

## pnw – A text manipulation and literate programming tool written in python

Matthias Lesch

**ABSTRACT.** pnw is a text processing tool with a wide range of applications. It can be used to compile chunks of text from various sources into a new text file. The advantage over cut-and-paste techniques is that changes in the source file are automatically followed. Secondly, we provide a thin wrapper around the pandoc document converter. Therefore, chunks of text can be converted from and to a wide range of lightweight markup languages as well as html and tex/latex. Although literate programming has become a little out of fashion, pnw can be used as a literate programming tool, too. Not surprisingly, pnw itself and its documentation is written using pnw. pnw is remotely inspired by noweb and heavily inspired by (even borrowing part of the syntax from) [antiweb](#). An important feature of pnw, which is not present in antiweb, is the support of namespaces. This allows to create reusable *block libraries* which can be imported and addressed in a *dot-notation syntax* similar to python modules.

A pnw file is a text file with tags defining named blocks of text. In a way this is similar to an xml file. Tags may be hidden behind comment characters, such that the file remains processable by another program. The blocks of text from various pnw files can be processed and rendered by another pnw file.

### CONTENTS

1. <a href="#">pnw language syntax</a>	2
2. <a href="#">Usage</a>	2
3. <a href="#">A more detailed example</a>	4
4. <a href="#">Implementation details</a>	5
<a href="#">References</a>	13

## 1. pnw language syntax

**1.1. pnw tags.** A line is recognized as a pnw tag if it is of the form

```
<whitespace><one non-word character>@tagname(args,kwargs)
```

The indent (length of the whitespace) is significant for some tags. The non-word character is completely arbitrary. Typically one might use a language specific comment character (% for latex, # for python) here. In the following we either use #, the python comment character, or omit it altogether and write tags in the form @tagname. Here is a quick summary of the available pnw tags:

**#@block(name,args,kwargs):** Starts block name; one may also write chunk instead of block, they are synonymous. A block ends at

- the start of another block or an #@end with a smaller or equal indent,
- another #@ tag with a smaller indent,
- the end of the file.

Several blocks may have the same name if they have the same indent. Then these blocks are combined into a *chunk* of that name and the individual subblocks are named name1, name2 etc.

**Beware:** due to namespace notation, Block names may NOT contain dots.

**#@end:** Explicit end of a block, indent is significant.

**#@import(fname,nspace):** Imports file fname into nspace. This means that the blocks and chunks in fname can be addressed as nspace.name. Within fname blocks/chunks can address each other without nspace prefix. This is analogous to the python import statement.

**#@include(name):** Includes the block/chunk name instead of this line. The indent is significant: the indent of the #@include| statement is added when rendering the content of block/chunk name.

All other lines outside of blocks are ignored. The content of each namespace nspace is collected in the block (resp. chunk if several files are imported into the same namespacefile fname) nspace.

**1.2. options, markup languages.** The syntax for the block/chunk tag is similar to the syntax for python functions. In detail

```
chunk(arg1,arg2,...,argn,kwarg1=value1,...,kwargm=valuem)
```

So far the following options are available

**remove:** Removes block completely from its parent block.

Furthermore, the following keyword arguments are observed:

**format={pandoc format}:** Specifies the formatting of the following block. Any format which pandoc can handle is allowed. E.g. if a block is markdown formatted and it includes another block using the include directive, the program tries to convert that block into markdown as well.

## 2. Usage

**2.1. CLI.** Command line usage is as follows

```
pnw <options> <file>
```

where `<file>` is the pnw file. The following options are available

- R, -chunk `<cname>`: Extracts chunk `cname` and renders it.
- F: Renders whole file as one chunk.
- f, -format: Output format, can be any format supported by pandoc.
- t, -tree: Show document tree.
- d, -depth: Traverses the tree up to depth `d`. Nodes of depths `d` are indicated as `<<name>>`, as in noweb, and not expanded any further.

2.1.1. *Example.* The command

```
pnw -Rmoduledoc -flatex pnw.pnw > pnw.tex
```

renders the chunk `moduledoc` from the source file `pnw.pnw` in latex format and writes the result to the file `pnw.tex`.

2.2. **python module.** The public interface consists of the following functions and classes,

```
class Block():
    def __init__(self, text, fname, lineno, parent_block = None, parent_chunk = None):
class Chunk(Block):
    def __init__(self, name, block):
def registerblock(name, block):
def pnwimport(fname, nspace, line=None):
```

We have the following global variables:

- paths:** Search path for files
- imports:** Dictionary mapping imported filenames to their associated namespaces
- chunks:** Dictionary of Chunks and Blocks
- roots:** Set of Root Nodes
- rootchunks:** Set of Root Chunks

2.2.1. *function pnwimport.* This is the main function for using pnw as a python module. The file `fname` is parsed and its blocks/chunks are read into the namespace `nspace`. The argument `line` is used internally for conveying error information and it can be ignored by external user.

