Preppy Spec

Lombard Business Park
8 Lombard Road
Wimbledon
London, ENGLAND SW19 3TZ

103 Bayard Street
New Brunswick
New Jersey, 08901
USA

Preppy Spec

Contents

Preppy Spec

```
Title: preppy spec
Version: Revision: 1.3
Author: Aaron Watters (aaron@reportlab.com)
Status: Final
Type: Design Specification
Created: 5-Oct-2000
```

Goals:

Preppy is intended as a simple and fairly general method for preprocessing text files containing special markup (which may contain loops, conditional flow control and generic python code) into python programs.

Installation and Dependency:

preppy requires the following Python libraries to be available for import:

```
string, sys, os, traceback, md5
```

Introduction

This document describes the operations of the module preppy.py. It is intended for an audience of proficient python programmers and is not intended as a user guide for novices.

Preppy is intended as a simple and fairly general method for preprocessing text files containing special markup into python programs. The markup may contain loops and conditional flow control and generic python code. A module generated by preppy sends text output to standard output. Nondirectives in the source file are sent unmodified and directives in the source file result in computation which may result in substitutions into the text output or evaluation of flow control that decides whether and how many times segments of text are injected into the output stream.

An example use for Preppy would be to generate a series of similar HTML files from the results of database queries -- where the general form of the document remains fixed but certain data elements change.

For example consider the Preppy source file

```
<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>
```

then with

```
dictionary = {"name": "Fred Flintstone", "sex": "m"}
```

the preppy output for this preprocessor source wedded with the dictionary input is

```
<html><head><title>Fred Flintstone</title></head><body>
hello my name is Fred Flintstone and I am
 a guy
        </body></html>
```

Alternatively with

```
dictionary = {"name": "Wilma Flintstone", "sex": "f"}
```

We obtain output

```
<html><head><title>Wilma Flintstone</title></head><body>
       hello my name is Wilma Flintstone and I am
        a gal
              </body></html>
```

Note that although these examples are using HTML, Preppy does no checking of any sort on whether it is valid. The *only* things Preppy acts on are the Preppy directives contained within double curly braces.

Intended Use

Preppy is intended for use in a professional environment with testing. In particular there are few provisions to prevent infinite loops or NameErrors, or Divide by Zero errors, or other problems familiar to professional programmers. In particular the preppy source

```
{{script}}from foo import bar; bar(){{endscript}}
```

may result in an infinite loop or a NameError and we make no provision to attempt to avoid this problem. Similarly for

```
{{while 1}}
 INFINITE LOOP!!! {{X}}
{{endwhile}}
```

This is only one example of many where using the preppy module may result in errors or bugs.

Operations

Load a preppy module (possibly from source text) using

```
m = getModule(name, directory=".", source_extension=".prep",
 verbose=0, savefile=1, sourcetext=None)
```

This function will look for the python module named "name" and if if exists check to see that it matches the preppy source module name+source_extension in the specified directory. If the module does not exist or does not match a new python module will be built from the source file, stored as a python module name+".py" in the directory (which must be writable if the modules don't match.)

If you do not want to save the regenerated module text then use savefile=0 and no file will be written during the load (but a matching existing python module will still be preferred).

If the source module is not present, but the python module is present the python module will be assumed correct and used. When the source text and the generated python module match the python code will not be regenerated (as an optimization).

The preppy module itself will keep a cache of those modules previously built by preppy and will not reexecute the test/generation code when a module of the same name is requested twice -- instead it will provide the previously built module.

It is also possible to use preppy to make a module without any use of the filesystem. In this case provide source text as a string with a module name (of no significance) to getModule

```
 m = getModule("dummymodule", sourcetext=my_string)
```

In this case (when sourcetext is provided) no file will be read and no file will be generated. The module m will be constructed in memory for use by the current process only.

The entry point for a Preppy module is the function

```
m.run(dictionary, __write__=None, outputfile=None)
```

Where dictionary provides external information to the run function. An example usage might be

```
m = getModule("salutation", directory="/usr/lib/preppy")
m.run({"name": "Wilma Flintstone", "sex": "f"})
```

Run has a second argument which allows the default "write" operation to be redefined. For example to append the output text segments to a list instead of writing to a file use:

```
L = []
app = L.append
m.run(dictionary, __write__=app)
```

NOTE: This will work only if the any script directive code segments either do not directly produce text output themselves (recommended) or use the __write__ function instead of printing directly to standard output.

Run has a third optional argument which allows the program to specify a file to use as the standard output for the function. In this case print statements will work (since sys.stdout will be set to the file while run() executes and set back to whatever it was previously afterwards. Thus you can call

```
f = open("flintstone.html", "w")
m.run(dictionary, outputfile=f)
```

WARNING: the preppy module will signal an error if both __write__ and outputfile are defined -- use one or the other or neither, not both.

Recursive Imports of preppy modules:

Recursive imports of preppy modules should use getModule which guarantees that once the module is loaded once it will not be loaded a second time. At present there is no special directive to do this. Use a script directive.

NonDirectives

Text not recognized as directives (or partial erroneous directives) is sent to standard output unmodified by the run() function.

Directives

Directives are set off by

```
STARTDELIMITER = "{{"
ENDDELIMITER = "}}"
```

these module constants can be altered, but alteration may result in parsing difficulties if the replacements are not chosen carefully.

NOTE ABOUT ESCAPING: The string {${ will be translated to {{ anywhere in the text after delimiters have been identified. Similarly }$} will be translated to }}. Furthermore $$ will translate to $. To specify two $'s in sequence use $$$$. Single $'s will be left alone outside of {${ or }$}.

