Openpyxl Templates Documentation

Release 0.1.1

Sverker Sjöberg

Jan 21, 2020

Contents

Openpyxl-templates is an extention to openpyxl which simplifies reading and writing excelfiles by formalizing their structure into templates. The package has two main components:

1. The `TemplatedWorkbook` which describe the excel file
2. The `TemplatedSheets` which describe the individual sheets within the file

The package is build for developers to be able to implement their own templates but also provides one useful templated sheet. The `TableSheet` which makes it significantly easier to read and write data from excel data tabels such as this one:

The TableSheet provides an easy way of defining columns and handles both styling and conversion to and from excel. See quickstart for a demo.

Github https://github.com/SverkerSbrg/openpyxl-templates

Documentation http://openpyxl-templates.readthedocs.io/en/latest/

pypi https://pypi.python.org/pypi/openpyxl-templates

openpyxl https://openpyxl.readthedocs.io/en/default/

If you have any questions or ideas regarding the package feel free to reach out to me via GitHub.

> **Warning:** This package is still in beta. The api may still be subject to change and the documentation is patchy.

CHAPTER 1

Table of contents

1.1 Welcome to openpyxl-templates!

Openpyxl-templates is an extention to openpyxl which simplifies reading and writing excelfiles by formalizing their structure into templates. The package has two main components:

1. The `TemplatedWorkbook` which describe the excel file
2. The `TemplatedSheets` which describe the individual sheets within the file

The package is build for developers to be able to implement their own templates but also provides one useful templated sheet. The `TableSheet` which makes it significantly easier to read and write data from excel data tabels such as this one:

The TableSheet provides an easy way of defining columns and handles both styling and conversion to and from excel. See quickstart for a demo.

Github https://github.com/SverkerSbrg/openpyxl-templates

Documentation http://openpyxl-templates.readthedocs.io/en/latest/

pypi https://pypi.python.org/pypi/openpyxl-templates

openpyxl https://openpyxl.readthedocs.io/en/default/

If you have any questions or ideas regarding the package feel free to reach out to me via GitHub.

Warning: This package is still in beta. The api may still be subject to change and the documentation is patchy.

1.2 Quick start

1.2.1 Installation

Install openpyxl-templates using pypi:

```
pip install openpyxl-templates
```

1.2.2 Creating your template

The first we create our TemplatedWorkbook, which describes the structure of our file using TemplatedWorksheets. This template can then be used for both creating new files or reading existing ones. Below is an example using the TableSheet (a TemplatedWorksheet) to describe a excel file of people and their favorite fruits.

```
from datetime import date
from enum import Enum
from openpyxl_templates import TemplatedWorkbook
from openpyxl_templates.table_sheet import TableSheet
from openpyxl_templates.table_sheet.columns import CharColumn, ChoiceColumn,␣
↪DateColumn


class Fruits(Enum):
    apple = 1
    banana = 2
    orange = 3


class PersonSheet(TableSheet):
    first_name = CharColumn()
    last_name = CharColumn()
    date_of_birth = DateColumn()
    favorite_fruit = ChoiceColumn(choices=(
        (Fruits.apple, "Apple"),
        (Fruits.banana, "Banana"),
        (Fruits.orange, "Orange"),
    ))


class PersonsWorkbook(TemplatedWorkbook):
    persons = PersonSheet()
```

1.2.3 Writing

To write create an instance of your templated workbook, supply data to the sheets and save to a file.

```
wb = PersonsWorkbook()
wb.persons.write(
    title="List of fruit lovers",
    objects=(
        ("John", "Doe", date(year=1992, month=7, day=17), Fruits.banana),
        ("Jane", "Doe", date(year=1986, month=3, day=2), Fruits.apple),
    )
)
```

The TableSheet in this case will handle all formatting and produce the following sheet.

