preppy Documentation

Release 2.5.0

ReportLab

Dec 25, 2017

Contents

1 Why another templating system? 3

2 Quick Start 5

3 Template syntax 7

4 Module import options 11
- 4.1 File system semantics . . . 11
- 4.2 Import semantics . . . 11

5 Executing the template 13
- 5.1 Quoting functions . . . 14
- 5.2 xmlQuote and SafeString . . . 14

6 Controlling compilation 15

7 Command line tools 17

8 But how do I. . . ? 19
- 8.1 Include . . . 19
- 8.2 Automatic escaping . . . 19
- 8.3 Filters . . . 20
- 8.4 Block inheritance . . . 20

9 Support 21

10 Credits 23

Preppy is ReportLab's templating system. It was developed in late 2000 and has been in continual production use since then. It is open source (BSD-license).

The key features are:

- *small*. Preppy is a single Python module. If you want a templating system 'in the box', it's easy to include it in your project
- *easy to learn*. It takes about one minute to scan all the features
- *just Python*. We have not invented another language, and if you want to do something - includes, quoting, filters - you just use Python
- *compiled to bytecode*: a .prep file gets compiled to a Python function in a .pyc file
- *easy to debug*: preppy generates proper Python exceptions, with the correct line numbers for the .prep file. You can follow tracebacks from Python script to Preppy template and back, through multiple includes
- *easy to type and read*. We've been using `{{this}}` syntax since well before Django was thought of
- *8-bit safe*: it makes no assumption that you are generating markup and does nothing unexpected with whitespace; you could use it to generate images or binary files if you wanted to.

CHAPTER 1

Why another templating system?

Since there are dozens of templating systems out there, it's worth explaining why this one exists.

We developed preppy in 2000. At the time, every python developer and his dog was busy inventing their own web framework, usually including a templating system for web development.

Most of these involved coming up with an unreliable parser, and an incomplete 'little language' for embedding in markup. We took the view that we already had a perfectly good little language - Python - and there was no need to invent a new one. Preppy has met all of our needs well, with minimal changes.

Our main product is a markup-based system for creating PDF documents, Report Markup Language. When we create reports for people, we use a template to generate an RML file, in the same way that most people make web pages. We need a templating system for our own demos and tutorials, as well as solutions we build, which is 'in the box' and easy to understand. We also build web applications, and when we want something strict and minimal, we use preppy.

Moving on, most major web frameworks have settled on a templating system. The most popular ones, such as Django or Jinja, are very full-featured. They have focused on creating a new language for web designers. They are getting fairly big and complex and they impose their own learning curve. They also result in template programming becoming a different skill to the main application, and sometimes with a different philosophy. For example, in the Django world, templates are supposed to be coded by designers who lack programming skills, and the system is deliberately forgiving of errors. In our experience, this laxity leads to more subtle problems a few days later on in development.

Within ReportLab we take the view that a template is code; it is expected to be correct, and to raise errors when misspelled or nonexistent variables are used. It is far easier to create a robust system when the templates use the same language as the rest of the application, and are debugged in exactly the same way, as the application code.

Later on, we'll show how many features of high-end templating systems are handled with short Python expressions.

CHAPTER 2

Quick Start

Preppy can be installed from `easy_install`, `pip`, by downloading and using ‘‘setup.py install‘, by cloning from bitbucket. Or, because it’s just a single Python module, you can grab it and put it on your path, or check it into your application.

If you’re too busy to read the whole page, the next few lines should get you going. First, get preppy on your path through one of the above methods.

