# PYM - A Macro Preprocessor Based on Python

**Robert F. Tobler**
**VRVis Research Center for Virtual Reality and Visualization**
**Vienna, Austria**

## Abstract

In a number of tasks the need for a macro preprocessor arises. Most macro preprocessors, are either syntactically tied to the language they support (e.g., cpp, the C preprocessor, or the lisp macro facility), or they are limited in their functionality (cpp), or may even have an arcane syntax (e.g. m4, chakotay).

We propose a macro preprocessor - PYM - based on the Python scripting language, which retains the complete expressivity of Python for writing macros, and thus is not limited by an arcane syntax or a limitation in its functionality. The complete implementation of this macro preprocessor is a Python script of around 200 lines of code, and includes the three main functions of macro definition, macro expansion, and file inclusion. Using Python's exception mechanism, conditional termination of expansion both on a per file, and overall level has been implemented. PYM has been shown to be useful for macro based generation of VRML files, and for macro based generation of HTML files for a dynamic web server.

## Keywords

macro preprocessor, python tool

## 1. Introduction

Macro facilities have been introduced to various programming and defintion languages for a variety of reasons. Among these reasons are:

- limited expressive power of the base language (e.g. the WEB system [Knuth], various assembly languages).
- optimization by inline expansion (`cpp` was often used for this)
- definition/inclusion of common code sequences
- addition of a module structure (e.g. `cpp`)

As varied as the reasons for introducing a macro facility to a languages, are the actual implementations of these facilities. Some macro facilties are not implemented as preprocessors, but are tightly tied to the language, so that the macro expansion process is directly integrated into the language parser. An example for such a macro facility is the LISP macro facility [Hart].

Most macro facilities are implemented as preprocessors, and are thus potentially usable not only for the language for which they were designed. The most well-known example for such a preprocessor is the C preprocessor. The C Preprocessor is however also tied to its language by the fact, that it searches the whole input text for symbols to expand, and the arguments to macros are scanned so that they contain a balanced number of opening and closing parantheses. Therefore its use as a generalized preprocessor is limited to languages or texts which have a similar lexical structure to C.

On the very other end of macro preprocessors are facilties that have no syntactical relation to any language, and use special tags for starting definitions of macros and use of macros. An example of such a macro preprocessor is Chakotay [Probst]

## 2. General Operation of a Macro Preprocessor

There are two main operations of a macro preprocessor:

- macro definition
- macro expansion

The well-known C preprocessor, as an example, uses the hash character `#` as the first non-space character in a line to introduce preprocessor commands, and defining macros is one of these commands. For macro expansion, the C preprocessor doesn't use a special macro, it just scans the input text for lexical symbols as defined by the C language, and expands these.

Other preprocessors such as Chakotay [Probst] use special characters to introduce both macro definition and macro expansion. This type of preprocessor is more generally applicable, as there is no restriction on the lexical structure of the language or text for which it is used.

In addtion to the two basic operations described above, a third, optional operation that can be performed by a macro preprocessor is file inclusion. If we take the C preprocessor as an example again, it introduces the `#include` command to expand the content of a referenced file in the output text. This facilitates a modularized structure of coding which may not be part of the original language that the preprocessor is used for.

## 3. PYM operation

As we needed a preprocessor that can be phased in step by step, one of our requirements was that it should not change any text without explicit notification. Thus we decided to mark both macro definition and macro expansion with special characters or character sequences.

**Macro Definition**

We decided that global Python variables containing strings should be used in PYM as constant macros, and Python functions returning strings should be macros accepting arguments. Thus the macro definition phase of PYM is just the definition of Python globals and Python functions. As Python, with its indentation based syntax is somewhat line-based, we decided to use a line based special sequence to introduce PYM macros:

```
#begin python

#python code for defining "macros" goes here

#end python
```

With these two special sequences it is possible to switch between literal text that is directly output, and Python code that is executed.

This is however only one part of the functionality. The second one is

**Macro Expansion**

For macro expansion we leave the choice of start and end sequences to the user. In contrast to the sequences for introducing macro definition, the sequences for macro expansion can appear anywhere in a line, and the start and end sequence can be placed on different lines.

