File Config Documentation

Release 1.0.0

Stephen Bunn

Oct 11, 2019

CONTENTS

Easily define configs by creating a *config()* decorated class...

```
from typing import List, Dict
import file_config


@file_config.config
class ProjectConfig(object):

   @file_config.config
   class Dependency(object):
      name = file_config.var(str, min=1)
      version = file_config.var(file_config.Regex(r"^v\d+$"))

   name = file_config.var(str, min=1)
   type_ = file_config.var(str, name="type", required=False)
   keywords = file_config.var(List[str], min=0, max=10)
   dependencies = file_config.var(Dict[str, Dependency])


config = ProjectConfig(
   name="My Project",
   type_="personal-project",
   keywords=["example", "test"],
   dependencies={
       "a-dependency": ProjectConfig.Dependency(name="A Dependency", version="v12")
   },
)
```

Configs can be dumped and loaded to various config formats...

```
>>> json_content = config.dumps_json()
>>> print(json_content)
{"name":"My Project","type":"personal-project","keywords":["example","test"],
↪"dependencies":{"a-dependency":{"name":"A Dependency","version":"v12"}}}
>>> new_config = ProjectConfig.loads_json(json_content)
ProjectConfig(name='My Project', type_='personal-project', keywords=['example', 'test
↪'], dependencies={'a-dependency': ProjectConfig.Dependency(name='A Dependency',␣
↪version='v12')})
>>> assert new_config == config
```

and validated...

```
>>> config.dependencies["a-dependency"].version = "12"
>>> file_config.validate(config)
Traceback (most recent call last):
  File "main.py", line 28, in <module>
    file_config.validate(config)
  File "/home/stephen-bunn/Git/file-config/file_config/_file_config.py", line 373, in␣
↪validate
    to_dict(instance, dict_type=dict), build_schema(instance.__class__)
  File "/home/stephen-bunn/.local/share/virtualenvs/file-config-zZO-gwXq/lib/python3.
↪6/site-packages/jsonschema/validators.py", line 861, in validate
    cls(schema, *args, **kwargs).validate(instance)
  File "/home/stephen-bunn/.local/share/virtualenvs/file-config-zZO-gwXq/lib/python3.
↪6/site-packages/jsonschema/validators.py", line 305, in validate
    raise error
jsonschema.exceptions.ValidationError: '12' does not match '^v\\d+$'
```

(continues on next page)

(continued from previous page)

```
Failed validating 'pattern' in schema['properties']['dependencies']['patternProperties
↪']['^(.*)$']['properties']['version']:
    {'$id': '#/properties/dependencies/properties/version',
     'pattern': '^v\\d+$',
     'type': 'string'}
On instance['dependencies']['a-dependency']['version']:
    '12'
```

To get started using this package, please see the *Getting Started* page!

CHAPTER

ONE

GETTING STARTED

Welcome to File Config!
This page should hopefully provide you with enough information to get you started defining, serializing, and validating config instances.

1.1 Installation and Setup

Installing the package should be super duper simple as we utilize Python's setuptools.

```
$ pipenv install file-config
$ # or if you're old school...
$ pip install file-config
```

Or you can build and install the package from the git repo.

```
$ git clone https://github.com/stephen-bunn/file-config.git
$ cd ./file-config
$ python setup.py install
```

1.2 Defining Configs

Similar to attrs, the most basic way to setup a new config is to use the *config* decorator.

```
@file_config.config
class ProjectConfig(object):
   pass
```

Creating an empty instance of a config class follows the same comparison rules as attrs.

```
>>> ProjectConfig()
ProjectConfig()
>>> ProjectConfig() == ProjectConfig()
True
>>> ProjectConfig() is ProjectConfig()
False
```

One of main features that *file_config* provides is the ability to generate a JSONSchema dictionary to use for validating the state of a config instance. You can get the schema by passing a config class to the `build_schema()` method.

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'properties': {},
 'required': [],
 'type': 'object'}
```

Currently the generated JSONSchema is pretty boring since the `ProjectConfig` class is totally empty. We can add a quick title and description to the root JSONSchema object by adding two arguments to the *`config`* decorator...

- `title` - Defines the title of the object in the generated JSONSchema
- `description` - Defines the description of the object in the generated JSONSchema

```
@file_config.config(
   title="Project Config",
   description="The project configuration for my project"
)
class ProjectConfig(object):
   pass
```

After building the schema again you can see the added `title` and `description` properties in the resulting JSONSchema dictionary.

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'description': 'The project configuration for my project',
 'properties': {},
 'required': [],
 'title': 'Project Config',
 'type': 'object'}
```

We can also control the `$id` and `$schema` properties within the generated JSONSchema but adding two more arguments to the *`config`* decorator...

- `schema_id` - Defines a custom id for the generated JSONSchema's `$id` property
- `schema_draft` - Defines a custom draft URL for the generated JSONSchema's `$schema` property

```
@file_config.config(
   schema_id="MyProjectSchema",
   schema_draft="http://json-schema.org/2019-09/schema#"
)
class ProjectConfig(object):
   pass
```

Building this schema will result in the following dictionary:

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'MyProjectSchema',
 '$schema': 'http://json-schema.org/2019-09/schema#',
 'properties': {},
 'required': [],
 'type': 'object'}
```

Important: We rely heavily on the feature specified in at least JSONSchema draft-07 for both union types and regex pattern matching types. If you provide an unsupported draft to `schema_draft` which might imply that internal

config validation will break, a `UserWarning` will be thrown.

In 99% of cases you shouldn't even specify a custom `schema_draft`.
Use this parameter at your own peril.

1.3 Config Vars

Now that you have an empty config class, you can start adding variables that should be part of the config. Adding config vars is simple, but the more constraints you have on your vars the more complex the definition of that var becomes.

You can start off with the most basic config var possible by using the *`var`* method.

```
@file_config.config
class ProjectConfig(object):
   name = file_config.var()
```

By default a config var...

- uses the name you assigned to it in the config class (in this case `name`)
- is `required` for validation

Checkout how the built JSONSchema looks now that you added a basic var.

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'properties': {'name': {'$id': '#/properties/name'}},
 'required': ['name'],
 'type': 'object'}
```

1.3.1 Required

You can make a config var "optional" by setting `required` to `False`.

```
@file_config.config
class ProjectConfig(object):
   name = file_config.var(required=False)
```

You'll notice that the `name` entry in the `required` list is now missing from the built JSONSchema.

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'properties': {'name': {'$id': '#/properties/name'}},
 'required': [],
 'type': 'object'}
```

1.3.2 Name

You can change the serialization name of the config var by setting `name` to some string. This is useful when you need to use Python keywords as attribute names in the config.

```
@file_config.config
class ProjectConfig(object):
   name = file_config.var()
   type_ = file_config.var(name="type")
```

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'properties': {'name': {'$id': '#/properties/name'},
                'type': {'$id': '#/properties/type'}},
 'required': ['name', 'type'],
 'type': 'object'}
```

Serialization dumps to/loads from the given `name` attribute.

```
>>> ProjectConfig(name="My Project", type_="config").dumps_json()
'{"name":"My Project","type":"config"}'
>>> ProjectConfig.loads_json('{"name":"My Project","type":"config"}')
ProjectConfig(name='My Project', type_='config')
```

1.3.3 Type

Defining a config var's type is straight forward but can be complex given your config requirements. A config var's type can either be passed in as the first argument or as the `type` kwarg to the *`var`* method.

