dataclasses to be constructed with the builder pattern. Once an instance is constructed simply assign to it's attributes, which are identical to the dataclass it was constructed with. When done use the *dataclass_builder.utility.build()* function to attempt to build the underlying dataclass.

Warning: Because this class overrides attribute assignment when extending it care must be taken to only use private or "dunder" attributes and methods.

Parameters

- **dataclass** – The dataclass_that should be built by the builder instance
- ****kwargs** – Optionally initialize fields during initialization of the builder. These can be changed later and will raise UndefinedFieldError if they are not part of the *dataclass*'s *__init__* method.

Raises

- **TypeError** – If *dataclass* is not a dataclass. This is decided via dataclasses.is_dataclass().

- ***dataclass_builder.exceptions.UndefinedFieldError*** – If you try to assign to a field that is not part of the *dataclass*'s *__init__*.
- ***dataclass_builder.exceptions.MissingFieldError*** – If *build()* is called on this builder before all non default fields of the *dataclass* are assigned.

__setattr__(*item: str*, *value: Any*) → None
Set a field value, or an object attribute if it is private.

Note: This will pass through all attributes beginning with an underscore. If this is a valid field of the dataclass it will still be built correctly but UndefinedFieldError will not be thrown for attributes beginning with an underscore.

If you need the exception to be thrown then set the field in the constructor.

Parameters

- **item** – Name of the dataclass field or private/"dunder" attribute to set.
- **value** – Value to assign to the dataclass field or private/"dunder" attribute.

Raises ***dataclass_builder.exceptions.UndefinedFieldError*** – If *item* is not initialisable in the underlying dataclass. If *item* is private (begins with an underscore) or is a "dunder" then this exception will not be raised.

__repr__() → str
Print a representation of the builder.

```
from dataclasses import dataclass
from dataclass_builder import DataclassBuilder, build, fields

@dataclass
class Point:
    x: float
    y: float
    w: float = 1.0
```

```
>>> DataclassBuilder(Point, x=4.0, w=2.0)
DataclassBuilder(Point, x=4.0, w=2.0)
```

Returns String representation that can be used to construct this builder instance.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

B

D

F

M

U