Internal Report

INR 31/13 – NUKLEAR 3462

November 2013

TSP: Python package to facilitate preparation of input files

Anton Travleev

Abstract

Preparation of input files for computer codes is an essential and routine task in nuclear reactor design. A concept to simplify this task is proposed and software, developed to support this concept, is described in this report.

Contents

1. Introduction

Application of computational codes often implies preparation of text input files. In a simple case, an input file can be written by hands, but often the amount of details to be passed to a code makes an input file long and complex so that the write-by-hands approach becomes almost impossible. Some codes provide means to simplify preparation of input files, for example, by allowing inclusion of external files, or by providing syntax to define variables that can be referenced later instead of explicit input data, but usually such possibilities are rather limited.

A widely used approach to create complex input files is to write a program that generates it. This approach has some drawbacks. Input files generated programmatically are usually very case-specific and even a small extension of the computational model might imply considerable changes in the program. Another drawback is that some parts of an input file can be rather simple and it is much more convenient to write it by hands than to write a programm that generate these simple parts.

Another approach that partly resolves the latter problem is to use templates. The generating program computes necessary data and inserts them into a template file that is prepared in advance. Thus, some parts of the input file can be written by hands to the template, and generation of the rest can be coded. In this approach one needs to define an algorithm to find places for insertion. The simplets way – to insert something into a predefined line and position, makes it difficult to change the template later on. More sophisticated algorithms, such as using particular strings in the template file to be replaced by the program are difficult to apply especially if the amount of inserted data is large. Another drawback of this approach is that information needed to write the input file is located at at least two different places: in the template file and in the program.

We propose a concept that is an extension of the concept of templates and resembles the idea of literate programming [1]. Accoring to this concept, the template file contains both parts written by hands that is transferred to the input file unchanged, and parts containing code chunks that will be substituted by the result of their evaluation during preprocessing. Thus, the template file contains all the information necessary to produce an input file, and places where evaluation result must be inserted is defined naturally by the place where the code chunks are found in the template.

Software that supports this concept is implemented in the Python programming language [2]. It is distributed as a Python package called TSP (stands for “Text with Snippets Preprocessor”) and can be installed onto any machine where Python is installed (tested on Windows 7 and Linux OS). The TSP package provides an executable that takes a text file as input, searches for pieces of Python code in it, executes these code chunks and writes an output text file that is a copy of the input text file, but with the Python code pieces substituted with the result of their evaluation.

This report explains the installation procedure of the TSP package, describes the syntax of the template file and gives some examples of the usage of TSP.

In the rest of the report, the following terminology is used. A file containing text and Python code pieces is called **template file**, or just **template**. Pieces of Python code inside a template are called **snippets**. The output file generated by processing of an template, is called **resulting file**. The script provided by the TSP package that reads a template file and generates correspondent resulting file, is called **preprocessor**.

To illustrate what the TSP package can be used for, consider the example that shows the template file on the left and the resulting file on the right:

Template

```
''
This is arbitrary text, and␣
↪this is
a snippet: 'pi=3.14159',␣
↪delimited
with characters from the first␣
↪line.
In this snippet, the variable␣
↪pi is
defined. After a variable is␣
↪defined,
it can be referenced: 'pi*2'.

A snippet can take several␣
↪lines. It
is like a python script, but␣
↪saved
directly in the template file.␣
↪It can
access variables defined in␣
↪previous
snippets.
'
from math import cos
print
for x in [0, 0.3, 0.7, 1.]:
    print "%6f %9.2e" % (x,␣
↪cos(pi*x))
'

The standard output generated␣
↪by a
snippet, if any, will be␣
↪inserted
into the resulting file right␣
↪after
the snippet.
```

Resulting file

```
This is arbitrary text, and␣
↪this is
a snippet: 'pi=3.14159',␣
↪delimited
with characters from the first␣
↪line.
In this snippet, the variable␣
↪pi is
defined. After a variable is␣
↪defined,
it can be referenced: 6.28318.

A snippet can take several␣
↪lines. It
is like a python script, but␣
↪saved
directly in the template file.␣
↪It can
access variables defined in␣
↪previous
snippets.
'
from math import cos
print
for x in [0, 0.3, 0.7, 1.]:
    print "%6f %9.2e" % (x,␣
↪cos(pi*x))
'
0.000000  1.00e+00
0.300000  5.88e-01
0.700000 -5.88e-01
1.000000 -1.00e+00


The standard output generated␣
↪by a
snippet, if any, will be␣
↪inserted
into the resulting file right␣
↪after
the snippet.
```

The template's first line contains two quotes, `'`, they will be used in the rest of the template to define the beginning and end of snippets. The first line is never copied to the resulting file.