Although the starting and ending sequence are user definable, we will use the defaults for using PYM as a html preprocessor in all examples that will follow. These two sequences are `<[` and `]>` (i.e. "special" html tags). With these two sequences defined, and the macro definition explained earlier we can write our first little PYM example:

```
#begin python
TITLE = "My Web page"
def EXP(e): return str(e)
#end python
<head>
<title><[TITLE]></title>
</head>
<body>
<h4><[TITLE]></h4>
<p>The value of 2 raised to the 4th power is
<[EXP(2**4)]>.</p>
</body>
```

The output of running this through PYM should be obvious:

```
<head>
<title>My Web Page</title>
</head>
<body>
```

```
<h4>My Web Page</h4>
<p>The value of 2 raised to the 4th power is 16.</p>
</body>
```

For another example of using PYM, see appendix A. Note that we use the generation of HTML code as a basis for all these examples. This is only due to the fact that most people understand HTML. The original motivation for writing PYM were the limitiations in the VRML 2.0 file format for describing 3 dimensional objects. PYM has been successfully used to generate quite complex VRML code by building on a modularized set of VRML macros.

**File Inclusion**

Now that the main functionality of a macro preprocessor has been covered, we can think of the one operation that most preprocessors support in addition: file inclusion. We decided to use a syntax similar to the C preprocessor for including files:

```
#include "filename"
```

With this addition common code and text can be shared by multiple files. Although we could have relied on Python's import mechanism, since we are executing Python code in the macro definition sequences, we decided to make file inclusion an explicitly supported feature of PYM, so that blocks of text can be directly included without resorting to the definition of macros.

**Conditional Text output**

A number of preprocessors support conditional text output. As an example the C preprocessor has the `#if` *`expr`*, `#elif` *`expr`*, `#else`, and `#endif` directives. This is supported by PYM with the expressions being standard Python expressions.

In addition to this standard mechanism, the following two tricks are supported, which may be more convenient in some circumstances

The first one of these tricks is *computed include*: we have made the filename behind the include statement an actual Python expression. So not only a direct string but also a function can be used that returns the filename of the file to be included. Thereby it is possible to include text based on arbitrary conditions.

The second trick is *signalled termination of expansion*: whenever Python code is executed in PYM, either while defining macros or while expanding macros, one of two exceptions may be explicitly raised by the code in order to affect PYM output: `PymEndOfFile` or `PymExit`.

Raising `PymEndOfFile` immediately terminates text output of the current file, continuing expansion at the previous level of file inclusion. Raising `PymExit` completely terminates text output.

With these two facilities conditional text output is fairly easily controlled, without too much additional functionality that has to be handled by PYM.

**Implementation Notes**

Implementing PYM consists of four major parsing tasks:

- scanning for Python macro definitions
- including files
- conditional output
- expanding Python macros

The first three tasks are handled by a line based parser, that searches for blocks of Python code and executes them, for include lines and recursively expands files, and for if-elif-else-endif sequences and conditionally outputs the relevant text.

The fourth task is handled by searching for pairs of the begin and end sequences for macro expansion, and evaluating the expression between these two sequences as a Python expression. Macro expansion is recursively applied to the result of each expression until no more special sequences are found.

The Python code for macro definition, the Python expressions for macro expansion, conditions, and the computed include file name are all executed in the same environment that is separate from the global name spaces of PYM itself. The two exception classes used for termination of expansion are defined in the global name space of PYM and mirrored in the environment used for macro definition and expansion, so that they can be used to communicate between these two execution environments.

## 4. Advantages of Python for implementing a macro preprocessor

Using Python as a base language for PYM had a number of advantages, among these are:

- *Multi-line strings*: Often the macros that need to be defined are blocks of text that need to be directly streamed to the output. Python's multi-line strings make it possible to simply take the desired text block, surround it with triple quotes, give it a name by assigning it to a variable and use it as a macro.
- *the `%` operator for strings*: this makes it possible to take text blocks and easily parametrize them and use them as parametrized macros.
- *named and default function arguments*: with Python's facility for using named function arguments, and define default argument values, it is possible to program macros that already have default values for their parameters, and parameters that need to be overridden can be specified by name.

