

pppp — Poor's Python Pre-Processor

(As within Pymacs 0.24-beta2)

Author: François Pinard

Email: pinard@iro.umontreal.ca

Copyright: © Progiciels Bourbeau-Pinard inc., Montréal 2010

Contents

1 Introduction	1
1.1 Why pppp?	1
1.2 Installation	1
2 Pre-processor syntax	2
2.1 Substitutions	2
2.2 Conditionals	2
3 Invoking pppp	3
3.1 Setting the context	3
3.2 Pre-processing files	3
3.3 Cleaning out files	4
3.4 Merging versions	5
4 Limitations	5

1 Introduction

1.1 Why pppp?

The Python community has long resisted the idea of a pre-processor for Python, and quite understandably. The usual features of a pre-processor for other languages are well served at run-time in Python, alleviating the need.

The advent of Python 3 changes the picture somehow, as Python 3 does not accept some Python 2 constructs, and vice-versa. In many situations, one cannot (at least without stretched stunts) write a single source file which can be compiled by both Python 2 and Python 3. The languages are so similar that it is irritating to keep sources separate: this is too much a burden for maintenance, in my opinion.

This **pppp** tool was written to help porting Pymacs to Python 3. I guess it could be useful for other Python programs or packages.

Report problems, documentation flaws, or suggestions to François Pinard:

- <mailto:pinard@iro.umontreal.ca>

1.2 Installation

There is no installation machinery for **pppp**, and Pymacs does not install it either. To install it, merely pick the `pppp` script from the top-level of the Pymacs distribution, either going through the main Pymacs site:

- <http://pymacs.progiciels-bpi.ca>

or more directly from GitHub:

- <http://github.com/pinard/Pymacs>

Copy that file somewhere on your search path, and make it executable. That's all to it.

2 Pre-processor syntax

There are two mechanisms in **pppp**. One does in-line substitutions, the other takes care of conditional compilation. In-line substitution occurs first, one line at a time, then conditional compilation occurs on the result of the substitutions.

The two mechanisms both rely on a preset *context*, which is a set of definitions. Each definition relates a name to a Python value. The context is built under the control of options given to the **pppp** program. The same context is used both for substitution and conditionals.

The behaviour of **pppp** is currently unspecified when substitutions of a single line produces multiple lines, for which the first or any other is meant for conditional compilation. So don't do that!

2.1 Substitutions

Substitution is triggered on each occurrence of `@NAME@` in the sources. In each case, if *NAME* is not the name of a context element, substitution just does not happen and the occurrence is left undisturbed — silently, without diagnostic. If substitution happens, *NAME* and the surrounding `@` delimiters get replaced by the string of the value associated with that name within the context.

Unless `@NAME@` sits within a Python string or a Python comment, it is invalid Python syntax. So (contrarily to conditionals described below), if the substitution notation is used, pre-processing is likely mandatory.

2.2 Conditionals

Conditional compilation is merely driven by usual Python `if` statements. However, to be considered for conditional compilation, the `if`, `elif` or `else` lines should have the following colon (`:`) on the same physical line. Moreover, such lines should not use Python comments.

The test expression associated with the `if` or any `elif` is evaluated using the pre-processor context. If all the variables or functions referred to by the expression are known in the context (and presuming there is no syntax error or other run-time error while evaluating the expression), the expression gets a dependable value. The `if` or `elif` line is itself removed (well, in some cases, an `elif` might become an `else`),

and the following block of lines is adjusted according to the expression value, likely shifted back or fully removed. Similarly, `else` clauses may sometimes get simplified.

While it is possible to use very invalid Python syntax which, through **pppp** conditional compilation, is turned into a valid Python program; users are much invited to use conditional compilation in such a way that sources meant for **pppp** are directly legal Python syntax.

This idea of writing conditionals as correct Python could be pushed even further. If the user manages to compute and assign the context variables at run-time in the Python program, conditional compilation for some name could be replaced by run-time checks on that name merely by *not* defining the name in the **pppp** context. By doing so, the test expressions involving that name may not be resolved by the pre-processor, and the simplifications just does not occur.

3 Invoking pppp

The **pppp** command is called using the usual syntax for Unix / Linux commands:

```
pppp [OPTION]... [ARGUMENT]...
```

The operating mode of the program, and the meaning of arguments, depend on some options being used or not. Option `-c` forces clean out mode, option `-m` forces merge mode. Otherwise, the program uses the pre-processing mode.

The `-h` option is special. When given, a short help summary is written on standard output, and then, the program exits immediately.

The `-v` option raises the verbosity level of the program, which then produces output about created directories, written files or deleted files.

3.1 Setting the context

The context used for the pre-processing is initially empty. It does not even have Python builtins. It is then filled through the use of `-C` or `-D` options, which may be repeated when there are many definitions to introduce, or when there is a need to override previous settings.

Option `-D name` adds *name* into the context, associating it with the Python value `True`. Option `-D name=expr` adds *name* into the context, associating with the value of the Python expression *expr*. Beware of Python characters which also have a meaning for the shell, proper quoting may be needed. Here is, for example, how to define a string while calling **pppp**:

```
pppp -D "version='0.24-beta2'" ...
```

While evaluating *expr*, there is no restriction to the context, and builtins are indeed available. For example, to add the builtin `ord` into the context, merely use `-D ord=ord`.