The first snippet, `'pi=3.14159'` is a valid Python statement defining variable `pi`. This snippet produces no output, its content is just copied into the resulting file, but the variable `pi` can be used later on.

The next snippet, `'pi*2'`, is a valid Python expression. It can be evaluated, and in the resulting file its code is substituted with the string representation of the result of evaluation.

The next snippet, starting with statement `from math import cos`, takes several lines and represents a valid Python script that can be executed. This snippet contains print statements that generates some standard output that, together with the snippet's code, will appear in the resulting file.

Note that the structure of the template is accurately repeated in the resulting file. Results of snippets evaluation appears exactly at places where the snippets were written in the template. Variables defined in previous snippets can be used in the following snippets. String representation of snippet evaluation can be controlled with optional snippet keys. Multi-line snippets can be commented out in the resulting file. A snippet can be as simple as an in-line reference to a variable, or it can represent a complete Python script, thus, the whole power of Python ans its packages can be used to facilitate preparation of input files.

2. How to use TSP

Basic usage of the TSP package is to simplify preparation of input decks for computer codes with limited syntax possibilities. Instead of writing an input deck directly, a template can be written that generally has syntax and structure of the input deck, but also can include Python code snippets, which will be replaced in the resulting file. Thus one can e.g. define and use variables, include external files, even if these operations are not permitted by the input file syntax of the target computer code. Ultimately, one gets possibility to use the whole legacy of Python when writing an input deck.

The TSP preprocessor, an executable script called `ppp.py`, is used to process templates. If a template file is stored in file `input.t`, the following command:

```
$ ppp.py input.t
```

will generate the file called `input.t.res` that contains the result of processing.

The content of the template file is logically divided into two parts. The template's first line is used to set the delimiters for the snippets and to specify some other options. The rest of the template file, i.e. everything from its second line till the end of file, defines the content of the resulting file (it is called below **template body**). It is arbitrary text, interlaced with Python snippets, the latter must be marked with delimiters defined at the first line. Details see in Section *Structure of template file*.

There are two types of snippets, **evaluation snippets** and **executable snippets**. Evaluation snippets are substituted by the result of evaluation. Executable snippets are copied into the resulting file, but if any standard output is generated during their execution, it is appended to the snippet in the resulting file. For details see *Processing of snippets*.

There are options that control the appearance of sippets in the resulting file, see *Optional snippet keys*. The multi-line snippets can be commented out in the resulting file, see *Multi-line snippets*.

2.1 Installation

To install the TPS package, unzip (untar) the downloaded archive, go to the created directory containing file `setup.py` and run

```
$ python setup.py install
```

This command will install files into the default location, which for a Linux-based OS is in `/usr/local` and requires administrator priviledges. Alternatively, one can install the package locally, using the command line option `--user`:

```
$ python setup.py install --user
```

For details see[1].

The TSP package provides the script called `ppp.py`. During the installation process, the script will be copied into the directory where the Python interpreter is installed (under Windows it can be something like `c:\Python27\Scripts`) and thus will be available at the command prompt. The installation is sucsessful, if the script is available at the command line, and when called without a command line parameter, generates the following message:

```
$ ppp.py

(P)ython (P)re(P)rocessor: script from the tsp Python package.␣
 ↪Abbreviation
"tsp" means (T)ext with (S)nippets (P)reprocessor.

Usage:

> ppp.py template [snippet]

where `template' is a text file containing python snippets. ␣
 ↪Snippets are
evaluated/executed and the snippet code is replaced with the result␣
 ↪of
evaluation/execution. The resulting file is saved to `template.res'.

When optional `snippet` is given, it is evaluated or executed before␣
 ↪the
snippets in `template`.
```

Script `ppp.py` uses function `pre_pro()`, defined in the module `text_with_snippets` of the TSP package. Although the most common use of TSP is to call `ppp.py` script from the terminal command line, one can also import this function and use it from the Python interpreter or in a Pyhon script.

