PyFiguration

Jul 07, 2020

Contents

PyFiguration is a configuration tool for Python. It allows you to define which configuration are used from right inside your code. The PyFiguration command line tool helps you inspect which configurations are available, what the allowed values are, and helps to inspect the validity of a configuration file for a specific script.

CHAPTER 1

Basic usage

In your code you can define which configurations should be available. This example creates a simple Flask server. The port on which the server should start depends on the configuration.

```
""" script.py
"""
from pyfiguration import conf
from flask import Flask


@conf.addIntField(
    field="server.port",
    description="The port on which the server will start",
    default=8000,
    minValue=80,
    maxValue=9999,
)
def startServer():
    app = Flask(__name__)
    port = conf["server"]["port"]
    print(f"Starting on port {port}")
    app.run(port=port)


if __name__ == "__main__":
    startServer()
```

You can use the PyFiguration command line tool to inspect this module/script:

```
$ pyfiguration inspect script -s script.py
The following options can be used in a configuration file for the module 'script.py':
server:
  port:
    allowedDataType: int
    default: 8000
```

(continues on next page)

(continued from previous page)

```
    description: The port on which the server will start
    maxValue: 9999
    minValue: 80
    required: true
```

This tells you that the default value for server.port is 8000, and that it should be an integer between 80 and 9999. Running the script (`python script.py`) will start the server on the default port. Lets create a configuration file to overwrite the default:

```
# config.yaml
server:
  port: 5000
```

Now we can start the script again, pointing to the config file to use it:

```
$ python script.py --config ./config.yaml
Starting on port 5000
 * Serving Flask app "script" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Success! The script has used the configuration we've defined in `config.yaml` file. It's possible to use multiple configuration files, and both in `YAML` and `JSON` formats. Note that the keys will be overwritten if there are duplicates. This is a useful feature that you can use, for example, to set defaults in `defaults.yaml` and then overwrite with deployment specific settings in `deployment.yaml`. It's also possible to reference a full folder. PyFiguration will read all the files in the folder. For a full example checkout the `./examples` folder of this repository.

If you have a configuration file and a script, you can also use the PyFiguration command line to check the config file for errors. Imaging this configuration file:

```
# config_with_warnings.yaml
server:
  port: 500.0
  not_needed_key: some random value
```

We've obviously made 2 mistakes here: 1: the port is a float, 2: there is a key that is not being used by our script. Lets use the command line tool to investigate.

```
$ pyfiguration inspect config -s script.py -c config_with_warnings.yaml
--------
 Errors
--------
    Value '500.0' is not of the correct type. The allowed data type is: int
----------
 Warnings
----------
   ! Key 'server.not_needed_key' doesn't exist in the definition and is not used in␣
↪the module.
```

1.1 Quickstart

The following instructions will get you up and running in no time.

1.1.1 Install PyFiguration

Run the following command to install PyFiguration:

```
$ pip install pyfiguration
```

Or add PyFiguration to your `requirements.txt` or `setup.py`.

1.1.2 Import PyFiguration in your script

Add the following line to any script to start making use of PyFiguration:

```
from pyfiguration import conf
```

1.1.3 Create new configuration options

Decorate your functions to define which configurations should be made available, and how to check if the provided values are valid. Example:

```
@conf.add_int_field("port", minValue=80, maxValue=9999, default=8000)
def my_function():
    ...
```

1.1.4 Use configuration in your code

The `conf` object will contain all the parsed configurations. Just access the value like this:

```
port = conf["port"]
```

1.1.5 Write a config file

Running with default values can be fine, but of course you want to be able to overwrite them. Define a `YAML` or `JSON` file with your config, and provide the file as an argument to your script when running:

```
# config.yaml
port: 6000
```

```
$ python my_script.py --config config.yaml
```

1.2 Installation guide

1.2.1 Installing PyFiguration

PyFiguration runs on Python 3.7+ and is available on PyPi. Run the following command to install PyFiguration on your machine:

```
$ pip install pyfiguration
```

1.2.2 External dependencies

The following external depencies will be installed automatically:

- `click`
- `colorama`
- `pyyaml`

1.3 Writing configuration

1.3.1 Specifying configurations

The recommended way to define configurations with PyFiguration is to decorate the function that uses the configuration. This way you always have the definition of the configuration close to where the configuration is used. PyFiguration comes with a set of decorators that create different types of fields.

The sections below show examples of how fields can be specified. For the full list of options per type, check out the *API* section of this documentation.

Integers