Option `-C FILE` reads and evaluates *FILE* as a Python source. All variables computed at the outer level then become names in the context, and the values of these variables become the values associated with the names within the context. Any function defined at the outer level of *FILE* also gets available to **pppp** pre-processing.

Beware of uncleaned variables in *FILE*. For example, an `import sys` creates a `sys` variable, which you normally clean with `del sys` near the end of *FILE*. If you do not do so, that variable is available to the pre-processor. So if you have a line like:

```
if sys.version_info[:2] == (2, 7):
```

somewhere in your **pppp** source, this might be evaluated as `True` or `False` at pre-processing time rather than at run-time, and this might not be what you wanted.

3.2 Pre-processing files

Without options `-c` nor `-m`, the arguments to the program indicate which files are going to be pre-processed. If there is no argument at all, this is a special case by which standard input is read, pre-processed and then written to standard output.

Otherwise, only eligible files are retained for pre-processing. To be eligible, the name of a file should end with `.in`. If an argument names a directory, that directory is recursively searched to find all files with such an `.in` suffix. When a directory has a `.in` suffix (either given as an argument, or a subdirectory of a directory argument), *all* the files it contains become eligible, including all files of its subdirectories, recursively.

Now, that `.in` suffix may be changed to something else, using the `-s NAME` suffix option. The period is part of the option value. For example, `-s '.in'` is equivalent to not specifying it.

Each eligible file is pre-processed and written on another file, the name of which is related to the name of the file being read. That name is produced by removing the `.in` suffix, and more precisely, by removing all `.in` suffixes, would they appear in directory names or file names. Moreover, the optional `-o OUTPUT_DIRECTORY` option introduces a directory into which all resulting files are collected: it effectively prepends `OUTPUT_DIRECTORY/` to all output names. If the suffix gets declared empty through `-s ''`, then *all* files are eligible, and because output names would be identical to the input names, the `-o` option becomes mandatory.

You do not have to prepare intermediate directories to receive output files. These are created on the fly, as needed.

Pre-processing uses substitutions and conditionals. Substitutions automatically occur on all eligible files. Conditionals, however, only apply for files which are known to be Python sources. If option `-p` is given, all files are considered to be Python sources. Otherwise, a Python source has a file name which ends with `.py` or `.py.in`, or appears to use a Python shebang line (the precise heuristic checks that the first line of the file starts with `!#` and has `ython` written somewhere in it).

The **pppp** tool assumes, by default, that the Python sources consistently use an indentation step, and that the indentation step is 4 columns. This can be changed with the `-i INDENT` option. For example, `-i 8` means that the indentation step is 8 columns.

By default, **pppp** generates white lines in the pre-processed results to replace any removed lines. The idea is to guarantee usable line numbers in any later traceback, that is, numbers that refer to the correct position within the original file, before it was pre-processed. The file name would still differ by the `.in` suffix, of course, which is a lesser worse. Whenever, as side-effect of substitutions, a single input line yields many output lines, line synchronisation may be lost. **pppp** then inhibits the production of replacement white lines until the line synchronisation is recovered. Option `-n` wholly inhibits the production of any white line only meant for synchronisation.

Because tracebacks mention the file name after pre-processing, and not the original source before pre-processing, users are likely to inspect the resulting file, and after a while, start modifying it without realizing their mistake: a resulting file might be overwritten by a later invocation of **pppp**, so losing user's modifications. To play safe, **pppp** attempts to detect this: it copies the modification time from the original

into any resulting file it produces. Then, whenever a resulting file is newer than the original source, **pppp** raises an error instead of deleting or rewriting it. Finally, as a way to force Python recompilation in case the resulting file becomes different, it removes an already compiled Python file, if any. If you want to force deletions or rewritings regardless, use option `-f`.

3.3 Cleaning out files

As a convenience for `Makefile` writers, there is an option to help at cleaning out derived files. With `-c` specified, any file that would have been produced in pre-processing mode is removed instead.

Of course, to be useful, the command arguments naming files or directories should be the same as those used for pre-processing.

3.4 Merging versions

As a way to help prepare a Python file for **pppp** pre-processing, the program offers a mode able to produce a pre-processable file out of two versions of a given Python source. For example:

```
pppp -mD VERSION2 script1.py script2.py > script.py.in
```

compares `script1.py` with `script2.py` and produces a merged version on `script.py.in`. Then, the command:

```
pppp -D VERSION2=False script.py.in
```

would produce a file `script.py` which is equivalent to `script1.py`, while the command:

```
pppp -D VERSION2 script.py.in
```

would produce a file `script.py` which is equivalent to `script2.py`.

Whenever option `-m` is used, exactly one `-D` option provides the segregating name used in added conditionals, and two arguments tell the versions to be compared.

Beware that this mode was quickly written, and stays rather crude and approximative. This is merely a way to get started. The real and patient work comes afterwards, with a text editor, to clean and fixup things, and bring the merged result closer to real Python syntax.

While editing the result, you might find some `#endif (pppp)` lines generated here and there. These are protective measures, so the later pre-processing does not clearly produce wrong results. These lines usually indicate problematic areas, for which revision and careful refactoring is especially needed.

4 Limitations

- The need of a very consistent indentation, as far as the indentation step is considered, may be too stringent a condition. It would surely be nicer if **pppp** was able to adapt to the indentation in use.
- This tool is easily fooled by unindented comments or multi-line strings, as it is driven only by textual line indentation. It does not follow whether a line is part of multi-line string or not.