

---

# **UIClasses**

***Release 2.3.2***

**NewStore GmbH**

**Jul 02, 2020**



## CONTENTS:

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	This is not an ORM . . . . .	1
1.2	Bags of Variables . . . . .	1
1.3	How to install . . . . .	1
<b>2</b>	<b>Tutorial</b>	<b>3</b>
2.1	Declaring a Model for user interfaces . . . . .	3
2.1.1	Two ways of instantiating models . . . . .	3
2.1.1.1	1. Passing a dict . . . . .	3
2.1.1.2	1. Passing a keyword-arguments . . . . .	4
2.2	Automatic getters and setters . . . . .	4
2.2.1	Invisible Getters/Setters . . . . .	4
2.2.1.1	Read-only Getters . . . . .	4
2.2.1.2	Write-only Getters . . . . .	5
2.2.1.3	Read-write Properties . . . . .	6
<b>3</b>	<b>API Reference</b>	<b>7</b>
3.1	Model . . . . .	7
3.2	Model.List . . . . .	8
3.3	Model.Set . . . . .	8
3.4	DataBag . . . . .	8
3.5	UserFriendlyObject . . . . .	8
3.6	DataBagChild . . . . .	8
3.7	IterableCollection . . . . .	9
3.8	Utils . . . . .	9
3.9	File-System Helpers . . . . .	9
3.10	Meta . . . . .	9
<b>4</b>	<b>Release History</b>	<b>11</b>
4.1	Changes in 2.3.2 . . . . .	11
4.2	Changes in 2.3.1 . . . . .	11
4.3	Changes in 2.3.0 . . . . .	11
4.4	Changes in 2.2.1 . . . . .	11
4.5	Changes in 2.2.0 . . . . .	11
4.6	Changes in 2.1.0 . . . . .	12
4.7	Changes in 2.0.3 . . . . .	12
4.8	Changes in 2.0.2 . . . . .	12
4.9	Changes in 2.0.1 . . . . .	12
4.10	Changes in 2.0.0 . . . . .	12
4.11	Changes in 1.1.1 . . . . .	12

4.12 Changes in 1.1.0 . . . . .	12
<b>5 Indices and tables</b>	<b>13</b>
<b>Python Module Index</b>	<b>15</b>
<b>Index</b>	<b>17</b>

## INTRODUCTION

UIClasses is a library that leverages data-modeling for the user-interface layer of your python project.

It provides the *Model* class that adds *Data Classe* features to its subclasses.

With UIClasses you can separate backend from frontend data modeling, preventing coupling in your python project.

### 1.1 This is not an ORM

UIClasses is designed to work in tandem with your favorite ORM, not a replacement for it.

Java has DAO (Data-access objects) and POJO (Plain Old Java Objects), usually POJOs are used at the UI-layer of an application with data mapped from a DAO whose data came from the storage layer of an application.

In this context, UIClasses are POJOs, and ORM Models, for example SQLAlchemy, are DAOs.

### 1.2 Bags of Variables

- Objects optimized for user interfaces.
- Methods to traverse nested dicts, convert to and from json
- ModelList and ModelSet collections for robust manipulation of collections of models.
- No I/O happens in models.
- Collections can be easily cached to leverage responsive user interfaces.

### 1.3 How to install

```
pip3 install uiclasses
```



## TUTORIAL

This is a basic guide to the most basic usage of the module.

In this guide we will define data models for data returned by the Github API to retrieve users <https://developer.github.com/v3/users/> and present this data to the UI layer in a concise way.

At the end we will also build a very simple client to the Github API.

### 2.1 Declaring a Model for user interfaces

The model below is defined according to the [Single User Response](#) from the Github V3 API.

Take a look here to see what a full json response [looks like](#) before continuing so the model definition below will make more sense.

Okay, so you see there are several fields, but only a few of the User properties are relevant for user-interface purposes.

```
from uiclasses import Model

class GithubUser(Model):
    login: str
    email: str
    hireable: bool
    public_repos: int
    public_gists: int
    followers: int
    following: int
```

Every field declared with [type annotations](#) is considered to be visible in the user interface.

This is powered by `dataclasses.fields()`.