Builtin Types

The *`var`* can take in any of the builtin Python types.

```
@file_config.config
class ProjectConfig(object):
   name = file_config.var(str)
   type_ = file_config.var(name="type", type=str)
```

This results in some extra rules being added to the properties in the built JSONSchema.

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'properties': {'name': {'$id': '#/properties/name', 'type': 'string'},
                'type': {'$id': '#/properties/type', 'type': 'string'}},
 'required': ['name', 'type'],
 'type': 'object'}
```

You'll notice now that both the `name` and `type` properties have a declared type of `string`. So when validating a `ProjectConfig` instance where `type_` is a string you get no errors...

```
>>> config = ProjectConfig(name='My Project', type_="config")
>>> print(config)
```

(continues on next page)

(continued from previous page)

```
ProjectConfig(name='My Project', type_="config")
>>> file_config.validate(config)
None
```

But if validating a `ProjectConfig` instance where `type_` is an integer, you'll get an error similar to the following...

```
>>> config.type_ = 0
>>> print(config)
ProjectConfig(name='My Project', type_=0)
>>> file_config.validate(config)
Traceback (most recent call last):
  File "main.py", line 82, in <module>
    file_config.validate(config)
  File "/home/stephen-bunn/Git/file-config/file_config/_file_config.py", line 355, in␣
↪validate
    to_dict(instance, dict_type=dict), build_schema(instance.__class__)
  File "/home/stephen-bunn/.local/share/virtualenvs/file-config-zZO-gwXq/lib/python3.
↪6/site-packages/jsonschema/validators.py", line 861, in validate
    cls(schema, *args, **kwargs).validate(instance)
  File "/home/stephen-bunn/.local/share/virtualenvs/file-config-zZO-gwXq/lib/python3.
↪6/site-packages/jsonschema/validators.py", line 305, in validate
    raise error
jsonschema.exceptions.ValidationError: 0 is not of type 'string'
Failed validating 'type' in schema['properties']['type']:
    {'$id': '#/properties/type', 'type': 'string'}
On instance['type']:
    0
```

Typing Types

The *var* can also use `typing` types as the `type` argument. This allows you to get a bit more specific with the exact format of the var type.

```
from typing import Set

@file_config.config
class ProjectConfig(object):
   name = file_config.var(str)
   versions = file_config.var(Set[str])
```

Using a fancy `typing` type like this will result in the following JSONSchema being built...

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'properties': {'name': {'$id': '#/properties/name', 'type': 'string'},
                'versions': {'$id': '#/properties/versions',
                             'items': {'$id': '#/properties/versions/items',
                                       'type': 'string'},
                             'type': 'array'}},
 'required': ['name', 'versions'],
 'type': 'object'}
```

You might notice that the `versions` var says to use `set()` as the loaded in type. However, you can't serialize set

out Python sets in many data formats such as JSON, but loading it back into a config instance can cast it back into a set.

```
>>> ProjectConfig.loads_json('{"name": "Testing", "versions": ["123", "123"]}')
ProjectConfig(name='Testing', versions={'123'})
```

Important: Using `typing` types requires a bit of intuition. Your defined var type must be JSON serializable.

```
from typing import Dict

@file_config.config
class ProjectConfig(object):
   name = file_config.var(str)
   depends = file_config.var(Dict[int, str])
```

Trying to build the schema with a non JSON serializable var type (`Dict[int, str]`) will throw an error similar to this...

```
>>> file_config.build_schema(ProjectConfig)
Traceback (most recent call last):
   File "main.py", line 83, in <module>
     file_config.build_schema(ProjectConfig)
   File "/home/stephen-bunn/Git/file-config/file_config/schema_builder.py", line 282,␣
↪in build_schema
     return _build_config(config_cls, property_path=[])
   File "/home/stephen-bunn/Git/file-config/file_config/schema_builder.py", line 265,␣
↪in _build_config
     var, property_path=property_path
   File "/home/stephen-bunn/Git/file-config/file_config/schema_builder.py", line 221,␣
↪in _build_var
     _build_type(var.type, var, property_path=property_path + [var_name])
   File "/home/stephen-bunn/Git/file-config/file_config/schema_builder.py", line 182,␣
↪in _build_type
     return builder(value, property_path=property_path)
   File "/home/stephen-bunn/Git/file-config/file_config/schema_builder.py", line 160,␣
↪in _build_object_type
     f"cannot serialize object with key of type {key_type!r}, "
ValueError: cannot serialize object with key of type <class 'int'>, located in var
↪'depends'
```

Nested Configs

You can also nest configs as types in your config classes.

```
from typing import List

@file_config.config
class ProjectConfig(object):

   @file_config.config
   class Dependency(object):
      name = file_config.var(str)
      version = file_config.var(str)
```

(continues on next page)

(continued from previous page)

```
name = file_config.var(str)
dependencies = file_config.var(List[Dependency])
```

Building the schema for this config will result in a format you might expect...

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'properties': {'dependencies': {'$id': '#/properties/dependencies',
                                 'items': {'$id': '#/properties/dependencies/items',
                                           'properties': {'name': {'$id': '#/
↪properties/dependencies/items/properties/name',
                                                                   'type': 'string'},
                                                          'version': {'$id': '#/
↪properties/dependencies/items/properties/version',
                                                                      'type': 'string
↪'}},
                                           'required': ['name', 'version'],
                                           'type': 'object'},
                                 'type': 'array'},
                'name': {'$id': '#/properties/name', 'type': 'string'}},
 'required': ['name', 'dependencies'],
 'type': 'object'}
```

Serialization and deserialization of an instance of this config builds instances of the config objects as you would expect...

```
>>> config = ProjectConfig(
...     name="My Project",
...     dependencies=[ProjectConfig.Dependency(name="A Dependency", version="1.2.3")],
... )
>>> config.dumps_json()
'{"name":"My Project","dependencies":[{"name":"A Dependency","version":"1.2.3"}]}'
>>> ProjectConfig.loads_json('{"name":"My Project","dependencies":[{"name":"A␣
↪Dependency","version":"1.2.3"}]}')
ProjectConfig(name='My Project', dependencies=[ProjectConfig.Dependency(name='A␣
↪Dependency', version='1.2.3')])
```

Regular Expressions

In some cases you might need to do string validation based on some regular expression. Since there is no decent builtin way to specify a pattern as a type you must use the custom `Regex` method to specify the regular expression to validate against.