	A	B	C	D
1	List of fruit lovers			
2	first_name	last_name	date_of_birth	favorite_fruit
3	John	Doe	1992-07-17	banana
4	Jane	Doe	1986-03-02	Apple
5				

1.2.4 Reading

To utilize the openpyxl-templates to read from an existing excel file, initialize your TemplatedWorkbook with a file (or a path to a file). Using the read method or simply itterating over a sheet will give you access to the data as namedtupels.

```
wb = PersonsWorkbook("fruit_lovers.xlsx")

for person in wb.persons:
    print(person.first_name, person.last_name, person.favorite_fruit)
```

1.3 TemplatedWorkbook

The `TemplatedWorkbook` is our representation of the excel file and describes the file as a whole. To create a TemplatedWorkbook extend the base class and declare which sheets it includes using TemplatedWorksheets.

```
from openpyxl_templates import TemplatedWorkbook, TemplatedWorksheet


class DemoTemplatedWorkbook(TemplatedWorkbook):
    sheet1 = TemplatedWorksheet()
    sheet2 = TemplatedWorksheet()
```

To use your template to generate new excel files simply create an instance...

```
templated_workbook = DemoTemplatedWorkbook()
```

... or provide it with a filename (or a file) to read an existing one.

```
templated_workbook = DemoTemplatedWorkbook(file="my_excel.xlsx")
```

The TemplatedWorkbook will find all sheets which correspond to a TemplatedWorksheet. Once identified the TemplatedWorksheets can be used to interact with the underlying excel sheets. The matching is done based on the sheetname.

The TemplatedWorkbook keeps track of the declaration order of the TemplatedWorksheets which enables it to make sure the the sheets are always in the correct order once the file has been saved. The identified sheets can also be iterated as illustrated below.

```
for templated_worksheet in templated_workbook.templated_sheets:
    print(templated_worksheet.sheetname)
```

To save the workbook simply call the save method and provide a filename

```
templated_workbook.save("my_excel.xlsx")
```

1.4 TemplatedWorksheet

The TemplatedWorksheet describes a a sheet in an excel file and is the bare bone used for building useful sheet templates such as the *TableSheet*. A TemplatedWorksheet is defined by following attributes:

- It's `sheetname` which is used for identifying the sheets in the excel file
- It's `read()` method which when implemented should return the relevant data contained in the sheet
- It's `write(data)` method which when implemented should write the provided data to the sheet and make sure that it is properly formatted

The TemplatedWorksheet will handle managing the openpyxl worksheet so you do not have to worry about whether the sheet is created or not before you start writing.

To create a TemplatedWorksheet you should implement the read and write method. We'll demonstrate this by creating a DictSheet a TemplatedWorksheet which reads and writes simple key value pairs in a python dict.

```
class DictSheet(TemplatedWorksheet):
    def write(self, data):
        worksheet = self.worksheet

        for item in data.items():
            worksheet.append(list(item))

    def read(self):
        worksheet = self.worksheet
        data = {}

        for row in worksheet.rows:
            data[row[0].value] = row[1].value

        return data
```

We can now add our DictSheet to a TemplatedWorkbook and use it to create an excel file.

```
class DictWorkbook(TemplatedWorkbook):
    dict_sheet = DictSheet(sheetname="dict_sheet")


workbook = DictWorkbook()

workbook.dict_sheet.write({
    "key1": "value1",
```

(continues on next page)

(continued from previous page)

```
    "key2": "value2",
    "key3": "value3",
})

workbook.save("key_value_pairs.xlsx")
```

We can use the same TemplatedWorkbook to read the data from the file we just created.

```
workbook = DictWorkbook(join(dirname(__file__), "key_value_pairs.xlsx"))

print(workbook.dict_sheet.read())
```

Have a look at the *TableSheet* for a more advanced example. It includes type handling and plenty of styling.

1.5 Working with styles

Styling in openpyxl-templates is entirely reliant on the `NamedStyle` provided by openpyxl. Using NamedStyles offers significantly better performance compared to styling each individual cells, as well as the benefits of making the styles avaliable in the produced excel file.

To manage all named styles within a TemplatedWorkbook openpyxl-templates uses the `StyleSet` which is a dictionary like collection of NamedStyles. Using the StyleSet when creating templates is entierly optional but offers several advantages:

- Using a common colleciton of styles for all TemplatedSheets makes it easier to avoid duplicated styles and name conflicts
- The StyleSet accepts `ExtendedStyle` s as well as NamedStyles which enables inheritance between styles within the StyleSet
- Styles will only be added to the excel file when they are needed allowing the developer to use a single StyleSet for multiple templates without having to worry about unused styles being included in the excel file.
- Using NamedStyles offers significantly better performance compared to styling each cell individually when writing a large amout of data