The name of the text file to be processed must be given to `ppp.py` as a command line argument. The output file will have the name of the input file, with suffix `.res`. So far the TSP package is installed, the `ppp.py` script should be availabe at the command line, and the command

[1] http://docs.python.org/install/index.html.

```
$ ppp.py input
```

should result in a new file called `input.res`.

2.2 Structure of template file

Generally, the content of a template file consists of the first line with information for the preprocessor, followed by arbitrary text interlaced with Python snippets.

Python snippets must be marked at the beginnig and at the end by the characters specified at the first line. Optionally, each snippet can be prefixed with a key that specifies details of how the snippet code and its evaluation/execution result appears in the resulting file. For available keys see *Optional snippet keys*.

Inside snippets, the snippet delimiting characters cannot be used, in all other respects a snippet must represent valid Python code.

2.2.1 The first line

The template's first line must contain information about snippet delimiters. Optionally, it can also contain the commenting string, which will be used to comment out the multi-line snippets in the resulting file, and the default key, which will be applied to all snippets.

The snippet delimiting characters are the last two characters of the first line with trailing spaces stripped off. They must be specified, i.e. the first line must have at least two characters. An example of an minimalistic first line is shown on the following template

```
''
Simple text and snippets: 'i=1', 'i'.
```

In this template, the apostrophe, `'`, is used to mark both the beginning and the end of the snippets. Any characters can be used as delimiters, although in the case of alphanumeric delimiters, a warning message will be issued. In the following template, character `y` is used to mark the snippet's begin, and characher `z` – to mark the snippet's end:

Template	Resulting file

`yz`
`Alpha-numeric characters can`
`be used as snippet's delimiters`
`as well, but a warning message`
`will be issued.`
 	
`ya=1z, ya*2z.`
	`Alpha-numeric characters can`
	`be used as snippet's delimiters`
	`as well, but a warning message`
	`will be issued.`
	
	`ya=1z, 2.`

This template is processed with the following warning message:

```
Process file alphanum.t
WARNING: the delimiter specified on the first line of
         the template, is alpha-numeric or blank.

         COMMENTING STRING:
         STARTING DELIMITER: y
         ENDING DELIMITER: z
WARNING: string of commenting characters
         is empty.  Multi-line  snippets
         will not be commented out
Result is written to alphanum.t.res
```

After the delimiters are read from the first line, it is searched for a substring representing a valid key (possible keys see in *Optional snippet keys*). If such substring is found, it will be used as the default snippet key. The following example shows the first line specifying `-l` key as the default one:

```
-l''
All snippets in this template will have the -l option,
if no other key is specified.
```

Only the first substring matching one of the available keys will be taken into account.

Additionally, in the first line one can define a string of commenting characters. It is used to comment out multi-line snippets in the resulting file. The commenting string is the rest of the first line after stripping the trailing spaces, reading (and removing) the delimiting characters and reading (and removing) the default key. The commenting string can be empty, as in the examples above, in this case the multi-line snippets will appear in the resulting file as is, without commenting out. In this case a warning is issued.

Example 1

In the following example, the commenting string consists of character `c` followed by one space, the delimiters are quotes `'`:

Template	Resulting file
`c ''`	`c`
`c -d'dr=1'`	`c dr is 1 cm.`
`c dr is '  dr  ' cm.`	`c '`
`c '`	`c   print`
`  print`	`c   for i in range(5):`
`  for i in range(5):`	`c     print "%2i cz %6f" % (i,␣`
`    print "%2i cz %6f" % (i,␣`	`↪i*dr)'`
`↪i*dr)'`	` 0 cz 0.000000`
	` 1 cz 1.000000`
	` 2 cz 2.000000`
	` 3 cz 3.000000`
	` 4 cz 4.000000`

This template describes a part of an MCNP input file, where `C` or `c` character at the first five positions of a line followed by a space means that this line is a comment.