```
@file_config.config
class ProjectConfig(object):
    name = file_config.var(str)
    version = file_config.var(file_config.Regex(r"^v\d+$"))
```

Generating the JSONSchema for this config results in the `pattern` property of the `version` config var to be populated with the appropriate regular expression.

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
```

(continues on next page)

(continued from previous page)

```
 'properties': {'name': {'$id': '#/properties/name', 'type': 'string'},
                'version': {'$id': '#/properties/version',
                            'pattern': '^v\\d+$',
                            'type': 'string'}},
 'required': ['name', 'version'],
 'type': 'object'}
```

Note: Using the `Regex` method uses `typing.NewType()` to generate a typing instance where the regex you supply is compiled by `re.compile()` and stored in the `__supertype__` attribute of the newly generated type.

This method **assumes** that the base type of the attribute is `string` (as you cannot do regex matching against any other type).

You can get pretty specific with your config validation by using regular expressions...

```
from typing import Dict

@file_config.config
class ProjectConfig(object):
   name = file_config.var(str)
   dependencies = file_config.var(Dict[str, file_config.Regex(r"^v\d+$")])
```

Here is what happens when you try to pass a value into the `dependencies` dictionary that doesn't match the provided regular expression...

```
>>> config = ProjectConfig(name="My Project", dependencies={"A Dependency": "12"})
>>> file_config.validate(config)
Traceback (most recent call last):
  File "main.py", line 88, in <module>
    print(file_config.validate(config))
  File "/home/stephen-bunn/Git/file-config/file_config/_file_config.py", line 363, in␣
↪validate
    to_dict(instance, dict_type=dict), build_schema(instance.__class__)
  File "/home/stephen-bunn/.local/share/virtualenvs/file-config-zZO-gwXq/lib/python3.
↪6/site-packages/jsonschema/validators.py", line 861, in validate
    cls(schema, *args, **kwargs).validate(instance)
  File "/home/stephen-bunn/.local/share/virtualenvs/file-config-zZO-gwXq/lib/python3.
↪6/site-packages/jsonschema/validators.py", line 305, in validate
    raise error
jsonschema.exceptions.ValidationError: '12' does not match '^v\\d+$'
Failed validating 'pattern' in schema['properties']['dependencies']['patternProperties
↪']['^(.*)$']:
    {'pattern': '^v\\d+$', 'type': 'string'}
On instance['dependencies']['A Dependency']:
    '12'
```

Important: The generated JSONSchema requires that the regular expression you give must be a full matching pattern (containing `^` and `$` or `\A` and `\Z`).

Without start and end terminators in the regular expression JSONSchema will not fully match the string and assume that it is correct.

1.3.4 Encoder and Decoder

Since not all types are supported by all serializers, you can specify a custom encoder callable and decoder callable. For example, `json` doesn't support the serialization of `datetime.datetime` instances. You can get around this by using the `encoder` and `decoder` arguments...

```
import datetime

@file_config.config
class ProjectConfig(object):
   updated = file_config.var(
      datetime.datetime,
      encoder=lambda x: x.timestamp(),
      decoder=datetime.datetime.fromtimestamp
   )
```

This is a simple solution to deal with `datetime.datetime` but works for all serializers.

Note: Trying to do validation on a variable with a type not supported by jsonschema (e.g. `datetime.datetime`) will always raise a `UserWarning` similar to the following...

```
>>> file_validate.validate(config)
/home/stephen-bunn/Git/file-config/file_config/schema_builder.py:195: UserWarning:␣
↪unhandled translation for type <class 'datetime.datetime'> with value␣
↪Attribute(name='updated', default=None, validator=None, repr=True, cmp=True,␣
↪hash=None, init=True, metadata=mappingproxy({'__file_config_metadata': _
↪ConfigEntry(type=<class 'datetime.datetime'>, default=None, name=None, title=None,␣
↪description=None, required=True, examples=None, encoder=<function ProjectConfig.
↪<lambda> at 0x7f7ef9c80d08>, decoder=<built-in method fromtimestamp of type object␣
↪at 0xa05540>, min=None, max=None, unique=None, contains=None)}), type=<class
↪'datetime.datetime'>, converter=None, kw_only=False)
warnings.warn(f"unhandled translation for type {type_!r} with value {value!r}")
```

This warning is raised whenever the `schema_builder` cannot handle the given type of a config var. The resulting JSONSchema will define the config var as a property but will **not** specify a type. Essentially, the built JSONSchema will just validate that the property exists as long as the `required` flag is set to `True`.

1.3.5 Extras

There are several other validation rules that *`var`* method exposes. These arguments are used only to add validation logic to the generated JSONSchema.

- `title` - *A title for the config var, used as the title of the JSONSchema property*
- `description` - *A description for the config var, used as the description of the JSONSchema property*
- `examples` - *A list of examples for what the config var might be*
- `min` - *A minimum value for the var (applies to numbers, strings, and arrays)*
- `max` - *A maximum value for the var (applies to numbers, strings, and arrays)*
- `unique` - *Indicates that the var must be unique (applies to arrays)*
- `contains` - *Indicates that the var must contain the given element (applies to arrays)*

If you try to use one of the rules on a *`var`* that doesn't actually take that rule into consideration, a `UserWarning` is raised when `build_schema()` is called...

```
@file_config.config
class ProjectConfig(object):
   name = file_config.var(str, unique=True)
```

```
>>> file_config.build_schema(ProjectConfig)
/home/stephen-bunn/Git/file-config/file_config/schema_builder.py:64: UserWarning:␣
↪field modifier 'unique' has no effect on var 'name' of type <class 'str'>
  f"field modifier {entry_attribute.name!r} has no effect on var "
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'properties': {'name': {'$id': '#/properties/name', 'type': 'string'}},
 'required': ['name'],
 'type': 'object'}
```

The appropriate schema (disregarding the unapplied `unique` rule on `name`) is still returned.

1.4 Validation

You've probably seen some examples of validation in the previous sections (as it relates pretty closely to how to declare config vars). Validation is done 100% through the use of dynamically generated JSONSchema based on the declarations of the `config`.

The method used to generate the JSONSchema is `build_schema()`. You can use this method by simply passing in a class wrapped by *`config`*...

For example take the following (pretty specific) config class...

```
from typing import List, Dict

@file_config.config
class ProjectConfig(object):

   @file_config.config
   class Dependency(object):
      name = file_config.var(str, min=1)
      version = file_config.var(file_config.Regex(r"^v\d+$"))

   name = file_config.var(str, min=1)
   type_ = file_config.var(str, name="type", required=False)
   keywords = file_config.var(List[str], min=0, max=10)
   dependencies = file_config.var(Dict[str, Dependency])
```

The resulting JSONSchema ends up being the following...

```
>>> file_config.build_schema(ProjectConfig)
{'$id': 'ProjectConfig.json',
 '$schema': 'http://json-schema.org/draft-07/schema#',
 'properties': {'dependencies': {'$id': '#/properties/dependencies',
                                 'patternProperties': {'^(.*)$': {'$id': '#/
↪properties/dependencies',
                                                                  'properties': {'name
↪': {'$id': '#/properties/dependencies/properties/name',
↪     'minLength': 1,
↪     'type': 'string'},
```

(continues on next page)

