py3o.types Documentation

Release 0.1

Jérémie Gavrel

May 25, 2016

Contents

Contents:

CHAPTER 1

Data type configuration for Py3o

1.1 Introduction

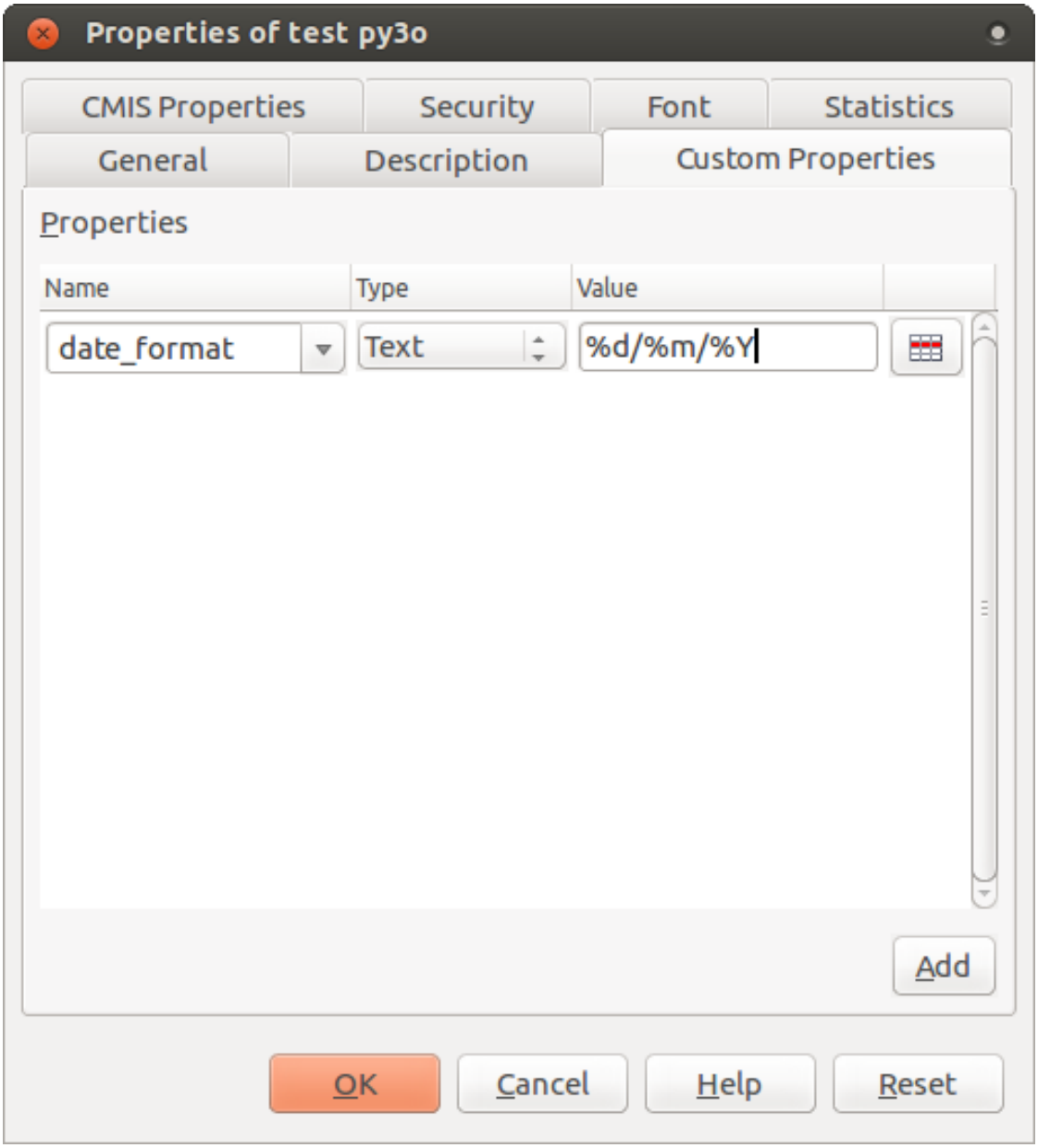
This module implements new, configurable data types that are intended to be used along with Py3o templates.

These data types should be used in the data source dictionary, as a replacement to builtin (int, float...) and standard (date, time...) types. Their rendering is based on a configuration that can be saved and loaded as custom properties inside a ODF file.

1.2 Define the configuration in a ODF file

Configuration keys can be defined directly inside the template, to be loaded by the rendering application.