Example 2

This example differs from the previous one by the first line only, where the default snippet key is set to `-l`:

Template	Resulting file
`c -l''`	`c`
`c -d'dr=1'`	`c dr is 1        cm.`
`c dr is '  dr  ' cm.`	`c '`
`c '`	`c   print`
`  print`	`c   for i in range(5):`
`  for i in range(5):`	`c     print "%2i cz %6f" % (i,␣`
`    print "%2i cz %6f" % (i,␣`	`↪i*dr)'`
`↪i*dr)'`	` 0 cz 0.000000`
	` 1 cz 1.000000`
	` 2 cz 2.000000`
	` 3 cz 3.000000`
	` 4 cz 4.000000`

The commenting string is `'c '`, the default key is `-l` and both delimiters are `'`. This example is similar to the previous one, except the default positioning of the snippet evaluation results is changed. By default, the space allocated by a snippet is not preserved (the `-D` option), here we specified another default behaviour: if the snippet evaluation result is shorter than the snippet itself, the result will be adjusted left (the `-l` option). With this option, the resulting file can be more similar to the template file, as compared with the resulting file obtained with default `-D` option.

Note that the key specified in the first line defines the default key; each snippet still can be preceded with its own key that overwrites the default one.

Example 3

The commenting string is `# -r`, the default key is `-l`, the beginning delimter is the back quote, `` ` ``, and the end delimiter is the single quote, `'`:

Template	Resulting file
-l# -r`'	`
`	# -r # multi-line snippet
# multi-line snippet	# -r a = 1
a = 1	# -r print
print	# -r print a
print a	# -r'
'	1
Check default key: ` a '.	
	Check default key: 1 .

This example illustrates that the default key and the commenting string can be specified in arbitrary order. Note that only the first substring that matches the key pattern is taken into account. The rest of the line (after removing the delimiters and the default key) is considered as the commenting string.

2.2.2 Template's body

Everything after the template's first line constitutes the template's body that contains arbitrary text and Python snippets.

Inside the template body, snippets must be delimited at the beginning and at the end by characters specified in the first line. The beginning delimiter can be preceded with any other character, and the end delimiter can be followed by any other character. In the following example, text within quotes will be recognized by the preprocessor as a snippet:

Template	Resulting file
''	'a="R"' Template can start with
'a="R"' Template can start with	a snippet. Snippet can appear
a snippet. Snippet can appear	inside a woRd.
inside a wo'a'd.	

A snippet prefix consisting of the minus sign followed by a letter can have special meaning. If it coincides with one of the snippet keys (see in *Optional snippet keys*) , the snippet representation in the resulting file will be changed and the prefix will not appear in the resulting file. Examples of snippet prefixes:

Template	Resulting file

`''`
`Snippet without any prefix: 'a=1`
`↪'`
`Snippet with key -l:  -l'a'.`
 	
`The following prefix does not`
`correspond to any snippet key,`
`it will be considered as usual`
`text: -M'a*2'.`
	`Snippet without any prefix: 'a=1`
	`↪'`
	`Snippet with key -l:    1   .`
	
	`The following prefix does not`
	`correspond to any snippet key,`
	`it will be considered as usual`
	`text: -M2.`

The snippet delimiter characters cannot be used inside snippets, since they would be considered by the preprocessor as the beginning or end of the snippet. Usually, this limitation can be simply overcome by choosing proper delimiting characters.[2]

All snippets are evaluated or executed in the common namespace. This means that a variable defined in one snippet will be “seen” in the following snippets. Similarly, modules loaded in a snippet will become available in all consequent snippets. This is also valid for inclusion of external templates: when a snippet calls the `pre_pro()` function to process another template, all variables defined in the processed template will become available in the “calling” template.

2.3 Processing of snippets

A snippet can be evaluated or executed. When a snippet is found, the preprocessor first tries to evaluate it using the `eval`[3] Python built-in function, and if the evaluation causes the `SyntaxError`, the snippet will be executed using the `exec`[4] Python built-in function. Thus, if a snippet code is an expression[5] (for example, `b`, `a + b` are expressions) it will be evaluated, and if a snippet is a statement[6] (or several statements taking several lines), it will be executed.

When a snippet is evaluated, the string representation of the evaluation result will substitute the snippet code in the resulting file. When a snippet is executed, its code is repeated in the resulting file and standard output generated by the snippet will be

[2] Note that in Python one can use interchangeably single and double quotes for strings.
[3] http://docs.python.org/library/functions.html#eval
[4] http://docs.python.org/reference/simple_stmts.html#exec
[5] http://docs.python.org/reference/expressions.html
[6] http://docs.python.org/reference/simple_stmts.html

caught and put right after the snippet code. This default behaviour can be modified to some extent by the snippet keys, described in the section *Optional snippet keys*.