(continued from previous page)

```
↪'version': {'$id': '#/properties/dependencies/properties/version',
↪        'pattern': '^v\\d+$',
↪        'type': 'string'}},
                                                    'required': ['name',
↪'version'],
                                                    'type': 'object'}},
                                'type': 'object'},
               'keywords': {'$id': '#/properties/keywords',
                            'items': {'$id': '#/properties/keywords/items',
                                      'type': 'string'},
                            'maxItems': 10,
                            'minItems': 0,
                            'type': 'array'},
               'name': {'$id': '#/properties/name',
                        'minLength': 1,
                        'type': 'string'},
               'type': {'$id': '#/properties/type', 'type': 'string'}},
 'required': ['name', 'keywords', 'dependencies'],
 'type': 'object'}
```

Performing validation is very simple. All you need to do is pass an **instance** of the config into the *validate* method...

```
>>> config = ProjectConfig(
...     name="My Project",
...     type_="personal-project",
...     keywords=["example", "test"],
...     dependencies={
...         "a-dependency": ProjectConfig.Dependency(name="A Dependency", version="v12
↪")
...     },
... )
>>> file_config.validate(config)
None
```

The nice thing about JSONSchema is that it's pretty specific about what exactly is failing when checking an instance that is invalid. For example, what happens if we give an empty `name` in our config instance?

```
>>> config.name = ""
>>> file_config.validate(config)
Traceback (most recent call last):
  File "main.py", line 108, in <module>
    file_config.validate(config)
  File "/home/stephen-bunn/Git/file-config/file_config/_file_config.py", line 363, in␣
↪validate
    to_dict(instance, dict_type=dict), build_schema(instance.__class__)
  File "/home/stephen-bunn/.local/share/virtualenvs/file-config-zZO-gwXq/lib/python3.
↪6/site-packages/jsonschema/validators.py", line 861, in validate
    cls(schema, *args, **kwargs).validate(instance)
  File "/home/stephen-bunn/.local/share/virtualenvs/file-config-zZO-gwXq/lib/python3.
↪6/site-packages/jsonschema/validators.py", line 305, in validate
    raise error
```

(continues on next page)

(continued from previous page)

```
jsonschema.exceptions.ValidationError: '' is too short
Failed validating 'minLength' in schema['properties']['name']:
    {'$id': '#/properties/name', 'minLength': 1, 'type': 'string'}
On instance['name']:
    ''
```

Pretty explicit right? Since we use the `jsonschema` package to perform validation, it provides some really useful information in the exceptions raised from failed validations...

```
>>> try:
...     file_config.validate(config)
... except jsonschema.exceptions.ValidationError as exc:
...     print(exc.__dict__)
{'cause': None,
 'context': [],
 'instance': '',
 'message': "'' is too short",
 'parent': None,
 'path': deque(['name']),
 'relative_path': deque(['name']),
 'relative_schema_path': deque(['properties', 'name', 'minLength']),
 'schema': {'$id': '#/properties/name', 'minLength': 1, 'type': 'string'},
 'schema_path': deque(['properties', 'name', 'minLength']),
 'validator': 'minLength',
 'validator_value': 1}
```

This might help you inform your project what to look for to fix in a config.

Important: Validation is only applied before loading a new instance from some serialized content or when you explicitly ask it to validate through *`validate`*.

Validation is **not** done as setter methods for *`config`* wrapped classes. This means you can throw whatever data you want into a config instance and it will never yell at you until you either try to load it from some content or when you explicitly ask for validation to occur.

1.5 Dumping / Loading

The serialization / deserialization steps of *`config`* wrapped objects are built from the `collections.OrderedDict`. You can get the resulting dictionary that is used for serialization by using the *`to_dict`* method...

Given the following config instance `config`...

```
from typing import List, Dict

@file_config.config
class ProjectConfig(object):

   @file_config.config
   class Dependency(object):
      name = file_config.var(str, min=1)
      version = file_config.var(file_config.Regex(r"^v\d+$"))

   name = file_config.var(str, min=1)
```

(continues on next page)

(continued from previous page)

```
    type_ = file_config.var(str, name="type", required=False)
    keywords = file_config.var(List[str], min=0, max=10)
    dependencies = file_config.var(Dict[str, Dependency])

config = ProjectConfig(
    name="My Project",
    type_="personal-project",
    keywords=["example", "test"],
    dependencies={
        "a-dependency": ProjectConfig.Dependency(name="A Dependency", version="v12")
    },
)
```

You can get the resulting `collections.OrderedDict` with the following method call...

```
>>> file_config.to_dict(config)
OrderedDict([('name', 'My Project'),
             ('type', 'personal-project'),
             ('keywords', ['example', 'test']),
             ('dependencies',
              {'a-dependency': OrderedDict([('name', 'A Dependency'),
                                            ('version', 'v12')])})])
```

This *to_dict* method is used implicitly by all of the available `file_config.handlers`. These handlers provide an abstract interface to dumping and loading config instances to and from different formats.

For every config instance you create, several methods and classmethods are added to the instance. These methods follow the naming standard `dumps_x`, `dump_x`, `loads_x`, and `load_x` where is the name of the handler. **Thsese methods will always exist**, however, if you are missing a module required to do serialization for a specific format, the handler will raise an exception...

```
>>> config.dumps_toml()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/stephen-bunn/.virtualenvs/tempenv-4b8296442234/lib/python3.6/site-
↪packages/file_config/_file_config.py", line 53, in _handle_dumps
    return handler.dumps(to_dict(self))
  File "/home/stephen-bunn/.virtualenvs/tempenv-4b8296442234/lib/python3.6/site-
↪packages/file_config/handlers/_common.py", line 49, in dumps
    dumps_hook_name = f"on_{self.imported}_dumps"
  File "/home/stephen-bunn/.virtualenvs/tempenv-4b8296442234/lib/python3.6/site-
↪packages/file_config/handlers/_common.py", line 13, in imported
    self._imported = self._discover_import()
  File "/home/stephen-bunn/.virtualenvs/tempenv-4b8296442234/lib/python3.6/site-
↪packages/file_config/handlers/_common.py", line 46, in _discover_import
    raise ModuleNotFoundError(f"no modules in {self.packages!r} found")
ModuleNotFoundError: no modules in ('pytoml', 'tomlkit') found
```

In order to serialize a config instance out into toml you will need to either have `pytoml` or `tomlkit` present (as indicated by the exception)...

```
pipenv install file-config[pytoml]
# or...
pip install file-config[pytoml]
```

After installing the pytoml extra dependency, you should be able to dump out to toml...

```
>>> config.dumps_toml()
name = "My Project"
type = "personal-project"
keywords = ["example", "test"]
[dependencies]
[dependencies.a-dependency]
name = "A Dependency"
version = "v12"
```

Note: pytoml typically does insert newlines betwen secions like `[dependencies]` and `[dependencies.a-dependency]`. Unfortunately, I don't know how to represent it in restructuredText examples as the `\` character is not actually escaped...