Place your template code in a file which, by convention, ends in .prep. Here is an example:

```
{{def(name, sex)}}
<html><head><title>{{name}}</title></head><body>
hello my name is {{name}} and I am
{{if sex=="f":}} a gal
{{elif sex=="m":}} a guy
{{else}} neuter {{endif}}
</body></html>
```

If this is in a file called, say, `template.prep`, you can invoke it like this:

```
mymodule = preppy.getModule('template.prep')
name = 'fred'
sex = 'm'
html = mymodule.get(name, sex)
```

`getModule` returns a Python module object. On first call, or when templates are edited, this will generate a .pyc file, just as a Python module does (assuming you have write access to the disk). This module contains a function, `get`, which implements your template. The function accepts the parameters which you specified at the top of the prep file, and returns a (non-Unicode, 8-bit) string.

This is preppy’s key design idea: we try to make a preppy template as much like a python function as possible.

CHAPTER 3

Template syntax

Preppy has a small number of tags, enclosed in pairs of curly braces:

{{def(YOUR_ARGUMENT_LIST}}

This is a special construct which must be placed near the top of a .prep file. If used, it should be the first preppy directive. It allows you to explicitly declare the parameters which will be passed in. You can do most of the things you can do with a Python function: positional arguments, keyword arguments, and `\*args` and `\*\*kwds`. The only difference is that there is no function name. We compile this into a Python function named *`get()`*.

There is an older way of using preppy, which omits this declaration, and lets you pass an arbitrary dictionary of arguments to the template, like most other templating systems. This should be regarded as deprecated and will be omitted in version 2, both for usability reasons, and for technical reasons to do with bytecode generation as we move to Python 3.x. I

We do it this way because, in over a decade working with preppy, we have found that it's very valuable to have an explicit declaration at the top of a .prep file telling you what variables are in the namespace. Otherwise, on large projects, one ends up constantly referring to the calling code in another module.

{{expression}}

Any Python expression will be evaluated in the current namespace, and thence converted to a string representation. Examples:

```
The total is {{2+2}}

{{range(10)}}

Dear {{client.name}},
```

By default, the expression is converted by Python's `str()` function. So the python value `None` will appear as the text `None` in the output, and any non-ascii characters in a string will trigger an exception. In each application, you have the option to define your own **quoting functions** to use instead, which we discuss below.

{{eval}}

This is exactly equivalent to `{{expression}}`, but is useful when you have a long Python expression which spans several lines, or the extra curly braces on the same line as the expression harm readability. For example:

```
{{eval}}
a_complex("and", "very", "verbose", function="call")
{{endeval}}
```

{{script}}....{{endscript}}

Multiple or single lines of python scripts may be embedded within `{{script}}...{{endscript}}` tags. Examples:

```
{{script}}import urllib2{{endscript}}

{{script}}
cur = conn.cursor()
cur.execute('select * from some_table')
data = cur.fetchall()
{{endscript}}
```

For `expression`, `eval`, and `script`, any newlines in the code text will be automatically indented to the proper indentation level for the run() module at that insertion point. You may therefore indent your code block to match the indentation level of any HTML/XML it is embedded in. This is only a concern for triple quoted strings. If this may be an issue, don't use triple quoted strings in preppy source. Instead of:

```
x = """
a string
"""
```

use:

```
x = ("\n"
"\ta string\n"
)
```

or something similar.

It is generally bad practice to have too much in script tags. If you find yourself writing long script sections to fetch and prepare data or performing calculations, it is much better to place those things in a separate python module, import it within the template, and call those functions in one line.

{{if EXPR}}...{{elif EXPR}}...{{else}}...{{endif}}

The `*{{if}}*` statement does exactly what Python's *if* statement does. You may optionally use multiple *elif* clauses and one *else* clause. The final colon after each clause ("*else:*") is optional.

{{for EXPR}}...{{else}}...{{endfor}}

This implements a for loop in preppy source. The EXPR should follow normal python conventions for python for loops. The resulting python code is roughly:

```
for EXPR:
    interpretation_of(block)
else:
    no break exit
```

An example:

```
{{for (id, name) in dataset}}
    <tr><td>{{id}}</td><td>{{name}}</td>
{{endfor}}
```

{{while CONDITION}}...{{else}}...{{endwhile}}

This implements a *while* loop in preppy source. The condition should be a python expression. The resulting python code is roughly:

```
while CONDITION:
    interpretation_of(block)
else:
    ....
```

{{try}}...{{except X}}...{{else}}...{{finally}}...{{endtry}}

The normal python *try* in preppy form. The the else clause is accepted only if an except clause is seen.

