Type safe config parsing using Python

Python $\geq$ 3.8

Dustin Smith[1]

[1]MLOps Lead
True Digital Group

Table of Contents

Table of Contents

Common Formats

Configuration files should be easy to read, allow comments, and easy deploy/use.

Common Formats

Configuration files should be easy to read, allow comments, and easy deploy/use.

- json

Common Formats

Configuration files should be easy to read, allow comments, and easy deploy/use.

- json
- yaml

Common Formats

Configuration files should be easy to read, allow comments, and easy deploy/use.

- json
- yaml
- ini

Common Formats

Configuration files should be easy to read, allow comments, and easy deploy/use.

- json
- yaml
- ini
- toml

Common Formats

Configuration files should be easy to read, allow comments, and easy deploy/use.

- json
- yaml
- ini
- toml
- pyhocon

PyHOCON

The assumption here will be familiarity with json, yaml, ini, and toml.
What is PyHOCON or HOCON anyways?

PyHOCON

The assumption here will be familiarity with json, yaml, ini, and toml. What is PyHOCON or HOCON anyways?

- HOCON is a json superset: Human-Optimized Config Object Notation.

PyHOCON

The assumption here will be familiarity with json, yaml, ini, and toml. What is PyHOCON or HOCON anyways?

- HOCON is a json superset: Human-Optimized Config Object Notation.
- HOCON has been available in Python for many years but isn't wildly used.

PyHOCON

The assumption here will be familiarity with json, yaml, ini, and toml. What is PyHOCON or HOCON anyways?

- HOCON is a json superset: Human-Optimized Config Object Notation.
- HOCON has been available in Python for many years but isn't wildly used.
- HOCON is used quite a bit in Java and Scala.

PyHOCON

The assumption here will be familiarity with json, yaml, ini, and toml. What is PyHOCON or HOCON anyways?

- HOCON is a json superset: Human-Optimized Config Object Notation.
- HOCON has been available in Python for many years but isn't wildly used.
- HOCON is used quite a bit in Java and Scala.
- pureconfig in Scala allows for easy, type safe configuration parsing.

Table of Contents

Why use dataconf?

If you have ever parsed a config in Python, you will know adding in type checking and checking for missing keys can be overly verbose. However, this has been primarily an issue with Python. In Scala, we can use HOCON, pureconfig, and case classes to easily load in configuration files handling all the necessary checks.

Up until Python 3.7, this wasn't builtin. Since the introduction of dataclasses, we can now have the same easy of use as our friends in JVM based languages.

Config

```
{
    name: Test Model
    date: 20211010
    input-source {
        table-name: db.table
        filter: "par_day between 20210101 and 20210102"
    }
    output-path: /some/dir/folder
}
```

Scala Implementation

```scala
case class InputType(
    tableName: String,
    filter: String
)

case class Params(
    name: String,
    date: Int,
    inputSource: InputType,
    outputPath: String
)

val conf = ConfigSource.file(myPath)
    .loadOrThrow[Params]
```

Python Implementation

```
@dataclass
class InputType:
    table_name: Text
    filter: Text


@dataclass
class Params:
    name: Text
    date: int
    input_source: InputType
    output_path: Text


conf = dataconf.load(my_path, Params)
```

Table of Contents

References

Links to referenced libraries.

- Demo dataconf
- dataconf
- PyHOCON
- HOCON
- pureconfig