2.3.1 Evaluation examples

When a snippet is evaluated and the evaluation result substitutes the snippet code in the resulting file, several situations are possible. If the resulting string is longer or shorter than the snippet code, the text on the line after the snippet will be shifted in the resulting file:

Template	Resulting file

`c ''`
`Let c be a long string: 'c=`
`↪"abcde"'.`
`Evaluating c, 'c', will shift␣`
`↪the`
`rest of the line.`
 	
`Let define a variable with a␣`
`↪very`
`lond name, "LongVariable", and`
`set it to very short string␣`
`↪value:`
`'LongVariable="a"'.`
`Evaluating it, 'LongVariable',␣`
`↪will`
`also shift the rest of the line.`
	`Let c be a long string: 'c=`
	`↪"abcde"'.`
	`Evaluating c, abcde, will shift␣`
	`↪the`
	`rest of the line.`
	
	`Let define a variable with a␣`
	`↪very`
	`lond name, "LongVariable", and`
	`set it to very short string␣`
	`↪value:`
	`'LongVariable="a"'.`
	`Evaluating it, a, will`
	`also shift the rest of the line.`

Note, in the template's 3-rd line, the snippet code `'c'` takes 3 characters (the delimiters are counted as well) and it is substituted with a string of 6 characters `abcdef` in the result. In the 9-th line the snippet `'LongVariable'` that takes 14 characters in the template, is replaced with only one character `'a'`.

If the evaluation result is shorter than the snippet code, the default behaviour can be changed using *Optional snippet keys*. If the resulting string and the snippet code have the same length, the text after the snippet will remain its position.

2.3.2 Execution examples

In the following example, the first snippet, `'a="abc"'` is executed. The second snippet that is just the variable reference, `'a'` is evaluated, and the third snippet that contains the `print` statement, is again executed:

Template	Resulting file
`c''`	`First, define variable: 'a="abc"`
`First, define variable: 'a="abc"`	` ↪'.`
` ↪'.`	`This snippet will be executed.`
`This snippet will be executed.`	
	`In the follooiwng snippet, a is`
`In the follooiwng snippet, a is`	`just evaluated: abc, and the`
`just evaluated: 'a', and the`	`evaluation result substitutes`
`evaluation result substitutes`	`the snippet code.`
`the snippet code.`	
	`The standard output from snippet`
`The standard output from snippet`	`execution will be put right␣`
`execution will be put right␣`	` ↪after`
` ↪after`	`the snippet: 'print a'abc`
`the snippet: 'print a', like␣`	`, like here.`
` ↪here.`	

Note that by default, the print[7] Python statement adds the new-line character at the end of the printed line, therefore in the above example the rest of the line after the last snippet jumped onto the next line. To avoid the new-line character, the comma-ended variant of the print statement should be used, e.g. `'print a,'`.

2.3.3 Multi-line snippets

A snippet can take several lines, in this case it is always executed. The usual indentation rules for Python code should apply to the multi-line snippets. Consider the following template:

Template	Resulting file
`c ''`	`This is a multi-line snippet: '`
`This is a multi-line snippet: '`	`c     a = 1`
`    a = 1`	`c     b = 2`
`    b = 2`	`c     print a, b'1 2`
`    print a, b'`	

The multi-line snippet here has 4 lines. The first line of the snippet is empty: it starts right after the opening apostrophe and goes till the end of the line. The 2-nd, 3-rd and 4-th lines contain Python statements that are indented, i.e. start at 5-th position. The indentation of the first statement is optional and used in this example to visually distinguish the snippet from the surrounding text. This indentation will be removed by the preprocessor before passing the snippet to execution. Indentation of the following statements should follow the Python rules.

[7] http://docs.python.org/reference/simple_stmts.html#print

The code can start on the first snippet line as well, but in this case one should take care about the proper indentation. The following example would cause the indentation error, since the first code line has no indentation and the other lines are indented (the snippet execution will be aborted so that no standard output will be generated):

Template	Resulting file
`c ''`	`A multi-line snippet: 'a = 1`
`A multi-line snippet: 'a = 1`	`c     b = 2`
`    b = 2`	`c     print a, b'`
`    print a, b'`	

This template would produce the following output:

```
Process file mline2.t
WARNING: Snippet on line 2 caused execution error:
         unexpected indent (<string>, line 2)
         <type 'exceptions.IndentationError'>
Result is written to mline2.t.res
```

The right way to put the code into the first snippet line is:

Template	Resulting file
`c ''`	`A multi-line snippet: '    a = 1`
`A multi-line snippet: '    a = 1`	`c     b = 2`
`    b = 2`	`c     print a, b'1 2`
`    print a, b'`	

Note that this form is visualy less clear than in the first example in this section, so it is better to avoid putting any code to the first line of the multi-line snippet.