{{with open('aaa') as x}}...{{endwith}}

As in python the contained block knows about x and handles finalization etc etc.

{{raise Exception}}

This allows raising an exception without using {{script}} {{endscript}}.

{{continue}}

Continues to next iteration without requiring {{script}} {{endscript}}. Only allowed inside a loop.

{{break}}

Breaks from a loop without requiring {{script}} {{endscript}}. Only allowed inside a loop.

{{import module}} or {{import module as x}} or {{from module import x}} etc.

The normal python import statements.

{{assert condition, "....message"}}

The python assert statement.

{{def NAME (ARGDEFS)}} preppy stuff {{enddef}}

Define a template function. This defines a function that encapsulates some preppy statements. The function may be called like any other known function, but unlike normal functions the preppy literals and *{{EXPR}}* expressions are renderd at the point of call followed by any returned values. Ass a concession to the template nature of the definition, {{enddef}} may be considered to be a {{return ''}}. The *{{return EXPR}}* is only legal between *{{def...}}* and *{{enddef}}*.

Example:

{{def weekday(d)}} {{script}}days=['','Sun','Mon','Tue','Wed','Thu','Fri','Sat']{{endscript}}
Today is {{days[d]}}day.

{{enddef}}

CHAPTER 4

Module import options

There are two ways to load a preppy module into memory. We refer to these as 'file system semantics' and 'import semantics'.

4.1 File system semantics

The file system method is implemented by *getModule()*:

getModule(*name*, *directory="."*, *source_extension=".prep"*, *verbose=0*, *savefile=None*, *sourcetext=None*, *savePy=0*, *force=0*, *savePyc=1*, *importModule=1*, *_globals=None*)

This loads your template, which is a Python module object.

There is no predefined search path or list of template directories; if you want to implement your own template search path, you're free to write a function of your own which wraps *getModule()*.

name can be a relative or full path. Commonly in web applications we work out the full path to the template directory and do everything with the *name* argument:

```
m = getModule(os.path.join(PROJECT_DIR, 'myapp/templates/template.prep'))
```

Alternatively, you can pass the module name and directory separately if you prefer:

```
m = getModule('template', directory='TEMPLATE_DIR'))
```

Finally, you can supply literal source if desired. This is primarily to help us in writing test cases; if you're doing it for real, you are probably either doing something brilliant or stupid ;-)

The resulting module should be treated just like a Python module: import it, keep it around, and call it many times.

4.2 Import semantics

In an attempt to make preppy templates even more like Python code, we have also provided an **import hook**.

installImporter()

Let's say you have a template called 'mytemplate.prep', on the current Python path. With the import hook, you can import your template instead of loading it with *getModule()*:

```
import preppy
preppy.installImporter()
...
import mytemplate
html = mytemplate.getOutput(namespace)
```

installImporter() only needs to be called once in your program.

uninstallImporter()

This does what it says. You probably don't need to call it, unless you have a reason to remove import hooks, or you're working on preppy's test suite.

CHAPTER 5

Executing the template

We provide two ways to execute a template and generate output. The preferred, new approach is as we demonstrated at the top of the page: you pass the same arguments through that are specified at the top of the .prep file.

get (*arg1*, *arg2*, *etc*, *key1=value1*, *key2=value2*, *etc=etc*, *quoteFunc=str*)

For example, if the template starts with this:

```
{{def(name, sex)}}
```

then you would call it with:

```
output = mymodule.get(name, sex)
```

The return value is always an 8-bit string (which will become a *bytes* object in Python 3.x).