1.5.1 Creating a StyleSet

To create a StyleSet simply pass NamedStyles or ExtendedStyles as arguments.

```
demo_style = StyleSet(
    NamedStyle(
        name="Default",
        font=Font(
            name="Arial",
            size=12
        )
    ),
    NamedStyle(
        name="Header",
        font=Font(
            name="Arial",
            size=12,
            bold=True,
```

(continues on next page)

(continued from previous page)

```
        ),
    )
)
```

If we want to avoid having to redeclare the font we could refactor the above example using an ExtendedStyle

```
demo_style = StyleSet(
    NamedStyle(
        name="Default",
        font=Font(
            name="Arial",
            size=12
        )
    ),
    ExtendedStyle(
        base="Default",  # Reference to the style defined above
        name="Header",
        font={
            "bold": True,
        }
    )
)
```

The ExtendedStyle can be viewed as a NamedStyle factory. It accepts the same arguments as a NamedStyle with the addition of the `base` which is the name of the intended parent. The StyleSet will make sure that the parent style is found irregardless of declaration order.

The font, border, alignment and fill arguments of the NamedStyle are supplied as objects which (since they have default values) prevent inheritance. To circomvent this limitation you can supply the kwarg dicts to the extended style instead of the object itself, as we have done in the example above. If the openpyxl class is used instead of kwargs, inheritance will be broken

```
bad_example = StyleSet(
    NamedStyle(
        name="Default",
        font=Font(
            name="Arial",
            size=12
        )
    ),
    ExtendedStyle(
        base="Default",
        name="Header",
        font=Font(
            # Openpyxl will set name="Calibri" by default which will override name=
↪"Arial and break inheritance.
            bold=True
        )
    )
```

if a name is declared multiple times the last declaration will take precedence making it easy to modify an existing StyleSet. One common usage for this is to modify the DefaultStyleSet. Which is demonstrated ModifyDefaultStyleSet.

1.5.2 Accessing styles

When you need to use a specific style just retrive it by name

TODO

1.5.3 DefaultStyleSet

Openpyxl-templates includes a `DefaultStyleSet` which is used as a fallback for all TemplatedWorkbook. Many of the styles it declares (or their names) are required by the `TableSheet`. The DefaultStyleSet is defined like this

```
            raise ValueError("StyleSet can only handle NamedStyles")

        if style.name in self:
            raise ValueError("Style already exists")

        self._styles[style.name] = style
        return style

    @property
    def names(self):
        return tuple(style.name for style in self._styles.values())

    def extend(self, extended_style):
        return self._add(extended_style)

    def style_cell(self, cell, style):
        if type(style) in (NamedStyle, ExtendedStyle):
            if style.name not in self:
                named_style = self._add(style)
            else:
                named_style = self[style.name]
        else:
            named_style = self[style]

        # print("    style cell", named_style.name, type(style), style.name if
↪type(style)!= str else "")
        cell.style = named_style


class DefaultStyleSet(StyleSet):
    def __init__(self, *styles):
        super(DefaultStyleSet, self).__init__(
            NamedStyle(
                name="Default",
                alignment=Alignment(vertical="top")
            ),
            ExtendedStyle(
                base="Default",
                name="Empty",
            ),
            ExtendedStyle(
                base="Empty",
                name="Title",
                font={"size": 20}
            ),
            ExtendedStyle(
                base="Empty",
                name="Description",
                font={"color": "FF777777"},
                alignment={"wrap_text": True}
```

(continues on next page)

(continued from previous page)

```
            ),
            ExtendedStyle(
                base="Default",
                name="Header",
                font={"bold": True, "color": "FFFFFFFF"}, fill=SolidFill(DEFAULT_
↪ACCENT_COLOR)
            ),
            ExtendedStyle(
                base="Header",
                name="Header, center",
                alignment={"horizontal": "center"}
            ),
            ExtendedStyle(
                base="Default",
                name="Row"
            ),
            ExtendedStyle(
                base="Row",
                name="Row, string",
                number_format="@",
            ),
            ExtendedStyle(
                base="Row",
```

If you which to modify the DefaultStyle you can easily replace any or all of the styles it contains by passing them as arguments to the constructor. Below we utilize the heavy usage of ExtendedStyles in the DefaultStyleSet to change of all styles.

```
    NamedStyle(  # Replace the existing "Default" font with a new one.
        name="Default",
        font=Font(
            name="Arial",
        )
    )
)
```

1.6 TableSheet

The `TableSheet` is a `TemplatedWorksheet` making it easy for reading and write sheets with excel Data Tables. It is made up of an ordered set of typed columns which support when converting to and from Excel. Read more about what the columns do *here*.