All of these Python features make PYM a very comfortable macro preprocessor (see appendix B for a short manual). These are of course also some of the features that make Python a very comfortable programming language. A similar extension could be applied to any other scripting languages, such as Perl for example, but the clear and easy syntax of Python serves as an additional advantage which makes programming macros, quite a bit more readable than any other macro lanaguage known to the author (e.g. compared to M4 [Kernighan], autogen [Korb], and chakotay [Probst]). In addtion to that, PYM profits from all extensions to python, there is no need to reimplement a feature specifically for the preprocessor, as PYM has access to all python features. This of course is an advantage that is not shared by any dedicated preprocessor.

## 5. Implementation Issues

Executing Python code that is embedded into text that is directly streamed to the output results in an offset between Python's notion of what the line number is and the actual line number in the file. When errors in the Python code are encountered, wrong line numbers will therefore be reported. In addtion to that Python does not know about the file name where ther error appeared.

In order to fix this, the exceptions for various errors are caught by PYM and the linenumber anf filename fields in these exeptions are corrected before they are reraised. This would be a straightforward job, if every exception in Python had a linenumber and a filename field. However this is not the case: some Python exceptions do not have linenumber and filename fields.

For this reason PYM is not able to report the linenumber and filename in which an error occured reliably. I consider this to be a small shortcoming of Python, as every Python exception should just contain a filename and linenumber field, so that this kind of operation can be easily implemented.

## 6. Conclusions and future work

A macro preprocessor based on the Python language - PYM - has been introduced, that makes it possible to use Python for defining macros for arbitrary other languages or text files. With the full expressivity of Python at hand this preprocessor simplifies the definition of macros, and increases the readability of macro code. Another nice feature of PYM is its compact implementation (see appendix C) of less than 200 lines of code.

Due to the compactness of PYM, issues such as security and conditional expressions are left (or not left, since the ?: operator is missing) to Python.

The performance of PYM has not been investigated, and since the two parsing tasks of finding Python code blocks, and Python expressions are handled by functions written in Python it is not expected to be overwhelmingly high. The main target of PYM was readability, and thus the performance was never under consideration. If there is enough

demand for a faster PYM parser arises, these two functions are the obvious targets for optimization, and possible reimplementation in C.

The applicability of PYM to various problem has been verified by generating rather complex VRML code with it, and by implementing a photography web site: http://ray.cg.tuwien.ac.at/rft/Photography/.

## 7. References


- [Hart], Timothy P. (1963), "MACRO Definitions for LISP", AI Memo, Massachusetts Institute of Technology, USA.
- [Kernighan], B.W., and Ritchie, D.M. (1979), "The M4 Macro Processor", Unix Programmer's Manual, Comp. Sci. Tech. Rep. No. 2, Bell Labs, Murray Hill, N.J.
- [Korb], Bruce (1992), "Autogen - The Automated Program Generator" http://autogen.sourceforge.net/ (accessed 01/15/01)
- [Knuth], Donald E. (1982), "The WEB System of Structured Documentation", Stanford University, CA, USA.
- [Probst], Mark, and Deinhart, Heinz (1998), "Chakotay - a preprocessor that can be applied to a myriad of applications" http://www.complang.tuwien.ac.at/~schani/chpp/ (accessed 11/05/00)


## Appendix A: A small Example for using PYM

Suppose you want to generate a HTML page with numbered chapters and figures. The whole functionality can be placed into a file `number.pym` that can be included by any HTML file:

```
#begin python
NUMBER_MAP = {}

def NUM(tag):
    num = NUMBER_MAP.get(tag, 0) + 1
    NUMBER_MAP[tag] = num
    return str(num)
#end python
```

This could be used in a file `page.html` as follows:

```
#include "number.pym"

<h4><[NUM("h4")]> Introduction</h4>
<p>Figure <[NUM("fig")]></p>
<h4><[NUM("h4")]> Why do we number Chapters?</h4>
<h4><[NUM("h4")]> Why do we number figures?</h4>
<p>Figure <[NUM("fig")]></p>
```

Subjecting this example to PYM by typing `pym page.pym` results in the following output:

```
<h4>1 Introduction</h4>
<p>Figure 1</p>
<h4>2 Why do we number Chapters?</h4>
<h4>3 Why do we number figures?</h4>
<p>Figure 2</p>
```

## Appendix B: A short PYM manual

Pym is a command line tool invoked by:

```
pym [options]filename
```

This subjects the file *filename* to macro expansion and putting the result on standard output. Although PYM does not prescribe a specific extension, we conventionally use `.pym` in order to indicate that a file needs to be run through PYM.