Below are the discussions of the directives in turn.

token: {{token}}

This construct is recognized if the token does not match any of the other directives. The result of this directive is the value of token evaluated as a python expression in the context of the m.run() function. An example might be

```
{{dictionary["name"]+2}}
```

The result of the evaluation is introduced into the output stream (using standard python string conversion, if needed).

eval: {{eval}}python_expression{{endeval}}

This is a longer way to spell {{python_expression}} :). It is useful for larger expressions like

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

The expression is evaluated and the result is inserted in the output stream as in token.

WARNING: for token, 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. 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 similar.

NOTE: inside the script and eval directives the STARTTAG {{ and ENDTAG }} will be ignored unless they occur within {{endeval}} or {{endscript}} respectively.

script: {{script}}python_code_line{{endscript}}

which can also be

```
{{script}}
many
lines
indented
properly
{{endscript}}
```

This executes a sequence of python code within the context of the run() function. An example would be

```
{{script}}import math; x = math.sin(math.pi/4.0){{endscript}}
```

or

```
{{script}}
import math
x = math.sin(math.pi/4.0){{endscript}}
```

RESTRICTION: in the multiline case the indentation should follow Python conventions (of course) and any initial indentation should be character for character the same between the lines of code (except for completely white lines). In particular if the first line begins with 8 spaces the next line should not begin with a TAB.

SCRIPT RECOMMENDATION 1: If possible the script tags should not directly produce standard output, but instead define string variables that are introduced in token directives.

SCRIPT RECOMMENDATION 2: If a script must produce standard output itself it should use the function

```
__write__(string)
```

which is always defined in the context of the run() function and allows the output to be redirected flexibly without modifying sys.stdout. In particular the script should not use the "print" statement or other direct writes to sys.stdout.

SCRIPT RECOMMENDATION 3: If you ignore the first 2 recommendations then never use the second argument of the run() function to redirect the output -- reset sys.stdout instead using the outputfile=file third argument. In this case the generated function cannot send output to any structure which doesn't support the Python file protocol.

if forms

```
{{if condition}} block {{endif}}
{{if condition}} block1 {{else}} block2 {{endif}}
{{if condition}} block1 {{elif condition2}} block2 {{endif}}
{{if condition}} block1 {{elif condition2}} block2
  {{else}} block3 {{endif}}
 ...etcetera...
```

This conditionally executes segments of text and directives. each condition should be a python expression in the context of the run() function. The blocks may contain other directives and text.

for loops

```
{{for for_target}} block {{endfor}}
```

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

```
for for_target:
 interpretation_of(block)
```

while loops

```
{{while condition}} block {{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)
```

STATIC COMPILATION

To statically compile a preppy module once and for all you can use preppy as command line program. For example suppose the example file given above is in "./flintstone.prep" in the same directory as preppy. Then compile it using

```
C:\reportlab\repository\rlextra\preppy> preppy.py flintstone
no module flintstone found (or error)
CHECKSUMS DON'T MATCH
regenerating python source from .\flintstone.prep
```

You can then test it using a static import

```
C:\reportlab\repository\rlextra\preppy>python
Python 1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)] on win32
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> from flintstone import run
>>> D =  {'sex': 'm', 'name': 'george'}
>>> run(D)

        <html><head><title>george</title></head><body>
        hello my name is george and I am
         a guy

        </body></html>
>>>
```

Note that preppy modules which do not dynamically import other preppy modules may be used "stand-alone" without the preppy module itself present.

SYNTAX CHECKING:

Python code expressions and code blocks are checked for syntactic correctness using the Python compiler during generation in an attempt to detect errors in a helpful way during code generation. Partial code blocks or expressions are not allowed.

For example the following will result in errors reported at generation time.

```
{{5*}} {{10}}  (5* is not a complete expression)
```

and

```
{{script}}
if x=40: # syntax error in expression x=40
 y = 9
{{endscript}}
```

and

```
{{script}}
if x==40: # incomplete if statement
{{endscript}}
 the value of x is 40
```

and

```
{{if x==40: y=9}} (not a valid if directive)
```

However NameErrors or other run time errors which are not detected by the Python compiler will not be detected at generation time. For example the compiler will not understand that this is an invalid function call (it is syntactically acceptable to the grammar)

```
{{ "this"("doesn't work") }}
```

This error will be detected during execution of the run() function if and when the code segment containing it is executed.

```
...TypeError: call of non-function (type string)
```

ADDITIONAL NOTES:

You may define classes and functions in scripts but remember that they are defined in the local context of the run() function and do not have access to local variables in the run() function. Any variable you wish to use in a class or function (except for __write__ and dictionary and outputfile, the run function arguments) can be declared global in the script

```
global variable
```

(But note that this is a slightly dangerous thing to do in the unlikely event that the generated module is used in a multithreaded python application). Also if you wish to make the classes or functions available to other modules you must declare them global and the run() function must be executed at least once before the exported functions or classes are used (NOT RECOMMENDED!).

WARNING: extra whitespace at the end of a line is discarded in the source. (rationale: had problems with extra carriage returns at ends of some lines and this was an easy fix, but AFAIK it may need to be undone at some later time if trailing whitespace is needed sometime...).

Commenting your code: There is no specific Preppy form of comments. It allows you to use python comments inside its directives, or you can use the style of comments in whatever language preppy is processing (eg in HTML). Be careful about mixing the two. Preppy will always execute preppy directives, no matter what surrounds them. So attempting to comment out Preppy directives with HTML comments will not work - something that's easily done if you are not concentrating.

```
<!--
{{# if this wasn't a comment, Preppy would execute it}}
-->
```