1.6.1 Elements of the TableSheet

The `TableSheet` recognizes the following elements:

- **Title** (optional) - A bold header for the Data Table
- **Description** (optional) - A smaller description intended for simple instructions
- **Columns** - The Columns in the datatable, which in turn is made up of **headers** and **rows**

```
from itertools import repeat
from openpyxl_templates import TemplatedWorkbook
from openpyxl_templates.table_sheet import TableSheet
from openpyxl_templates.table_sheet.columns import CharColumn


class TableSheetElements(TableSheet):
    column1 = CharColumn(header="Header 1")
    column2 = CharColumn(header="Header 2")
    column3 = CharColumn(header="Header 3")
    column4 = CharColumn(header="Header 4")


class TableSheetElementsWorkook(TemplatedWorkbook):
    table_sheet_elements = TableSheetElements()


wb = TableSheetElementsWorkook()
wb.table_sheet_elements.write(
    title="Title",
    description="This is the description, it can be a couple of sentences long.",
    objects=(tuple(repeat("Row %d" % i, times=4)) for i in range(1, 4))
)
wb.save("table_sheet_elements.xlsx")
```

	A	B	C	D
1	Title			
2	This is the description, it can be a couple of sentences long.			
3	Header 1	Header 2	Header 3	Header 4
4	Row 1	Row 1	Row 1	Row 1
5	Row 2	Row 2	Row 2	Row 2
6	Row 3	Row 3	Row 3	Row 3
7				

The TableSheet does not support reading the title or description.

1.6.2 Creating the TableSheet

A `TableSheet` is created by extending the TableSheet class, declaring columns and optionally changing styling and other settings. Once the TableSheet class has been created an instance of this class is be supplied to the `TemplatedWorkbook`.

Declaring columns

The columns are declared as class variables on a TableSheet which will identify and register the columns (in order of declaration). The columns are avaliable under the `columns` attribute.

A TableSheet must always have atleast one `TableColumn`.

```
from openpyxl_templates.table_sheet import TableSheet
from openpyxl_templates.table_sheet.columns import CharColumn, IntColumn
```

(continues on next page)

(continued from previous page)

```
class DemoTableSheet(TableSheet):
    column1 = CharColumn()
    column2 = IntColumn()
```

The column declaration supports inheritance, the following declaration is perfectly legal.

```
class ExtendedDemoTableSheet(DemoTableSheet):
    column3 = CharColumn()
```

Note that the columns of the parent class are always considered to have been declared before the columns of the child.

All columns must have a header and there must not be any duplicated headers within the same sheet. The TableSheet will automatically use the attribute name used when declaring the column as header.

```
class DemoTableSheet(TableSheet):
    column1 = CharColumn(header="Header 1")
    column2 = IntColumn()  # The header of column2 will be set automatically to
↪"column2"
```

An instance of a column should never be used on multiple sheets.

TODO: Dynamic columns

TODO: Describe the ability to pass additional columns to __init__ via `columns=[...]`

1.6.3 Writing

Simple usage

Writing is done by an iterable of objects to the write function and optionally a title and/or description. The write function will then:

- Prepare the workbook by registering all required styles, data validation etc.
- Write title, and description if they are supplied
- Create the headers and rows
- Apply sheet level formatting such as creating the Data Table and setting the freeze pane

Writing will always recreate the entire sheet from scratch, so any preexisting data will be lost. If you want to preserve your data you could read existing rows and combine them with the new data.

```
class DemoTableSheet(TableSheet):
    column1 = CharColumn()
    column2 = IntColumn()


class DemoTemplatedWorksheet(TemplatedWorkbook):
    demo_sheet1 = DemoTableSheet()
    demo_sheet2 = DemoTableSheet()

wb = DemoTemplatedWorksheet()
```

(continues on next page)