Currently PYM only understands one option: `-I` *directory*, which appends the given directory to the Python include path `PYM_PATH`. If the directory is given by a relative path, it is taken to be relative to the call directory of PYM. The includ option can be used multiple times to append multiple directories to the include path

As a file is run through PYM, each sequence of Python code surrounded by the following two lines is executed:

```
#begin python
python-code
#end python
```

For each line in *filename* of the following form:

```
#include file-name-expression
```

*file-name-expression* is evaluated as a Python expression. The resulting string is taken as a filename to be streamed to the output instead of the include line. The included file is recursively subjected to all rules for macro definition and macro expansion. Note that a filename surrounded by double quotes is a valid Python expression, and can thus be used to include a file:

```
#include "file-name"
```

In order to find the file specified in the include statement, the file is first searched locally relative to the including file. If this fails, the directories in the Python include path

`PYM_PATH` are checked, one by one, until a match is found. Finding no matching file, is currently handled silently

In order to facilitate conditional text output and conditional macro definition, the following syntax is supported (note that you can have any number of `#elif` blocks, including none):

```
#if python-expression
text block
#elif python-expression
text block
#else
text block
#endif
```

Each of the python-expressions is evaluated, and if the result is true, the following block is routed to output. If the result is false, the following text block is suppressed, and Python code inside the text block is not executed. The block following the `#else` block is routed to output if the previous block was suppressed.

All other text in *`filename`* is subject to macro expansion. Each sequence of the form:

```
<[python-expression]>
```

Is replaced by the result of evaluating *`python-expression`* and recursively replacing such sequences by their expanded output. Note, that this recursive expansion is not applied to sequences that are shorter than the sum of the length of the two character sequences that start and end the expression (`<[`, and `]>`, in total 4 characters), thus it is also possible to generate these two sequences, if each one of them is the result of a complete python-expression. The recursive process is only performed on each of the results of a python-expression individually, the concatenated string of the results is left unchanged.

Within any Python code executed, the two special strings starting and ending a python-expression can be set to new strings with the following sequence:

```
PYM_EXPRESSION = ("start-sequence", "end-sequence")
```

If the output should be placed in a file instead of just be printed on standard output, the extension of a result file can be specified in any Python code executed:

```
PYM_EXTENSION = "output-extension"
```

Within any Python code executed, the output for the current file can be terminated by the following statement:

```
raise PymEndOfFile
```

Within any Python code executed, the output can be completely terminated by the following statement:

```
raise PymExit
```

The search path for include files can be extended in any Python code executed by:

```
PYM_PATH.append("directory")
```

## Appendix C: Using PYM with an http-server

A nice application of PYM is the dynamic generation of html pages, based on PYM sources. In order to faciltate that, PYM checks for the existence of the environment variable `DOCMENT_ROOT` which is part of the standard http server interface. The environment variable `PATH_TRANSLATED` is used to locate the file that has been requested from the http server. The `DOCUMENT_ROOT` is also appended to the `PYM_PATH` so that PYM file inclusion starts searching files in a well defined place. By telling the server to filter each file with extension `.pym` through the PYM command, it is possible to implement a whole web-site based on pym-macros.

As an example, in order to configure the apache web server (http://www.apache.org/), so that each file with extension `.pym` is filtered through the PYM command, add the following directives, in the respecitve places in `httpd.conf`:

```
DirectoryIndex index.pym index.html
...
AddType application/x-httpd-pym .pym
...
Action application/x-httpd-pym /cgi-bin/pym.py
```

## Appendix D: The source code of PYM

The source code to PYM 1.0 as of January 15th, 2001, is actually less than 200 lines long. As a reference you can find all of PYM at http://ray.cg.tuwien.ac.at/rft/Papers/PYM/.