The older approach is to pass in an arbitrary-sized dictionary, as done by most other templating systems (e.g. django, jinja). In this case, you must NOT define a *def* declaration at the top of the module. This uses a function *getOutput()* defined as follows:

getOutput (*dictionary*, *quoteFunc=str*)

This would be used as follows:

namespace = {'name':'fred','age':42, 'sex':'m'} html = template.getOutput(namespace)

In both cases, the *quoteFunc* argument lets you control how non-text variables are displayed. In a typical web project, you will want to supply your own quoting function, to do things like escaping '&' as '&'. This is covered in detail below.

If you prefer a streaming or file-based approach, you can use the *run()* function in the old style approach:

run (*dictionary*, *__write__=None*, *quoteFunc=str*, *outputfile=None*, *code=__code__*)

You may either supply a function callback to *__write__*, which will be called repeatedly with the generated text; or a file-like object to *outputfile*.

5.1 Quoting functions

By default, preppy will use Python's *str* function to display any expression. This causes a problem in the markup world, where output us usually utf-8 encoded. The *quoteFunc* argument lets you pass in an alternative function which will be used to quote any output.

If you are generating XML or HTML (which most people are), and you have a database field or variable containing one of the characters '<', '>' or '&', then it is easy to generate invalid markup.:

```
<p>{{companyName}}</p>   -->  <p>Johnson & Johnson</p>
```

An expression like the one below will fail on the first foreign accent in a name, raising a traceback, because Python can't convert this to ASCII:

```
<p>{{client.surname}}</p>
```

A third use is to identify and remove javascript or SQL snippets, which might have been passed in by a hacker.

In general, you should decide on the quoting function you need, and pass it in when templates are called.

5.2 xmlQuote and SafeString

We have provided one such function inside preppy.py, *stdQuote*, which is useful for XML and HTML generation. It behaves as follows:

- 8 bit strings will be xml-escaped.
- Any null value will produce no output. This might be useful if you are displaying a lot of numbers in table cells and don't want the word 'None' appearing everywhere.
- Anything you DON'T want to be quoted can be wrapped in a special SafeString or SafeUnicode class, and it won't be quoted.
- Anything else will be converted to a Unicode string representation (just in case the string representation contains non-ascii characters), and then encoded as utf8, then escaped.

You would use this as follows:

```
output = mymodule.get(name, sex, quoteFunc=preppy.stdQuote)
```

If you are using this and you want to output a string which preppy should NOT quote (perhaps because it comes from some utility which already generates correct, escaped HTML), wrap it in the *SafeString* or *SafeUnicode* class, and it will not be escaped:

```
<h1>My markup</h1>
{{script}}from preppy import SafeString{{endscript}}}}

<p>{{SafeString(my_already_escaped_text)}}</p>
```

Note that SafeString and SafeUnicode are string wrappers designed to work with the *stdQuote* function, and have no useful behaviour in other contexts.

CHAPTER 6

Controlling compilation

In normal use, assuming the current user has write access to the file system, preppy will function like Python: edit your template, run your program, and the calls to getModule will trigger a recompile. However, if you want to control this for your own application (for example, in deployment scripts), three functions are provided.

compileModule(*fn*, *savePy=0*, *force=0*, *verbose=1*, *importModule=1*)

compileModules(*pattern*, *savePy=0*, *force=0*, *verbose=1*)

compileDir(*dirName*, *pattern="'*'.prep"*, *recursive=1*, *savePy=0*, *force=0*, *verbose=1*)

The last one works recursively, so is convenient for compiling all .prep files within a project.

CHAPTER 7

Command line tools

preppy can also function as a script to let you control compilation. In some web frameworks (including CGI-based ones), the application runs as a restricted user, and it is important to precompile all templates and python modules during deployment.