Multi-line snippets are always executed. By default, the code of the multi-line snippets is repeated in the resulting file (for the other behaviour use `-d` key, see *Optional snippet keys*), and if any standard output is generated by the snippet, it is placed right after the snippet's end delimiter, on the same line.

When a multi-line snippet is repeated in the resulting file, all lines of the snippet code, except the first one, are commented out by appending the string of commenting characters to every snippet line. The commenting string is defined in the template first line, see *The first line*. In the following template, the two-characters string `'c '` (character `c` followed by the space) will be used to comment mulitline snippets:

Template	Resulting file

`c ''`
`a multiline snippet:'`
`    # this is usual python␣`
`↪script`
`    a = 1`
`    b = 2`
`    c = "abc"'`
 	
`The variables defined in the`
`snippet above: 'a', 'b', 'c'.`
	`a multiline snippet:'`
	`c     # this is usual python␣`
	`↪script`
	`c     a = 1`
	`c     b = 2`
	`c     c = "abc"'`
	
	`The variables defined in the`
	`snippet above: 1, 2, abc.`

The string of commenting characters defined in the templates's first line can be empty. In this case a warning message will be printed and multi-line snippets will not be commented out in the resulting file.

The standard output of the multi-line snippet is added right after the snippet code, on the same line. In this case the first line of the output can become commented out, like in the following example:

Template	Resulting file

`c ''`
`A multi-line snippet to␣`
`↪generate a`
`list of increasing numbers:'`
`    for i in range(3):`
`        print i`
`'`
	`A multi-line snippet to␣`
	`↪generate a`
	`list of increasing numbers:'`
	`c     for i in range(3):`
	`c         print i`
	`c '0`
	`1`
	`2`

Sometimes, it is desired to put the snippet output on extra lines. To do this, use additional `print` statements in the snippet code. The above example would be:

Template	Resulting file
`c ''`	`A multi-line snippet to generate a`
`A multi-line snippet to generate a`	`list of increasing numbers:'`
`list of increasing numbers:'`	`c     print`
`    print`	`c     for i in range(3):`
`    for i in range(3):`	`c         print i`
`        print i`	`c '`
`'`	`0`
	`1`
	`2`

2.3.4 Optional snippet keys

Optionally, a snippet can be prefixed with a key. A key consists of the minus sign followed by a character, and must precede the snippet starting delimiter without any blanks. The default key can be specified at the template first line. It will apply to all snippets without keys.

Currently, the following keys are defined:

Key	Meaning
`-D`	default. Do not preserve place
`-r`	adjust evaluation result right
`-l`	adjust evaluation result left
`-c`	center evaluation result
`-d`	delete snippet code
`-s`	skip snippet evaluation

Keys `-D`, `-r`, `-l`, and `-c` specify how the result of snippet evaluation should be aligned within the place allocated by the snippet code (if no key is given before the snippet and no default key is specified at the first template line, the `-D` key will apply, i.e. the snippet's place will not be preserved). These keys have meaning only if a snippet is evaluated, i.e. when the evaluation result should fit into the place allocated by the snippets code.

Key `-d` will delete the snippet code, so that the code will not enter the resulting file. In the case of snippet evaluation, the snippet will be evaluated, but nothing will go to the resulting file. If the snippet is executed, the resulting file will contain only the standard output of the execution, not the snippet code itself.

Key `-s` specifies that the snippet should not be evaluated nor executed. In this case, the snippet code is simply copied into the resulting file.

Snippet keys, except `-s`, do not appear in the resulting file.