(continued from previous page)

```
wb.demo_sheet1.write(
    objects=(
        ("Row 1", 1),
        ("Row 2", 2),
        ("Row 3", 3),
    ),
    title="The first sheet"
)
wb.demo_sheet2.write(
    objects=(
        ("Row 1", 1),
        ("Row 2", 2),
        ("Row 3", 3),
    ),
    title="The second sheet",
    description="Lorem ipsum dolor sit amet, consectetur adipiscing elit. In euismod,␣
↪sem eu."
)
wb.save("read_write.xlsx")
```

Using objects

The write accepts rows iterable containing tuples or list as in the example above. If an other type is encountered the columns will try to get the attribute directly from the object using `getattr(object, column.object_attribute)`. The object_attribute can be defined explicitly and will default to the attribute name used when adding the column to the sheet.

```
class DemoObject():
    def __init__(self, column1, column2):
        self.column1 = column1
        self.column2 = column2


wb = DemoTemplatedWorksheet()
wb.demo_sheet1.write(
    objects=(
        DemoObject("Row 1", 1),
        DemoObject("Row 2", 2),
        DemoObject("Row 3", 3),
    )
)
```

Styling

The TableSheet has two style attributes:

- `tile_style` - Name of the style to be used for the title, defaults to *"Title"*
- `description_style` - Name of the style to be used for the description, defaults to *"Description"*

```
class PrettyDemoSheet(TableSheet):
    def __init__(self):
        super().__init__(
            title_style="Bold & red, title",
```

(continues on next page)

(continued from previous page)

```
            description_style="Extra tiny, description"
        )

    column1 = CharColumn()
    column2 = IntColumn()
```

Styling of columns done on the columns themselves.

Make sure that the styles referenced are available either in the workbook or in the `StyleSet` of the `TemplatedWorkbook`. Read more about styling styling.

Additional settings

The write behaviour of the TableSheet can be modified with the following settings:

- `format_as_table` - Controlling whether the TableSheet will format the output as a DataTable, defaults to *True*
- `freeze_pane` - Controlling whether the TableSheet will utilize the freeze pane feature, defaults to *True*
- `hide_excess_columns` - When enabled the TableSheet will hide all columns not used by columns, defaults to *True*

1.6.4 Reading

Simple usage

The `read` method does two things. First it will verify the format of the file by looking for the header row. If the headers cannot be found a en exception will be raised. Once the headers has been found all subsequent rows in the excel will be treated as data and parsed to namedtuples automatically after the columns has transformed the data from excel to python.

```
wb = DemoTemplatedWorksheet("read_write.xlsx")
for row in wb.demo_sheet1.read():
    print(row)
```

Iterate directly

The TableSheet can also be used as an iterator directly

```
for row in wb.demo_sheet2:
    print(row)
```

Exception handling

The way the TableSheet handles exceptions can be configured by setting the `exception_policy`. It can be set on the TableSh

- `RaiseCellException` (default) - All exceptions will be raised when encountered
- `RaiseRowException` - Cell level exceptions such as type errors, in the same row will be collected and raised as a RowException

- `RaiseSheetException` - All row exceptions will be collected and raised once reading has finished. So that all valid rows will be read, and all exceptions will be recorded.
- `IgnoreRow` - Invalid rows will be ignored

The policy only applies to exceptions occuring when reading rows. Exceptions such as `HeadersNotFound` will be raised irregardless.

Reading without looking for headers

Looking for headers can be disabled by setting `look_for_headers` to *False* or passing it as a named argument directly to the read function. When this is done the TableSheet will start looking for valid rows at once. This will most likely cause an exception if the title, description or header row is present since they will be treated as rows.

1.6.5 Customization

The TableSheet is built with customization in mind. If you want your table to yield something else then a `namedtuple` for each row. It is easy to achieve by overriding the `create_object` method.

```
class IceCream:
    def __init__(self, name, description, flavour, color, price):
        self.name = name
        self.description = description
        self.flavour = flavour
        self.color = color
        self.price = price


class IceCreamSheet(TableSheet):
    name = CharColumn()
    description = TextColumn(width=32)
    flavour = CharColumn()
    color = CharColumn()
    price = FloatColumn()

    def create_object(self, row_number, **data):
        return IceCream(**data)
```

Feel free to explore the source code for additional possibilities. If you are missing hook or add a feature useful for others, feel free to submit a push request.

1.7 TableColumn