The command line interface lets you test, compile and clean up. **We expect to change this to use the more modern *optparse* module soon**:

```
preppy modulename [arg1=value1, arg2=value2.....]
   - shorthand for 'preppy run ...', see below.

preppy run modulename [arg1=value1, arg2=value2.....]
   - runs the module, optionally with arguments.  e.g.
     preppy.py flintstone.prep name=fred sex=m

preppy.py compile [-f] [-v] [-p] module1[.prep] module2[.prep] module3 ...
   - compiles explicit modules

preppy.py compile [-f] [-v] [-p] dirname1  dirname2 ...
   - compiles all prep files in directory recursively

preppy.py clean dirname1 dirname2 ...19
   - removes any py or pyc files created from past compilations
```

CHAPTER 8

But how do I. . . ?

People with experience of bigger templating systems typically wonder where their beloved features are. The answer is almost always that you can do it with Python.

Here are some of the common 'missing features':

8.1 Include

How do you include other content? With a Python function or method call.

If you want to include totally static content, it's as easy as this:

```
<h1>Appendix</h1>
{{open('templates/appendix.html').read()}}
```

If you want to call other templates, then import them at the top of the module in a script tag, and call them inline:

```
{{script}}
appendix = preppy.getModule('appendix.prep')
{{endscript}}

<h1>Appendix</h1>
{{appendix.get(data, options)}}
```

Being able to see what is passed in when you are editing the outer template, as well as what is expected in the inner template, turns out to be a huge advantage in maintenance when you are dealing with large systems of nested templates.

8.2 Automatic escaping

Many systems can escape all expressions output into HTML as a security measure. Some go further and try to remove Javascript. Preppy solves this by letting you pass in your own quote function.

In systems which do this, they commonly require an extra construct to mark some expressions as 'safe', and not to be escaped. This can be accomplished by having a string subclass, and having your quote function recognise and pass it through. See the *stdQuote* function above

8.3 Filters

Django has a nice syntax for filters - functions which tidy up output:

```
{{number | floatformat}}
```

Our approach is to have functions with short names, and avoid introducing extra syntax. This is very slightly more verbose. For example, if a template had to display many values in pounds sterling, we could write a function *fmtPounds* which adds the pound sign, formats with commas every thousand and two decimal places. These functions can also be set to output an empty string for None or missing values, to set a class on the output for negative number display, or whatever you require.

We then display a value like this:

```
<td>{{fmtPounds(total)}}</td>
```

This approach requires a couple of extra parentheses, but is easy to understand and saves us from having to write a ton of filters. It also encourages consistency in your formatting.

It is common and useful to define these once per application in a helper module and import them. For example with our own Report Markup Language (used for PDF generation), we will commonly have a large template called 'rmltemplate.prep', and a helper Python module 'rmlutils.py'. Developers know that this module contains utilities for use in the template.

8.4 Block inheritance

We don't support this. It doesn't really fit with the nice metaphor of a template working just like a Python function. If anyone can suggest a way of doing it, we'll consider it.

Block inheritance is mostly used to let a designer set out the outer structure of web pages, with high level <*div*> tags which get filled in later. This can be done with a master template and included sub-templates.

CHAPTER 9

Support

We suggest using the `reportlab-users` mailing list for any support issues, or raising a bug report on bitbucket. ReportLab's commercial customers can also email our support address; others will be blocked by a whitelist.

CHAPTER 10

Credits

The initial implementation was done by Aaron Watters, to a broad design by Andy Robinson. This worked by actual code generation, creating a python module with the `getOutput()` function, which was optionally save to disk.

Robin Becker then optimised this by generating bytecodes directly, and later implemented the newer *function declaration* style.

Index

C

compileDir() (built-in function), 15
compileModule() (built-in function), 15
compileModules() (built-in function), 15

G

get() (built-in function), 13
getModule() (built-in function), 11
getOutput() (built-in function), 13

I

installImporter() (built-in function), 11

R

run() (built-in function), 13

U

uninstallImporter() (built-in function), 12