Loading from this toml content is also super straight forward...

```
>>> ProjectConfig.loads_toml('''name = "My Project"\ntype = "personal-project
↪"\nkeywords = ["example", "test"]\n\n[dependencies]\n\n[dependencies.a-
↪dependency]\nname = "A Dependency"\nversion = "v12"\n''')
ProjectConfig(name='My Project', type_='personal-project', keywords=['example', 'test
↪'], dependencies={'a-dependency': ProjectConfig.Dependency(name='A Dependency',␣
↪version='v12')})
```

You can do this for all supported serialization types listed in the `file_config.handlers` module.
By default several serialization types are supported without any need for extra dependencies...

- `json` - via `json` (prefers ujson if module is present)
- `pickle` - via `pickle`

```
>>> config.dumps_json()
'{"name":"My Project","type":"personal-project","keywords":["example","test"],
↪"dependencies":{"a-dependency":{"name":"A Dependency","version":"v12"}}}'
>>> config.dumps_pickle()
b
↪'\x80\x04\x95\xbb\x00\x00\x00\x00\x00\x00\x8c\x0bcollections\x94\x8c\x0bOrderedDict\x94\x93\x94
↪Project\x94\x8c\x04type\x94\x8c\x10personal-
↪project\x94\x8c\x08keywords\x94]\x94(\x8c\x07example\x94\x8c\x04test\x94e\x8c\x0cdependencies\x94}
↪\x94\x8c\x0ca-dependency\x94h\x02)R\x94(h\x04\x8c\x0cA␣
↪Dependency\x94\x8c\x07version\x94\x8c\x03v12\x94usu.'
```

1.5.1 Handler Preference

There are usually multiple handlers that can deal with serialization / deserialization of a specific format (e.g. `json` and `ujson`). You can declare a preference by setting the `prefer` keyword argument...

```
>>> file_config.dumps_json(prefer="json")
'{"name":"My Project","type":"personal-project","keywords":["example","test"],
↪"dependencies":{"a-dependency":{"name":"A Dependency","version":"v12"}}}'
```

This will prefer `json` over `ujson` if both are present.

1.5.2 Dumping Options

Some dumping handlers have special options that are not immediately present on the `dumps` or `dump` methods. For example, the `json` handler has the ability to dump with a specific indentation level by passing in the `indent` keyword argument to the `dumps_json` method...

```
>>> config.dumps_json(indent=2)
{
  "name":"My Project",
  "type":"personal-project",
  "keywords":[
    "example",
    "test"
  ],
  "dependencies":{
    "a-dependency":{
      "name":"A Dependency",
      "version":"v12"
    }
  }
}
```

Unfortunately, some serialization packages (that handle the same format) do not have the ability to implement the same features. For example, `tomlkit` can support inline tables, but `pytoml` can not. You will notice that if you try to use the `inline_table` argument using `pytoml` you will get a warning similar to this...

```
>>> config.dumps_toml(prefer="pytoml", inline_tables=["dependencies.*"])
/home/stephen-bunn/Git/file-config/file_config/handlers/toml.py:72: UserWarning:␣
↪pytoml does not support 'inline_tables' argument, use tomlkit instead
  "pytoml does not support 'inline_tables' argument, use tomlkit instead"
name = "My Project"
type = "personal-project"
keywords = ["example", "test"]
[dependencies]
[dependencies.a-dependency]
name = "A Dependency"
version = "v12"
```

Whereas using `tomlkit` will handle the argument appropriately...

```
>>> config.dumps_toml(prefer="tomlkit", inline_tables=["dependencies.*"])
name = "My Project"
type = "personal-project"
keywords = ["example", "test"]
[dependencies]
a-dependency = {name = "A Dependency",version = "v12"}
```

1.5.3 Loading Defaults

When loading a config instance from some serialized content, you should be aware of what kind of data you want the built config instance to contain. There are currently three different ways the *var* `default` kwarg is treated when loading serialized content.

For example, take the following config...

```
import file_config

@file_config.config
class ParentConfig(object):

   @file_config.config
   class ChildConfig(object):

      bar = var(str, default="Default", required=False)

   foo = var(str, default=False, required=False)
   bar = var(int)
   child = var(ChildConfig, default=ChildConfig, required=False)
```

If `default` is `None` (which it is by default) the config instance attribute is set to `None`. *Notice what happens when we load from content missing the "bar" attribute.*

```
>>> config_1 = ParentConfig.loads_json(
...     '{"foo": "Testing", "child": {"bar": "Testing"}}'
... )
ParentConfig(foo='Testing', bar=None, child=ParentConfig.ChildConfig(bar='Testing'))
```

If the `var` type is another config class and `default` is the **exact** same config class, the empty state of the given config class is built as the default. *Notice what happens when we load from content missing the "child" attribute.*

```
>>> config_2 = ParentConfig.loads_json('{"foo": "Testing", "bar": 1}')
ParentConfig(foo='Testing', bar=1, child=ParentConfig.ChildConfig(bar='Default'))
```

If `default` is any other value, the config instance attribute is set to that given value. *Notice what happens when we load from content missing the "foo" attribute.*

```
>>> config_3 = ParentConfig.loads_json(
...     '{"bar": 1, "child": {"bar": "Testing"}}'
... )
ParentConfig(foo=False, bar=1, child=ParentConfig.ChildConfig(bar='Testing'))
```

CHAPTER

TWO

HANDLERS

Handlers are what provide functionality to serialize to and deserialize from various file formats. All of the supported file formats should be attribute-value pair data exchange formats that can be serialized to and from dictionaries.

Whenever a new config instance is created some generic serialization and deserialization methods are *magically* created based on what handlers are currently available. The exposed methods are the following:

- `config_instance.dump_x(file_object)` - *dumps an instance to a file object*
- `config_instance.dumps_x()` - *dumps the content to a string*
- `ConfigClass.load_x(file_object)` - *loads an instance from a file object*
- `ConfigClass.loads_x(string)` - *loads an instance from a string*

where `x` is the generic name of the handler (e.g. `json`, `ini`, `pickle`, `toml`, etc...). I will commonly refer to these methods as the **config serialization methods**.

Important: Since handlers such as `JSONHandler` have support for multiple serializer libraries, a keyword `prefer` is supplied to all the config serialization methods. You can use this keyword to indicate what library you wish to use for that specific dump or load.

```
# if you have python-rapidjson installed but would prefer to use the builtin json
config_instance.dumps_json(prefer="json")
```

The value of the `prefer` keyword is the **module name** not the package name. For example pyyaml's package name is `pyyaml` but it's importable module name is `yaml`. So you would put `prefer="yaml"` rather than `prefer="pyyaml"`.

For the following documentation I will be using the following config instance to showcase the handler's serialization.

```
from typing import List, Dict
from enum import Enum
import file_config


class Status(Enum):
   STOPPED = 0
   STARTED = 1


@file_config.config
class ProjectConfig(object):

   @file_config.config
```

(continues on next page)

(continued from previous page)

```
    class Dependency(object):
        name = file_config.var(str, min=1)
        version = file_config.var(file_config.Regex(r"^v\d+$"))

    name = file_config.var(str, min=1)
    type_ = file_config.var(str, name="type", required=False)
    keywords = file_config.var(List[str], min=0, max=10)
    status = file_config.var(Status)
    dependencies = file_config.var(Dict[str, Dependency])


config_instance = ProjectConfig(
    name="My Project",
    type_="personal-project",
    keywords=["example", "test"],
    status=Status.STOPPED,
    dependencies={
        "a-dependency": ProjectConfig.Dependency(name="A Dependency", version="v12")
    },
)
```

2.1 JSON

JSON is probably the most popular, supported, and straightforward data exchange formats available right now.