#### 2.1.1 Two ways of instantiating models

##### 2.1.1.1 1. Passing a dict

```
octocat = GithubUser({
    "login": "octocat",
})

print(octocat.to_dict())
```

```
{
    "login": "octocat",
}
```

### 2.1.1.2 1. Passing a keyword-arguments

```
octocat = GithubUser(
    login="octocat",
)
print(octocat.to_dict())
```

```
{
    "login": "octocat",
}
```

## 2.2 Automatic getters and setters

Every visible field becomes a property that can be accessed directly via instance as if it were a regular `@property`

```
user1 = GithubUser()
user1.login = "octocat"

print(user1.to_dict())
```

```
{
    "login": "octocat",
}
```

### 2.2.1 Invisible Getters/Setters

Sometimes it can be useful to define properties that act on the internal data of a model without making them visible to the user interface.

UIClasses provides special annotations to achieve this with 3 variations:

- Read-only Getters
- Write-only Setters
- Read-Write Properties

#### 2.2.1.1 Read-only Getters

```
from uiclasses import Model

class User(Model):
    id: int
    username: str
    token: Getter[str]
```

(continues on next page)



(continued from previous page)

```

foobar = User(id=1, username="foobar", token="some-data")
print(foobar.to_dict())
print(foobar.token)
print(foobar.get_table_columns())

try:
    foobar.token = 'another-value'
except Exception as e:
    print(e)

```

```

{
  "id": 1,
  "username": "foobar",
  "token": "some-data",
}
"some-data"
["id", "username"]
"'User' object has no attribute 'token'"

```

### 2.2.1.2 Write-only Getters

```

from uiclasses import Model

class User(Model):
    id: int
    username: str
    token: Setter[str]

foobar = User(id=1, username="foobar", token="some-data")
print(foobar.to_dict())
foobar.token = 'another-value'
print(foobar.to_dict())
print(foobar.get_table_columns())

try:
    print(foobar.token)
except Exception as e:
    print(e)

```

```

{
  "id": 1,
  "username": "foobar",
  "token": "some-data",
}
{
  "id": 1,
  "username": "foobar",
  "token": "another-value",
}
["id", "username"]

```

(continues on next page)

(continued from previous page)

```
"'User' object has no attribute 'token'"
```

### 2.2.1.3 Read-write Properties

```
from uiclasses import Model

class User(Model):
    id: int
    username: str
    token: Property[str]

foobar = User(id=1, username="foobar", token="some-data")
print(foobar.token)
print(foobar.to_dict())
foobar.token = 'another-value'
print(foobar.token)
print(foobar.to_dict())
print(foobar.get_table_columns())
```

```
"some-data"
{
    "id": 1,
    "username": "foobar",
    "token": "some-data",
}
"another-value"
{
    "id": 1,
    "username": "foobar",
    "token": "another-value",
}
["id", "username"]
```

## API REFERENCE

### 3.1 Model

```
>>> from uiclasses import Model
>>>
>>> class User(Model):
...     email: str
... 
```

**class** uiclasses.**Model** (*\_\_data\_\_*: *dict* = *None*, \**args*, \*\**kw*)

Base class for User-interface classes.

Allows declaring what instance attributes are visible via type annotations or `__visible_attributes__`.

Example:

```
from uiclasses.base import Model

class BlogPost(Model):
    id: int
    title: str

post = BlogPost(
    id=1,
    title='test',
    body='lorem ipsum dolor sit amet'
)

print(str(post))
print(repr(post))
print(post.format_robust_table())
```

## 3.2 Model.List

```
>>> user1 = User(email="aaaa@test.com")
>>> user2 = User(email="bbbb@test.com")
>>>
>>> users = User.List([user1, user2, user1])
>>> users
User.List[user1, user2, user3]
```

**class** uiclasses.collections.**ModelList** (*children: Iterable[uiclasses.base.Model]*)  
Implementation of *IterableCollection* for the `list` type.

## 3.3 Model.Set

An ordered set for managing unique items.

```
>>> user1 = User(email="aaaa@test.com")
>>> user2 = User(email="bbbb@test.com")
>>>
>>> users = User.Set([user1, user2, user1])
>>> users
User.Set[user1, user2]
```

**class** uiclasses.collections.**ModelSet** (*\*args, \*\*kws*)  
Implementation of *IterableCollection* for the `OrderedSet` type.

