figgis Documentation

Release 1.8.1

Scott Kruger

Aug 01, 2017

Contents

1 Usage **3**
1.1 Introduction 3
1.2 API 4

2 Indices and tables **7**

Python Module Index **9**

Contents:

CHAPTER 1

Usage

Introduction

Using *figgis* involves defining one or more subclass of *Config*, each containing one or more *Field* or *ListField*. These configs may be used to consume json-like data structures (i.e. a dictionary containing only primitive data types, lists, and other dictionaries. Assuming that all of your data is valid, the result will be an object with attributes that match the fields that you defined and have the appropriate type.

By default, fields are strings. However, you may use any data type that you like, including arbitrary functions:

```
>>> from datetime import datetime

>>> class Event(Config):
...
...     def parse_date(date_string):
...         return datetime.strptime(date_string, '%Y-%m-%d')
...
...     name = Field(required=True)
...     date = Field(parse_date, required=True)

>>> birthday = Event(name="Jane's Birthday", date='1985-08-29')
>>> birthday.date
datetime.datetime(1985, 8, 29, 1, 0, 0)
```

Fields may also use another *Config*, so that you can make arbitrarily-nested data structures:

```
>>> class Calendar(Class):
...
...     events = ListField(Event)


>>> this_year = Calendar(events=[
...     {'name': 'My birthday', 'date': '2015-03-30'},
...     {'name': 'Your birthday', 'date': '2015-07-11'},
... ])
```

```
>>> this_year.events[1].name
'Your birthday'
```

figgis will throw an exception when your data does not match the definition:

```
>>> Calendar(events=[
    {'date': '2014-01-01'},
])
figgis.PropertyError: 'Missing property: events.0.name'
```

In addition to ensuring that all fields exist (if required) and are of the proper type, *figgis* can check your data against any number of arbitrary validation functions:

```
>>> class Person(Config):
...     name = Field()
...     age = Field(int, validator=lambda value: value > 0)

>>> Person(name='John', age=-1)
figgis.ValidationError: Field 'age' is invalid
```

You may also make your validators throw more useful error messages:

```
>>> from figgis import ValidationError

>>> class Person(Config):
...     def validate_age(age):
...         if age < 0:
...             raise ValidationError('Should be a non-negative integer')
...         return True
...
...     name = Field()
...     age = Field(int, validator=validate_age)

>>> Person(name='John', age=-1)
figgis.ValidationError: Field 'age' is invalid: Should be a non-negative integer
```

Sometimes, you may have data that has keys that can not be used as python variable names. In this case, you can use the *key* argument to perform a translation:

```
>>> class WeirdData(Config):
...     irregular = Field(key='123-this is a bad field')

>>> WeirdData({'123-this is a bad field': 'mydata'}).irregular
'mydata'
```

API

class figgis.**Config**(**args*, ***kwargs*)

Base class for configuration objects. See *Field* and *ListField* for usage.

When you create a *Config* instance, you pass to it parameters as you would a *dict*. Similar to a *dict*, you can either use keyword arguments or pass in an actual *dict* object.

If you wish to inherit from another *Config*, do not do it in the traditional class inheritance sense. Instead, specify any parents using the *__inherits__* attribute:

```
>>> class Parent(Config):
...     id = Field(int, required=True)
>>> class Child(Config):
...     __inherits__ = [Parent]
...
...     name = Field()
```

classmethod describe()
Return a pretty-formatted string that describes the format of the data

to_dict()
Convert the config to a plain python dictionary

class figgis.**Field**(**types*, ***kwargs*)
Represents a typed field in a configuration file. *type* may be a python type (e.g. *int*, *str*, *float*, *dict*), another *Config* object, or a function/lambda that takes a single argument and returns the desired parsed value.

When a *Config* is instantiated, each field is type-checked against the corresponding value. You may provide additional validation by passing a function to *validator*. This function takes one argument, the config value, and must return a *boolean* value.

```
>>> class Address(Config):
...     number = Field(int, required=True)
...     street = Field(required=True)
...     suffix = Field(default='St.', choices=['St.', 'Ave.'])  # optional
...
...     def __str__(self):
...         return '{0} {1} {2}'.format(
...             self.number, self.street, self.suffix)
>>> class Person(Config):
...     name = Field(required=True)
...     age = Field(int, required=True, validator=lambda age: age > 0)
...     address = Field(Address)
>>> joe = Person(
...     name='Joe',
...     age=45,
...     address=dict(
...         number=123,
...         street='Easy',
...     )
... )
>>> print('{0}, age {1}, lives at {2}'.format(
...     joe.name, joe.age, joe.address))
Joe, age 45, lives at 123 Easy St.
```

Sometimes, you may need to consume data for which some keys can not be expressed as a python identifier, e.g. *@attr*. In this case, you may use the *key* option to provide a mapping from the data to your fields:

```
>>> class Translated(Config):
...     crazy_key = Field(required=True, key='@crazy_key')
>>> Translated({'@crazy_key@': 'value'}).crazy_key
'value'
```

__init__(**types*, ***kwargs*)
Field(*types, type=None, required=False, default=NotSpecified, validator=None, choices=None, help=None, hidden=None, key=None, nullable=True)

Parameters

- **types** – One or more types or functions to apply, in order, to the field value.
- **type** – Either a built-in data type, another *Config* object, or a function that takes the raw field value and produces the desired result. May not be used with *types*
- **required** – If *True*, throw an exception if the data does not exist
- **default** – Default value to use if the data does not exist
- **nullable** – If *True*, then allow the field value to be null
- **choices** – List of values that constrain the possible data values
- **key** – The key in the data that should be read to produce this field (defaults to the variable name to which this field is assigned)
- **validator** – Either function or a list of functions, taking the parsed field data, that either returns *False* or throws `ValidationError` if the data is invalid, or returns *True* otherwise
- **hidden** – Hide this field from the output of *Config.describe()*

class `figgis.`**ListField**(**types*, ***kwargs*)

Similar to *Field*, except that it expects a list of objects instead of a single object.

```
>>> class Product(Config):
...     name = Field(required=True)
...     price = Field(float, default=0.0)
>>> class Catalog(Config):
...     products = ListField(Product, required=True)
>>> catalog = Catalog(
...     products=[
...         {'name': 'Orange', 'price': 0.79},
...         {'name': 'Apple', 'price': 0.59},
...     ]
... )
>>> for product in catalog.products:
...     print('{0} costs {1}'.format(product.name, product.price))
Orange costs 0.79
Apple costs 0.59
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

figgis, 3

Index

Symbols

__init__() (figgis.Field method), 5

C

Config (class in figgis), 4

D

describe() (figgis.Config class method), 5

F

Field (class in figgis), 5
figgis (module), 3

L

ListField (class in figgis), 6

T

to_dict() (figgis.Config method), 5