The custom properties can be set in File > Properties > Custom Properties:



1.3 Use the configuration in Python code

The configuration is represented by a `Py3oTypeConfig` instance. This object can be created by extracting the custom properties from a ODT file, using the class method `Py3oTypeConfig.from_odf_file()`. An optional defaut argument can be specified to provide default values for configuration keys that are not already present in the template:

```
readable_odf_file = open('template.odt', 'rb')
default = {'digit_separator': u"\u00A0"}
config = Py3oTypeConfig.from_odf_file(readable_odf_file, default=default)
```

It is also possible to use the constructor to define a configuration directly:

```
config = Py3oTypeConfig(date_format='%d/%m/%y')
```

The generated, configured data types can then be accessed as attributes of the configuration instance:

```
data_dict = {
    'invoice_date': config.date.strptime('2016-04-20', '%Y-%m-%d'),
    'amount': config.float(430.27),
}
```

1.4 Apply the configuration to a ODF file

Conversely, the configuration parameters can be stored in a ODF file for later extraction by calling the method `Py3oTypeConfig.apply_to_odf_file()`. This method returns a file-like object that contains a copy of the initial ODF file, with the metadata that corresponds to the configuration instance:

```
default_configured_template = config.apply_to_odf_file(readable_odf_file)
requests.post(url, template=default_configurated_template, data=data_dict)
```

A configuration instance can also be applied to another ODF file:

```
origin_odf_file = open('other_template.odt', 'rb')
new_odf_file = open('configured_other_template.odt', 'wb')
config.apply_to_odf_file(origin_odf_file, out_file=new_odf_file)
```

CHAPTER 2

Py3o Data Types

2.1 Integer

See also:

`Py3oInteger`

2.1.1 Configuration Keys

digit_separator The string used to separate digits in the number's string representation.

digit_format (default: 3) The interval between each digit separator.

2.2 Float

See also:

`Py3oFloat`

2.2.1 Configuration Keys

digit_separator The string used to separate digits in the integer part of the number's string representation.

digit_format (default: 3) The interval between each digit separator.

decimal_separator (default: .) The string used to separate the integer part from the decimal part.

2.3 Date

See also:

`Py3oDate`

2.3.1 Configuration Keys

date_format The Python datetime format string that corresponds to the string representation of the date. If not defined, the default format %Y-%m-%d will be used. For more information, see the documentation for `datetime`.

2.4 Time

See also:

`Py3oTime`

2.4.1 Configuration Keys

time_format The Python datetime format string that corresponds to the string representation of the time. If not defined, the default format %H:%M:%S will be used. For more information, see the documentation for `datetime`.

2.5 Datetime

See also:

`Py3oDatetime`

2.5.1 Configuration Keys

datetime_format The Python datetime format string that corresponds to the string representation of the datetime. If not defined and date_format and time_format are present, datetime_format will be deduced from them, separated by a space. If they are undefined as well, the default format %Y-%m-%d %H:%M:%S will be used. For more information, see the documentation for `datetime`.

CHAPTER 3

JSON Encoding and Decoding

3.1 Encoding

`Py3oJSONEncoder` is a JSON encoder that can interpret Py3o type instances. It is used in the same way as any other JSON encoder:

```
data_dict = {
    'invoice_date': config.date.strptime('2016-04-20', '%Y-%m-%d'),
    'amount': config.float(430.27),
}
data_json = Py3oJSONEncoder().encode(data_dict)
```

Note that no configuration will be included in the result. The JSON output is intended to be interpreted in the context of a separate `Py3oTypeConfig` object, presumably extracted from a ODF template.

In addition to the basic JSON types and the Py3o types, the parser will also encode some objects from the Python standard library. They will be encoded in the same way as their equivalent Py3o type.

Standard Class	Py3o Class
`datetime.date`	`Py3oDate`
`datetime.time`	`Py3oTime`
`datetime.datetime`	`Py3oDatetime`

3.2 Decoding

`Py3oJSONDecoder` is a JSON decoder intended to interpret the output of a `Py3oJSONEncoder` instance:

```
decoder = Py3oJSONDecoder(config=config)
data_dict = decoder.decode(data_json)
```

The config argument will be used to provide the appropriate types when decoding JSON data that corresponds to a Py3o object. In addition, integers and floats are decoded as config.integer and config.float instances respectively.

CHAPTER 4

Source code documentation

4.1 Configuration

4.2 Data Types

4.2.1 Numeric Types

4.2.2 Date / Time Types

4.3 JSON Encoding / Decoding

4.3.1 Encoding

4.3.2 Decoding