Available formatting options for JSON serializers are…

- `indent=2` - *the number of spaces to use for indentation*
- `sort_keys=True` - *sorts keys alphabetically in the resulting json*

2.1.1 `json`

Uses the builtin `json` module for serialization.

```
config_instance.dumps_json(prefer="json", indent=2)
```

```
{
    "name": "My Project",
    "type": "personal-project",
    "keywords": [
        "example",
        "test"
    ],
    "status": 0,
    "dependencies": {
        "a-dependency": {
            "name": "A Dependency",
            "version": "v12"
        }
    }
}
```

2.1.2 python-rapidjson

Compatability with `python-rapidjson` requires the installation of it as an extra...

```
pipenv install file-config[python-rapidjson]
```

Serialization is the same as the default *json* handler...

```
config_instance.dumps_json(prefer="rapidjson", indent=2)
```

```
{
   "name": "My Project",
   "type": "personal-project",
   "keywords": [
      "example",
      "test"
   ],
   "status": 0,
   "dependencies": {
      "a-dependency": {
         "name": "A Dependency",
         "version": "v12"
      }
   }
}
```

2.1.3 ujson

In my opinion people shouldn't be using `ujson` since they don't follow the JSON spec and are thus incompatible with the default `json` module for many edge cases. But I know that lots of packages still depend on it (*for unknown reasons*).

Support for `ujson` requires it to be installed as an extra...

```
pipenv install file-config[ujson]
```

Usage is similar, however notice the resulting json lacks whitespace between key value pairs (a quirk of `ujson`).

```
config_instance.dumps_json(prefer="ujson", indent=2)
```

```
{
   "name":"My Project",
   "type":"personal-project",
   "keywords":[
      "example",
      "test"
   ],
   "status":0,
   "dependencies":{
      "a-dependency":{
         "name":"A Dependency",
         "version":"v12"
      }
   }
}
```

2.2 INI

INI is another popular configuration file format. Although it lacks features available in other file formats people still tend to use it over better solutions (*TOML* following a very similar specification).

Important: INI does not support arrays of dictionaries. Doing so would break the specification and be difficult to read. If your resulting dictionary from `file_config.to_dict(config_instance)` has an array containing dictionaries the config serialization methods for ini serialization will raise a `ValueError`.

If you need to structure your data this way, please switch to using *TOML* as that is one of their key features.

Available formatting options for INI serializers are...

- `root="root"` - *the name of the root section of the resulting ini*
- `delimiter=":"` - *the delimiter to use to indicate nested dictionaries*
- `empty_sections=True` - *allows empty ini sections to exist*

2.2.1 `configparser`

Uses the builtin `configparser` module for serialization. A custom `INIParser` is used since the default `configparser` module is pretty lackluster.

```
config_instance.dumps_ini(prefer="configparser", root="root")
```

```
[root]
name = "My Project"
type = personal-project
keywords = example
   test
status = 0

[root:dependencies:a-dependency]
name = "A Dependency"
version = v12
```

2.3 Pickle

You really shouldn't ever be serializing and storing things to `pickle` syntax since it has pretty serious security flaws with how it is loaded back in. Anyway, here is how you can do it in `file_config`.

There are no format options available for the pickle handler.

2.3.1 `pickle`

Uses the builtin `pickle` module for serialization.

```
config_instance.dumps_pickle(prefer="pickle")
```

```
\x80\x04\x95\xc6\x00\x00\x00\x00\x00\x00\x00\x8c\x0bcollections\x94\x8c\x0bOrderedDict\x94\x93\x94)R\
↪Project\x94\x8c\x04type\x94\x8c\x10personal-
↪project\x94\x8c\x08keywords\x94]\x94(\x8c\x07example\x94\x8c\x04test\x94e\x8c\x06status\x94K\x00\x8
↪\x94\x8c\x0ca-dependency\x94h\x02)R\x94(h\x04\x8c\x0cA␣
↪Dependency\x94\x8c\x07version\x94\x8c\x03v12\x94usu.
```

(continues on next page)

(continued from previous page)

2.4 TOML

There are a thousand libraries for parsing toml and they are all terrible. Maybe toml is just poorly designed but every single toml parsing library I've used (in Python) either doesn't fully parse toml correctly or has some odd quirks that make it hard to work with.

Nevertheless, here are the three best libraries that *currently* exist for parsing toml.

The format options available to toml are the following:

- `inline_tables=["dependences.*"]` - *a glob pattern for inlining tables*

2.4.1 tomlkit

Using tomlkit requires it to be installed as an extra...

```
pipenv install file-config[tomlkit]
```

Usage is just as you might expect...

```
config_instance.dumps_toml(prefer="tomlkit", inline_tables=["dependencies.*"])
```

```
name = "My Project"
type = "personal-project"
keywords = ["example", "test"]
status = 0

[dependencies]
a-dependency = {name = "A Dependency",version = "v12"}
```

2.4.2 toml

Using toml requires it to be installed as an extra...

```
pipenv install file-config[toml]
```

Usage is just the same as *tomlkit*...

```
config_instance.dumps_toml(prefer="toml", inline_tables=["dependencies.*"])
```

```
name = "My Project"
type = "personal-project"
keywords = [ "example", "test",]
status = 0

[dependencies]
a-dependency = { name = "A Dependency", version = "v12" }
```

2.4.3 pytoml

Using pytoml requires it to be installed as an extra...

```
pipenv install file-config[pytoml]
```

Pytoml does not support serializing inline tables in any easy manner. So the `inline_tables` keyword won't be applied when dumping with pytoml.

```
config_instance.dumps_toml(prefer="pytoml")
```

```
name = "My Project"
type = "personal-project"
keywords = ["example", "test"]
status = 0

[dependencies]

[dependencies.a-dependency]
name = "A Dependency"
version = "v12"
```

2.5 YAML

Yaml is a data exchange langauge used by many projects (Docker, TravisCI, etc...) for simple configuration files.

There are no format options for yaml serialization.

2.5.1 pyyaml (yaml)

Using pyyaml requires it to be installed as an extra...

```
pipenv install file-config[pyyaml]
```

Usage is straightforward...

```
config_instance.dumps_yaml(prefer="yaml")
```

```
name: My Project
type: personal-project
keywords: [example, test]
status: 0
dependencies:
a-dependency:
   name: A Dependency
   version: v12
```

2.6 Message Pack

MessagePack is a byte sized json format which retains the same structure as json. It's valuable for quick and fast json streams over network protocols.

There are no format options for msgpack.