## 3.4 DataBag

**class** uiclasses.**DataBag** (*\_\_data\_\_: dict = None*)  
base-class for config containers, behaves like a dictionary but is a enhanced proxy to manage data from its internal dict `__data__` as well as traversing nested dictionaries within it.

## 3.5 UserFriendlyObject

**class** uiclasses.**UserFriendlyObject**

## 3.6 DataBagChild

**class** uiclasses.**DataBagChild** (*data, \*location*)  
Represents a nested dict within a `DataBag` that is aware of its location within the parent.

## 3.7 IterableCollection

**class** `uiclasses.collections.IterableCollection`

Base mixin for `ModelList` and `ModelSet`, provides methods to manipulate iterable collections in ways take advantage of the behavior of models.

For example it supports filtering by instance attributes through a call to the `attribute_matches()` method of each children.

**Features:**

- `sorted_by()` - sort by a single attribute
- `filter_by()` - to filter by a single attribute
- `sorted()` - alias to `MyModel.List(sorted(my_model_collection))` or `.Set()`
- `filter()` - alias to `MyModel.List(filter(callback, my_model_collection))`
- `format_robust_table()`
- `format_pretty_table()`

## 3.8 Utils

## 3.9 File-System Helpers

runtime helper functions used for leveraging idiosyncrasies of testing.

## 3.10 Meta



## RELEASE HISTORY

### 4.1 Changes in 2.3.2

- Add same `only_visible` of `serialize()` to `to_dict()` and `serialize_field()`.
- Pass correct `only_visible` within `serialize_all()` and `serialize_visible()`.

### 4.2 Changes in 2.3.1

- Change behavior of `serialize_visible()` to omit `None` values.

### 4.3 Changes in 2.3.0

- Add method `serialize_visible()` and `serialize_all()` to `Model` and `IterableCollection`.
- Add boolean parameter `only_visible` in `uiclasses.base.Model.serialize()` - to decide whether to call `serialize_visible()` or `serialize_all()`.

### 4.4 Changes in 2.2.1

- Add behavior to `uiclasses.collections.is_iterable()` consider anything with a callable `__iter__` attribute a callable.

### 4.5 Changes in 2.2.0

- Change behavior of explicit `__visible_attributes__` declaration: when declared, the visible fields will be exactly those. If not declared, visible fields will be extracted from type annotations.
- The old behavior of `__visible_attributes__` is now available through `Model.__declared_attributes__` which `__visible_attributes__` (if any) with types from annotations.
- Support casting `IterableCollection` with itself and introduce `uiclasses.collections.is_iterable()` helper function.
- Show `RuntimeWarning` if typing module is installed as distribution package in python > 3.6.1.

## 4.6 Changes in 2.1.0

- Support nested model types.
- Cast values to their known type when instantiating a new `Model`.

## 4.7 Changes in 2.0.3

- perform `super().__setattr__` behavior even when an explicit setter is not defined and the attribute does not exist in the instance.

## 4.8 Changes in 2.0.2

- fix python 3.6 support.

## 4.9 Changes in 2.0.1

- don't try to cast annotations containing `typing.Generic` or `typing.Any`.

## 4.10 Changes in 2.0.0

- support explicit declaration of getters and setters that are not visible properties.
- implement type casting for all model attributes.
- automatic parsing of boolean-looking strings for fields of type `bool`.

## 4.11 Changes in 1.1.1

- Allow `Model(x)` where `x` is not a dict but can be cast into a dict.

## 4.12 Changes in 1.1.0

- `Model.Set()` and `Model.List()` not support generators.



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### U

`uiclasses.fs`, 9  
`uiclasses.meta`, 9  
`uiclasses.utils`, 9



## INDEX

### D

DataBag (*class in uiclasses*), 8

DataBagChild (*class in uiclasses*), 8

### I

IterableCollection (class in *uiclasses.collections*), 9

### M

Model (*class in uiclasses*), 7

ModelList (*class in uiclasses.collections*), 8

ModelSet (*class in uiclasses.collections*), 8

module

    uiclasses.fs, 9

    uiclasses.meta, 9

    uiclasses.utils, 9

### U

uiclasses.fs

    module, 9

uiclasses.meta

    module, 9

uiclasses.utils

    module, 9

UserFriendlyObject (*class in uiclasses*), 8