The meaning of the keys should be clear form the following example:

Template	Resulting file
`c -D''`	`The defalut key can be specified`
`The defalut key can be specified`	`in the first line. If no key is`
`in the first line. If no key is`	`given, the default is -D.`
`given, the default is -D.`	
	`Example of snippet keys:`
`Example of snippet keys:`	
	`|        2| adjust right`
`|-r'  1+1'| adjust right`	`|  2      | adjust left`
`|-l'  1+1'| adjust left`	`|     2   | adjust center`
`|-c'  1+1'| adjust center`	`|  2| do not preserve place`
`|-D'  1+1'| do not preserve␣`	`|  2| no key: use default`
`↪place`	`|| delete snippet`
`|  '  1+1'| no key: use default`	`|-s'  1+1'| skip evaluation.`
`|-d'  1+1'| delete snippet`	
`|-s'  1+1'| skip evaluation.`	

3. Examples

This section shows how the TSP package can be applied to tasks that often arise during preparation of complex input files. This covers inclusion of external files and templates and generation of a set of input files that differ by some parameter (for e.g. parametric studies).

3.1 Inclusion of external files

The TSP package provides no special mechanism to include external files into the resulting file, however, one can use standard Python capabilities. In the following example, an external file is read by the `open`[1] Python function and its content is printed to the standard output. Since the standard output of the snippet execution appears right after the snippet code, the file content will appear in the resulting file.

[1] http://docs.python.org/library/functions.html#open

Template	Resulting file

`c ''`
`A snippet to put the content of␣`
`↪the`
`external file "incl.txt" into␣`
`↪the`
`resulting file. For details see`
`description of the open() Python`
`function. '`
`    print`
`    for l in open("incl.txt", "r`
`↪"):`
`        print l,'`
 	
`Or as one-liner:`
`'print; print open("incl.txt").`
`↪read()'`
where `incl.txt` has the following content:
`1 This is the content`
`2 of file`
`3 incl.txt`
	`A snippet to put the content of␣`
	`↪the`
	`external file "incl.txt" into␣`
	`↪the`
	`resulting file. For details see`
	`description of the open() Python`
	`function. '`
	`c      print`
	`c      for l in open("incl.txt",`
	`↪"r"):`
	`c          print l,'`
	`1 This is the content`
	`2 of file`
	`3 incl.txt`
	
	`Or as one-liner:`
	`'print; print open("incl.txt").`
	`↪read()'`
	`1 This is the content`
	`2 of file`
	`3 incl.txt`

The `open` function returns a file object that can be iterated line by line; in the example above, the `for` loop iterates over all lines of the opened file. The file is opened with `"r"` key that means that the file is open only for reading. Each line of the opened file is printed out. Note that the print statement ends with the comma, this prevents extra empty lines in the output. The snippet starts with the empty print statement, it adds a new line just after the snippet's end marker in the resulting file, so that the included file starts on the new line.

3.2 Inclusion of other templates

While in the above example inclusion of text file without any preprocessing is shown, often it is necessary to include another template that needs first to be preprocessed. To accomplish this task one needs to use directly the function `pre_pro()` from the module `text_with_snippets` of the TSP package. In simple situations this function is called by the preprocessor `ppp.py` so that a user does not use it directly; this example shows how to use this function directly in a template (or in a Python script).

The `pre_pro()` function takes the name of a template file as argument and returns the corresponding resulting file content as a multiline string. This function is imported by the script `ppp.py` and thus is accessible inside snippets by default. To include the result of evaluation of another template, simply call this function and print out its result:

Template

```
c ''
Example of including another␣
 ↪template.
File "itempl.txt" is first␣
 ↪preprocessed
and its result is inserted here:

-d'print pre_pro("itempl.txt"),'

Note that variable A defined in␣
 ↪the
included template is available␣
 ↪here:
A = 'A'.
```

where `itempl.txt` has the following content:

```
//
    Included templates can have
    their own delimiters.

    For better representation,␣
 ↪the
    lines of the included␣
 ↪template
    are indented.

    Varible A is defined here:
    /A=5/.
```

Resulting file

```
Example of including another␣
 ↪template.
File "itempl.txt" is first␣
 ↪preprocessed
and its result is inserted here:

    Included templates can have
    their own delimiters.

    For better representation,␣
 ↪the
    lines of the included␣
 ↪template
    are indented.

    Varible A is defined here:
    /A=5/.

Note that variable A defined in␣
 ↪the
included template is available␣
 ↪here:
A = 5.
```

In this example, the snippet code is removed from the resulting file by applying the `-d` snippet key. The snippet's output – the result of preprocessing of the template `itempl.txt` is thus written instead of the snippet. To remove the extra new-line after the inserted template, the `print` statement is ended with comma.