```
@conf.add_int_field("port", description="The port number of a server")
@conf.add_int_field("port", minValue=80, maxValue=999)
@conf.add_int_field("port", default=10)
@conf.add_int_field("port", required=False)
```

Floats

```
@conf.add_float_field("length", description="The length in cm")
@conf.add_float_field("length", minValue=10.0, maxValue=100.0)
@conf.add_float_field("length", default=10.0)
@conf.add_float_field("length", required=False)
```

Strings

```
@conf.add_string_field("host", description="The url of the host to connect to")
@conf.add_string_field("host", allowedValues=["localhost", "remotehost"])
@conf.add_string_field("host", required=False)
```

Booleans

```
@conf.add_boolean_field("isRemote", description="Whether or not something is true")
@conf.add_boolean_field("isRemote", required=False)
```

Lists

```
@conf.add_list_field("users", description="A list of users that should have access")
@conf.add_list_field("users", allowedValues=["admin", "myusername"])
@conf.add_list_field("users", required=False)
```

1.3.2 Nested fields

It is possible to nest configurations (e.g. in sections). This is standard practice in `YAML` and `JSON` files and is supported by PyFiguration. Just define the configuration field in "dot-notation". Combine all the sections of the path to the key you would like to define with ".". Here is an example:

```
# config.yaml
database:
  host: localhost
  port: 8000
```

```
# script.py
from pyconfiguration import conf

@conf.add_string_field("database.host", description="The database URL")
@conf.add_int_field("database.port", description="The port to connect to on the␣
↪database")
def connect():
    host = conf["database"]["host"]
    port = conf["database"]["port"]
```

1.4 Using configurations

When you've defined the configuration your script needs, using the decorators, and you've written a configuration file it's time to use this configuration with your script. PyFiguration will automatically parse the `--config CONFIGFILES` arguments to your script and use them (in order!) to load the configuration for your script.

1.4.1 Simple configurations

In the simplest form you can use a single config file with your script like this:

```
# config.yaml
database:
  host: localhost
  port: 8000
```

```
# script.py
from pyconfiguration import conf

@conf.add_string_field("database.host", description="The database URL")
@conf.add_int_field("database.port", description="The port to connect to on the␣
↪database")
def connect():
    host = conf["database"]["host"]
    port = conf["database"]["port"]
    print(f"Host: {host}")
    print(f"Port: {port}")
    ...
```

```
$ python script.py --config config.yaml
Host: localhost
Port: 8000
...
```

1.4.2 Configuration inheritance

When you specify more than 1 configuration file to be used with your script, PyFiguration will go over the specified files one by one, and load all of them. The order in which you provide the configuration files matters! Configurations from the first file will be overwritten with configurations from the second, and so on.

This allows patterns like this:

```
$ ls .
config.yaml        defaults.yaml       script.py

$ python script.py --config defaults.yaml config.yaml
```

In this example PyFiguration will first load the defaults from `defaults.yaml` and will then overwrite the configuration that are also specified in `config.yaml`.

1.4.3 From directories

PyFiguration is also able to accept a directory of configuration files on the command line. If a directory is specified, PyFiguration will go through the directory and load all `.yaml`, `.yml`, and `.json` files, also from subdirectories. Note that the order is not guaranteed!

This allows patterns like this:

```
$ ls
defaults/          deployments/        script.py

$ ls defaults/
database.yaml      server.yaml

$ ls deployments/
deployment_a.yaml  deployment_b.yaml

$ python script.py --config defaults/ deployments/deployment_a.yaml
```

In this example we have default configurations for the server and the database in separate files in the `defaults/` folder. We provide our script with the folder so it will load both. Then we provide the script with one of the deployment files to overwrite some of the defaults.

1.5 Command line tool

PyFiguration ships with a convenient command line tool that can generate documentation for the configuration of a script, and that can inspect a give configuration for errors and potential mistakes.

1.5.1 Getting help

Run the following command to get the full usage instructions for the command line tool:

```
$ pyfiguration --help
usage: pyfiguration [-h] {inspect} ...

PyFiguration commandline tool

Use this commandline tool to inspect scripts and have a look at the
configuration options that the script provides. Furthermore you can
inspect configuration files directly.

optional arguments:
  -h, --help  show this help message and exit

Commands:
  {inspect}
    inspect   Inspect configurations and scripts



$ pyfiguration inspect --help
usage: pyfiguration inspect [-h] {config,script} ...