2.6.1 msgpack

Using msgpack requires it to be installed as an extra...

```
pipenv install file-config[msgpack]
```

Usage is just as you would expect...

```
config_instance.dumps_msgpack(prefer="msgpack")
```

```
\x85\xa4name\xaaMy Project\xa4type\xb0personal-
↪project\xa8keywords\x92\xa7example\xa4test\xa6status\x00\xacdependencies\x81\xaca-
↪dependency\x82\xa4name\xacA Dependency\xa7version\xa3v12
```

2.7 XML

XML is an older data exchange format that follows a nested tag-attribute type structure. A custom `XMLParser` is used since the many dictionary to xml helper packages out there are not reflective.

The format options available to xml are the following:

- `root="root"` - *the name of the root element in the resulting xml*
- `pretty=True` - *indicates that the resulting xml should be pretty formatted*
- `xml_declaration=True` - *indicates that the xml header should be added*
- `encoding="utf-8"` - *the encoding to use for the resulting xml document*

2.7.1 lxml

Using XML requires `lxml` to be installed as an extra...

```
pipenv install file-config[lxml]
```

Usage is straightforward...

```
config_instance.dumps_xml(prefer="lxml", pretty=True, xml_declaration=True)
```

```
<?xml version='1.0' encoding='UTF-8'?>
<ProjectConfig>
   <name type="str">My Project</name>
   <type type="str">personal-project</type>
   <keywords>
      <keywords type="str">example</keywords>
      <keywords type="str">test</keywords>
   </keywords>
   <status type="int">0</status>
   <dependencies>
      <a-dependency>
         <name type="str">A Dependency</name>
         <version type="str">v12</version>
```

(continues on next page)

(continued from previous page)

```
      </a-dependency>
   </dependencies>
</ProjectConfig>
```

CHAPTER

THREE

CONTRIBUTING

When contributing to this repository, please first discuss the change you wish to make via an issue to the owners of this repository before submitting a pull request.

Important: We have an enforced style guide and a code of conduct. Please follow them in all your interactions with this project.

3.1 Style Guide

- We somewhat follow PEP8 and utilize Sphinx docstrings on **all** classes and functions.
- **We employee flake8 as our linter with exceptions to the following rules:**
  - D203
  - F401
  - E123
  - W503
  - E203
- Maximum line length for Python files is 88 characters.
- Maxiumum McCabe complexity is 13.
- Linting and test environments are configured via `tox.ini`.
- Imports are sorted using isort with multi-line output mode `3`.
- An `.editorconfig` file is included in this repository which dictates whitespace, indentation, and file encoding rules.
- Although `requirements.txt` and `requirements-dev.txt` do exist, Pipenv is utilized as the primary virtual environment and package manager for this project.
- We strictly utilize Semantic Versioning as our version specification.
- All Python source files are post-processed using ambv/black.

3.2 Issues

Issues should follow the included `ISSUE_TEMPLATE` found in `.github/ISSUE_TEMPLATE.md`.

- **Issues should contain the following sections:**
  - Expected Behavior
  - Current Behavior
  - Possible Solution
  - Steps to Reproduce (for bugs)
  - Context
  - Your Environment

These sections help the developers greatly by providing a large understanding of the context of the bug or requested feature without having to launch a full fleged discussion inside of the issue.

3.3 Pull Requests

Pull requests should follow the included `PULL_REQUEST_TEMPLATE` found in `.github/PULL_REQUEST_TEMPLATE.md`.

- Pull requests should always be from a **topic/feature/bugfix** (left side) branch. *Pull requests from master branches will not be merged.*
- Pull requests should not fail our requested style guidelines or linting checks.

3.4 Code of Conduct

Our code of conduct is taken directly from the Contributor Covenant since it directly hits all of the points we find necessary to address.

3.4.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

3.4.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances

- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

3.4.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

3.4.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

3.4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at stephen@bunn.io. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

3.4.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

CHAPTER

FOUR

CHANGELOG

All notable changes to this project will be documented in this file.
The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

4.1 1.0.0 (*2019-10-07*)

4.1.1 Features

- Adding the ability to define custom JSONSchema `$id` and `$schema` properties. Will raise user warnings if unsupported schema draft is specified. #41

4.1.2 Documentation

- Adding some documentation around `schema_id` and `schema_draft` parameters for the `@config()` decorator in Getting Started

4.1.3 Miscellaneous

- Adding `long_description_content_type` to `setup.cfg` for new twine package check warnings

4.2 0.3.10 (*2019-10-04*)

4.2.1 Bug Fixes

- Allowing Python 3 typehints to be considered as the config var's type just like the var's `type` kwarg #38

4.2.2 Miscellaneous

- Fixing access to collections.abc for Iterable deprecation warning
- Updating package stubs to mimic true kwargs typings

4.3 0.3.9 (*2019-06-15*)

4.3.1 Miscellaneous

- Adding kwarg passthrough for `config` so you can now use things like `config(hash=True)`
- Adding *extremely* basic reflection tests for `ujson` serialization (please use `rapidjson` instead of `ujson`)

4.4 0.3.8 (*2019-04-01*)

4.4.1 Bug Fixes

- Fixed issue related to the `prefer` keyword not loading the proper modules for serialization / deserialization when `prefer` had already been invoked in subsequent calls for a given config #36

4.4.2 Documentation

- Adding codecov for alternate coverage reporting since codacy struggles sometimes and isn't always correct

4.4.3 Miscellaneous

- Adding and improving tests for serialization handlers
- Improving testing and code coverage for `schema_builder` and `_file_config`

4.5 0.3.7 (*2019-03-22*)

4.5.1 Bug Fixes

- Adding the ability for nested configs to default to the empty state of the nested config if desired #26

4.5.2 Documentation

- Adding documentation `Loading Defaults` in "Getting Started" regarding the different results when loading defaults from serialized content

4.5.3 Miscellaneous

- Updating the copyright year in the docs to 2019

4.6 0.3.6 (*2019-03-15*)

4.6.1 Bug Fixes

- Fixing failures when loading from serialized content missing configuration for nested configs #24

4.6.2 Miscellaneous

- Adding latest PyPi version badge to documentation and README
- Adding basic auto-generated typing stubs under `stubs/`

4.7 0.3.5 (*2019-02-15*)

4.7.1 Bug Fixes

- Allowing default values to be used in configs when loading from dictionaries that are missing the value #22

4.8 0.3.4 (*2019-01-11*)

4.8.1 Bug Fixes

- Fixing `typing._GenericAlias` typecasting raising `AttributeError` #21

4.9 0.3.3 (*2019-01-10*)

4.9.1 Bug Fixes

- Fixing regex check bug where `type_.__name__` raises an `AttributeError` on `typing.List` edge case #19
- Fixing `dump_x` handlers not using kwargs like `dumps_x` handlers #20

4.10 0.3.2 (*2019-01-09*)

4.10.1 Bug Fixes

- Fixing schema builder where building schemas for object types with nested typing types silently fails #18

4.10.2 Documentation

- Showing newly documented private methods in package documentation
- Adding basic docstrings for private schema_builder functions
- Adding basic docstrings for util functions

4.10.3 Miscellaneous

- Updating copyright statements from 2018 to 2019
- Fixing missing wheel in release

4.11 0.3.1 (*2018-12-18*)