3.3 Multiple resulting files for parametric studies

Sometime it is necessary to prepare many input files differing from each other by some parameters. This task can be automatized without writing many template files or manually changing parameters in the template.

Since one can process a template file from another template, one can call the main template from within a wrapper, where certain value of parameter is set. Let say we want to prepare three input files described by the following template, so that the input files differ only by the value of parameter `N`:

```
''
c Input file for 'N = 1'.
c Some lines depending on
c the value of 'N'.
```

Three different input files can be created by sequentially changing the value of `N` in the template, processing the template and saving the resulting file under a unique name. Alternatively, one can process this template from another one, and write the result of processing not to standard output, but to a file. Let the above template is saved in the file `main.t`. The wrapper template could look like:

Template	Generated files

```
''
Here we loop over three values␣
 ↪of N.
For each N, the main template is
processed and the result is␣
 ↪written
into a file. All this is done␣
 ↪in the
following snippet:
'
for N in ["V1", "V2", "V3"]:
    f = open( "input_"+N, "w")
    f.write(pre_pro("main.t"))
    f.close()
'
```
	The first generated file, `input_V1`:
	```
	c Input file for -s'N = 1'.
	c Some lines depending on
	c the value of V1.
	```
	the last generated file, `input_V3`:
	```
	c Input file for -s'N = 1'.
	c Some lines depending on
	c the value of V3.
	```

where `main.t` is

```
''
c Input file for -s'N = 1'.
c Some lines depending on
c the value of 'N'.
```

Note that the main template was slightly modified: the snippet defining the value of `N` is disabled by the `-s` key. It is not needed here anymore, since `N` is defined in the wrapper template. Processing the wrapper template with `ppp.py` will result in three files named `input_V1`, `input_V2` and `input_V3` that will differ from each other only by the value of variable `N`.

4. Conclusion and acknowledgement

Basic usage of the TSP package is to simplify preparation of input decks for computer codes with limited syntax possibilities. Instead of writing an input deck directly, a template can be written, which generally has syntax and structure of the input deck, but also can include Python code snippets, which will be replaced in the resulting file. Thus one can e.g. define and use variables, include external files, even if these operations are not permitted by the input file syntax of the target computer code. Ultimately, one gets possibility to use the whole legacy of Python when writing input decks.

TSP was used by the author to generate input files for MCNP code [3] and KANEXT system [4] under Linux and Windows operating systems.

Current version of the package provides the following features:

- Snippet keys to define positioning of the result and to skip the snippet's evaluation.
- Multi-line snippets are optionally commented out in the resulting file.
- Snippet delimiters and the string of commenting characters can be set arbitrarily.
- Warning messages if alpha-numeric characters are used as snippets delimiters. This should warn a user in the case he (she) forgets to specify the template's first line.
- Warning messages that specify line number of a snippet that cannot be executed.

The package is currently not actively developed, since it already provides features necessary to prepare input decks effectively. However, future improvements can include the following:

- more flexible format to define snippets output formatting, like default formatting for integers, floats, etc.
- Specification of the default snippet key, commenting characters and delimiters in the preprocessor command line rather than in the template's first line.

The TSP Python package has been written during work on several projects funded by the European Commission: ELSY [5], LEADER [6] and CDT [7].

Bibliography

[1] “Literate Programming,” http://www.literateprogramming.com.

[2] “Python Programming Language – Official Website,” http://www.python.org.

[3] X-5 Monte Carlo team, *MCNP – A general N-Particle transport code, Version 5 – Volume I: overview and Theory*, Los Alamos National Laboratory, 2003.

[4] M. Becker, S. Criekingen, and C. Broeders, “KArlsruhe PROgram System KAPROS and its successor KArlsruhe Neutronic EXtentable Tool KANEXT,” http://inrwww.webarchiv.kit.edu/kanext.html, last accessed September 2013.

[5] A. Alamberti et al., “ELSY – European LFR Activities,” *Journal of Nuclear Science and Technology*, **48:4**, 479-482 (2011).

[6] “LEADER Lead-cooled European Advanced DEmonstration Reactor,” http://www.leader-fp7.eu/.

[7] P. Baeten, H. A. Abderrahim, and D. DeBruyn, “The next step for MYRRHA: the central design team FP7 project,” *Proc. AccApp’09: International Topical Meeting on Nuclear Research Applications and Utilization of Accelerators*, VIenna, Austria, 2009.