Inspect configuration files, scripts or modules to see which values are
allowed, or to check if a provided configuration file is valid for a specific
script.

optional arguments:
  -h, --help       show this help message and exit

Commands:
  The type object you would like to inspect

  {config,script}
    config         Inspect a configuration file to see if it is valid for a
                   given script
    script         Inspect a script to see what configuration options are
                   available



$ pyfiguration inspect script --help
usage: pyfiguration inspect script [-h] [-s SCRIPT]
```

(continues on next page)

(continued from previous page)

```
Provide a file or script to inspect it with PyFiguration. This command will
load the script from file and inspect the PyFiguration decorators to find out
what the configuration options are. Then, it will display all the option as
the output of this command. SCRIPT is the filename of the script to inspect
with PyFiguraton

optional arguments:
  -h, --help            show this help message and exit
  -s SCRIPT, --script SCRIPT
                        The script against which to inspect the config



$ pyfiguration inspect config --help
usage: pyfiguration inspect config [-h] [-c [CONFIG [CONFIG ...]]] [-s SCRIPT]

This command will load the SCRIPT and look at the defintion. Then it will load
the CONFIG file and makes sure the CONFIG file is valid for the provided
SCRIPT. SCRIPT is the filename of the SCRIPT to inspect with PyFiguraton,
CONFIG file is the configuration file to inspect, against the SCRIPT.

optional arguments:
  -h, --help            show this help message and exit
  -c [CONFIG [CONFIG ...]], --config [CONFIG [CONFIG ...]]
                        The configuration file to inspect
  -s SCRIPT, --script SCRIPT
                        The script against which to inspect the config
```

1.5.2 Inspect a script/module

To inspect a script or module and find out what the allowed configurations are, the command line tool can be used. An example:

```
$ pyfiguration inspect script -s basic.py
The following options can be used in a configuration file for the module 'basic.py':
db:
  host:
    allowedDataType: str
    default: localhost
    description: Location of the database, e.g. localhost
    required: true
  port:
    allowedDataType: int
    default: 8000
    description: Port of the database to connect on
    maxValue: 9999
    minValue: 80
    required: true
```

In this example we've inspected a script called `basic.py`. The output shows (in `YAML` format) which fields can be specified in a configuration file, what type the value should have, and how this value is checked (e.g. `minValue: 80` will check any configuration file to make sure the value for the field `port` is >= 80).

1.5.3 Inspect a configuration file

If you have a script, and a configuration file, you can check the configuration file to see if it's valid for your script. Example:

```
$ pyfiguration inspect config -s script.py -c config.yaml

$ pyfiguration inspect config -s script.py -c config_with_warnings.yaml
--------
 Errors
--------
    Value '500.0' is not of the correct type. The allowed data type is: int
----------
 Warnings
----------
   ! Key 'server.not_needed_key' doesn't exist in the definition and is not used in␣
↪the module.
```

In this example we run the inspector 2 times. The first time with a valid configuration file (so nothing is returned). The second time we have a configuration file with errors. Apparently we've specified a float where we should have specified an integer. Also, we've defined a key in our configuration that is not used by the script. This will not break the script, but because it might result in unexpected behaviour it's raised as a warning.

1.6 API

1.6.1 PyFiguration

The PyFiguration class is the class that is used for the *conf* object that is imported (*from pyfiguration import conf*). This class can be used to define what the configurations should look like, and to access the configurations once the're set.

class pyfiguration.pyfiguration.**PyFiguration**

Load and document configuration files the right way!

NOTE: All functions are implemented in snake case and have an alias in camel case (e.g. *add_int_field()* and *addIntField()*)

addBooleanField(*field: str*, *default: Optional[Any] = None*, *required: bool = True*, *description: Optional[str] = None*)

Add a boolean field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

addFloatField(*field: str*, *allowedValues: Optional[List[float]] = None*, *minValue: Optional[float] = None*, *maxValue: Optional[float] = None*, *required: bool = True*, *default: Optional[Any] = None*, *description: Optional[str] = None*)

Add a float field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **allowedValues** – The allowed values for this field in the configuration (optional)
- **minValue** – The minimum value for this field
- **maxValue** – The maximum value for this field
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

addIntField(*field: str*, *allowedValues: Optional[List[int]] = None*, *minValue: Optional[int] = None*, *maxValue: Optional[int] = None*, *required: bool = True*, *default: Optional[Any] = None*, *description: Optional[str] = None*)

Add a integer field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **allowedValues** – The allowed values for this field in the configuration (optional)
- **minValue** – The minimum value for this field
- **maxValue** – The maximum value for this field
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

addListField(*field: str*, *default: Optional[Any] = None*, *required: bool = True*, *description: Optional[str] = None*)

Add a list field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

addStringField(*field: str*, *allowedValues: Optional[List[str]] = None*, *required: bool = True*, *default: Optional[Any] = None*, *description: Optional[str] = None*)

Add a string field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **allowedValues** – The allowed values for this field in the configuration (optional)
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

add_boolean_field(*field: str*, *default: Optional[Any] = None*, *required: bool = True*, *description: Optional[str] = None*)
Add a boolean field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

add_float_field(*field: str*, *allowedValues: Optional[List[float]] = None*, *minValue: Optional[float] = None*, *maxValue: Optional[float] = None*, *required: bool = True*, *default: Optional[Any] = None*, *description: Optional[str] = None*)
Add a float field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **allowedValues** – The allowed values for this field in the configuration (optional)
- **minValue** – The minimum value for this field
- **maxValue** – The maximum value for this field
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

add_int_field(*field: str*, *allowedValues: Optional[List[int]] = None*, *minValue: Optional[int] = None*, *maxValue: Optional[int] = None*, *required: bool = True*, *default: Optional[Any] = None*, *description: Optional[str] = None*)
Add a integer field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **allowedValues** – The allowed values for this field in the configuration (optional)
- **minValue** – The minimum value for this field
- **maxValue** – The maximum value for this field
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

add_list_field(*field: str*, *default: Optional[Any] = None*, *required: bool = True*, *description: Optional[str] = None*)
Add a list field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

add_string_field(*field: str*, *allowedValues: Optional[List[str]] = None*, *required: bool = True*, *default: Optional[Any] = None*, *description: Optional[str] = None*)
Add a string field to the definition of the configuration.

Parameters

- **field** – The field to add to the definition
- **allowedValues** – The allowed values for this field in the configuration (optional)
- **required** – Whether this field is required or not
- **default** – The default value for this field if no value is specified in the configuration

Returns A wrapped method to use this method as a decorator

Return type wrapped

setConfiguration(*sources: Optional[List[str]] = None*)
Method to set the configuration for this PyFiguration object. Configuration is loaded from a YAML or JSON file.

set_configuration(*sources: Optional[List[str]] = None*)
Method to set the configuration for this PyFiguration object. Configuration is loaded from a YAML or JSON file.

1.6.2 Utils

Utility methods that are used by PyFiguration.

pyfiguration.utils.**fromDotNotation**(*field: str, obj: Dict[Any, Any]*) → Any
Method to retrieve a value from the configuration using dot-notation. Dot-notation means nested fields can be accessed by concatenating all the parents and the key with a "." (e.g. db.driver.name).

Parameters

- **field** – The field (in dot-notation) to access
- **obj** – The object to access using dot-notation

Returns The value at the specified key, in the specified obj

Return type value

pyfiguration.utils.**from_dot_notation**(*field: str, obj: Dict[Any, Any]*) → Any
Method to retrieve a value from the configuration using dot-notation. Dot-notation means nested fields can be accessed by concatenating all the parents and the key with a "." (e.g. db.driver.name).

Parameters

- **field** – The field (in dot-notation) to access
- **obj** – The object to access using dot-notation

Returns The value at the specified key, in the specified obj

Return type value

pyfiguration.utils.**mergeDictionaries**(*a: dict*, *b: dict*, *path: Optional[List[str]] = None*) → dict

Merges dictionary b into a, prefering keys in b over keys in a.

Parameters

- **a** – The destination dictionary
- **b** – The source dictionary
- **path** – The full path in the destination dictionary (for recursion)

Returns The merged dictionaries

Return type merged

pyfiguration.utils.**merge_dictionaries**(*a: dict*, *b: dict*, *path: Optional[List[str]] = None*) → dict

Merges dictionary b into a, prefering keys in b over keys in a.

Parameters

- **a** – The destination dictionary
- **b** – The source dictionary
- **path** – The full path in the destination dictionary (for recursion)

Returns The merged dictionaries

Return type merged

1.7 Contributor Covenant Code of Conduct

1.7.1 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, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

1.7.2 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

1.7.3 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.

1.7.4 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.

1.7.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at . 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.

1.7.6 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

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

1.8 License

Python Module Index

p

Index