4.11.1 Features

- Adding defusedxml as fromstring reader in `XMLParser` #17

4.11.2 Miscellaneous

- Fixing lxml required for import

4.12 0.3.0 (*2018-12-16*)

4.12.1 Features

- Adding basic ini support through `configparser` #10
- Adding basic xml support through `lxml` #12

4.12.2 Documentation

- Splitting up Sphinx autodocs into separate sections
- Adding Handlers section to documentation

4.12.3 Miscellaneous

- Adding `TYPE_MAPPINGS` to `utils.py` as a way of generically representing available types and their translations
- Project Restructure - restructuring project to provide a better development experience
- Updating from MIT to ISC licensing

4.13 0.2.0 (*2018-11-07*)

- adding serialization and deserialization support for enums

4.14 0.1.0 (*2018-10-26*)

- adding `encoder` and `decoder` var kwargs for customizing how a specific var is serialized/deserialized
- adding support for python-rapidjson as json serializer

4.15 0.0.8 (*2018-10-16*)

- adding `sort_keys` support for `json` dumpers
- adding conditional `validate` boolean flag for `load_<json,toml,yaml,etc...>` config method (performs pre-validation of loaded dictionary)
- fixing typecasting of loaded content when var is missing in content, now sets var to None
- improved tests via a hypothesis dynmaic config instance builder
- removing support for complex vars since no serializers support them

4.16 0.0.7 (*2018-10-12*)

- adding `prefer` keyword to use specific serialization handler
- adding `inline_tables` argument for `toml` handlers (takes a list of fnmatch patterns)
- adding support for toml

4.17 0.0.6 (*2018-10-08*)

- fixing `make_config` not using any passed in `file_config.var` instances
- added `indent` dumping argument for `JSONHandler`
- improved documentation in `file_config.schema_builder`
- improved sphinx linking from `getting-started.rst` to generated autodocs

4.18 0.0.5 (*2018-10-05*)

- added better docstrings
- added better documentation in rtd
- fixed `file_config._file_config._build` for `file_config.Regex` types
- fixed `file_config.utils.typecast` for serializing to `str` instead of None

4.19 0.0.4 (*2018-10-04*)

- added basic sphinx documentation
- fixing dynamic type casting for config var typing types

CHAPTER

FIVE

PROJECT REFERENCE

5.1 File Config Package

The module that exposes mostly all of the package's functionality.

Although these methods are privatized under the *_file_config* namespace they can be accessed from the imported `file_config` module...

```
import file_config

@file_config.config
class Config(object):
   name = file_config.var(str)
```

5.1.1 Submodules

- `handlers` — *Dumping / Loading handlers for different formats*
- `schema_builder` — *Config JSONSchema builder*
- `utils` - *Utilities used within the module*
- `contrib` — *Additional utilities that benefit the package*

`file_config._file_config.`**`config`**(*maybe_cls=None*, *these=None*, *title=None*, *description=None*, *schema_id=None*, *schema_draft=None*, ***kwargs*)

File config class decorator.

Usage is to simply decorate a **class** to make it a *config* class.

```
>>> import file_config
>>> @file_config.config(
        title="My Config Title",
        description="A description about my config"
    )
    class MyConfig(object):
        pass
```

Parameters

- **maybe_cls** (*class*) – The class to inherit from, defaults to None, optional
- **these** (*dict*) – A dictionary of str to `file_config.var` to use as attribs
- **title** (*str*) – The title of the config, defaults to None, optional
- **description** (*str*) – A description of the config, defaults to None, optional
- **schema_id** (*str*) – The JSONSchema `$id` to use when building the schema, defaults to None, optional
- **schema_raft** (*str*) – The JSONSchema `$schema` to use when building the schema, defaults to None, optional

Returns Config wrapped class

Return type class

`file_config._file_config.`**from_dict**(*config_cls*, *dictionary*, *validate=False*)

Loads an instance of `config_cls` from a dictionary.

Parameters

- **config_cls** (*type*) – The class to build an instance of
- **dictionary** (*dict*) – The dictionary to load from
- **validate** (*bool*) – Preforms validation before building `config_cls`, defaults to False, optional

Returns An instance of `config_cls`

Return type object

`file_config._file_config.`**make_config**(*name*, *var_dict*, *title=None*, *description=None*, *schema_id=None*, *schema_draft=None*, ***kwargs*)

Creates a config instance from scratch.

Usage is virtually the same as `attr.make_class()`.

```
>>> import file_config
>>> MyConfig = file_config.make_config(
        "MyConfig",
        {"name": file_config.var(str)}
    )
```

Parameters

- **name** (*str*) – The name of the config
- **var_dict** (*dict*) – The dictionary of config variable definitions
- **title** (*str*) – The title of the config, defaults to None, optional
- **description** (*str*) – The description of the config, defaults to None, optional
- **schema_id** (*str*) – The JSONSchema `$id` to use when building the schema, defaults to None, optional
- **schema_raft** (*str*) – The JSONSchema `$schema` to use when building the schema, defaults to None, optional

Returns A new config class

Return type class

file_config._file_config.**to_dict**(*instance*, *dict_type=<class 'collections.OrderedDict'>*)
Dumps an instance to an dictionary mapping.

Parameters

- **instance** (*object*) – The instance to dump
- **dict_type** (*object*) – The dictionary type to use, defaults to OrderedDict

Returns Dictionary serialization of instance

Return type OrderedDict

file_config._file_config.**validate**(*instance*)
Validates a given instance.

Parameters **instance** (*object*) – The instance to validate

Raises **jsonschema.exceptions.ValidationError** – On failed validation

file_config._file_config.**var**(*type=None*, *default=None*, *name=None*, *title=None*, *description=None*, *required=True*, *examples=None*, *encoder=None*, *decoder=None*, *min=None*, *max=None*, *unique=None*, *contains=None*, ***kwargs*)
Creates a config variable.

Use this method to create the class variables of your *config* decorated class.

```
>>> import file_config
>>> @file_config.config
    class MyConfig(object):
        name = file_config.var(str)
```

Parameters

- **type** (*type*) – The expected type of the variable, defaults to None, optional
- **default** – The default value of the var, defaults to None, optional
- **name** (*str*) – The serialized name of the variable, defaults to None, optional
- **title** (*str*) – The validation title of the variable, defaults to None, optional
- **description** (*str*) – The validation description of the variable, defaults to None, optional
- **required** (*bool*) – Flag to indicate if variable is required during validation, defaults to True, optional
- **examples** (*list*) – A list of validation examples, if necessary, defaults to None, optional
- **encoder** – The encoder to use for the var, defaults to None, optional
- **decoder** – The decoder to use for the var, defaults to None, optional
- **min** (*int*) – The minimum constraint of the variable, defaults to None, optional
- **max** (*int*) – The maximum constraint of the variable, defaults to None, optional
- **unique** (*bool*) – Flag to indicate if variable should be unique, may not apply to all variable types, defaults to None, optional
- **contains** – Value that list varaible should contain in validation, may not apply to all variable types, defaults to None, optional

Returns A new config variable

Return type attr.Attribute

PYTHON MODULE INDEX

f

INDEX