Indent is taken care of in chunk defs, not in imports, so the indentation of import statement is irrelevant.

2.2.2. **class Block.** This is the base class for nodes of the document tree. For each line of a pnw document an instance of `Block` is created. A `Block` needs the following default information:

- text:** The (line of) text
- fname:** The name of the file
- lineno:** Line number of the text line

**Important attributes:**

- parent\_block:** the parent block in the document tree
- parent\_chunk:** if several blocks have the same name they are combined into a Chunk. This attribute points to the parent Chunk
- children:** list of child nodes
- args,kwargs:** list of args, resp. dictionary of keyword args found in () after the tag

**tag:** name of the tag

**indent:** indentation (int) of the block/chunk

**Public methods:**

**def** expand(self,indent=0,depth=0,toformat=None):

**def** render(self,indent=0,depth=0,toformat=None):

expand returns a list of lines, render just joins them into a string. indent and toformat are hopefully self-explanatory. depth is the recursion level up to which includes are expanded. Beyond that level include(foo) is rendered as <<foo>> with the correct indentation.

2.2.3. **class** *Chunk(Block)*. A Chunk is a node which is outside the normal document tree. A Chunk cname is created when a second Block cname is found. All n Blocks of name cname become children of the newly created Chunk. The Blocks are renamed cname<sub>i</sub> where i runs from 1 to n.

2.2.4. *function registerblock*. Fills the dictionary chunks and takes care of the tree structure. If the name does not yet exist, the block is added to the dictionary chunks with key name. Otherwise, if necessary, a new chunk name is created and the block is appended to its list of children.

2.2.5. *Example*. Assuming that pnw is installed, from the main directory of the pnw package open an interactive python session, I recommend to use ipython, and write

```
import pnw
```

```
pnw.pnwimport('pnw.pnw','')
```

Now you can explore the pnw document tree of the pnw.pnw source file by inspecting the dictionary

```
pnw.chunks
```

To see the structure of a block, just print its repr

```
pnw.chunks['moduledoc']
```

To render it, try

```
print pnw.chunks['moduledoc'].render(indent=0,depth=2,toformat='rst')
```

### 3. A more detailed example

The pnw source consists of one file pnw.pnw. It is a working python file containing pnw tags hidden behind the # comment character. To produce a clean python file we invoke (compare the makefile in the main directory)

```
python ./pnw.pnw -F pnw.pnw > pnw.py
```

The -F directive means that the whole file is treated as one block named u". The previous command is used when installing pnw for the first time.

The main driver file for the documentation is pnw-doc.pnw in the doc directory. Here is the head of this file

```
@chunk(main,format=latex)
@path(..../..)
@path(..)
@import(latex-blocks.tex)
@import(md-blocks)
@import(pnw.pnw)
@include(ltxheader)
```

First the whole file is declared to be a block named *main* which is latex formatted. Then we add the first two parent directories to the search path for imports. Then the files latex-blocks.tex, md-blocks, pnw.pnw are included into the main namespace u''. The block ltxheader, which contains what its name says, and which can be found in latex-blocks.tex is included.

pnw-doc.pnw contains a little of the documentation written in latex. Mostly, it includes blocks from other files. The bulk of the documentation, written in markdown, can be found in md-blocks.

Finally, an executable latex file is produced by invoking

```
pnw -Rmain pnw-doc.pnw > pnw-doc.tex
```

## 4. Implementation details

### 4.1. Parsing the document tree.

#### 4.1.1. Constants. Constants

```
""" The strings in 'options', 'keywords', and 'texttypes'
are inserted into the module's namespace as uppercase. I.e.
COLLAPSE = 'collapse' etc.
"""
```

```
options = [u'collapse',u'remove']
keywords = [u'chunk',u'block',u'end',u'import',u'include',u'path']
texttypes = [u'text',u'empty']
tags = list(keywords)
tags.extend(texttypes)
reserved_words = list()
reserved_words.extend(keywords)
reserved_words.extend(texttypes)
reserved_words.extend(options)
```

```
for item in reserved_words:
    globals()[item.upper()] = item
```

#### 4.1.2. pnwimport. Code for the function pnwimport

```
def pnwimport(fname,nspace,line=None):
    if line is None:
        line = Block(u'Command_Line','CLI',o)
    imports[fname] = nspace
```

```

f = None
for path in paths:
    tmpname = os.path.join(path, fname)
    try:
        f = open(tmpname, "r")
    except IOError:
        continue
    break
if f is None:
    raise PNWError(line, "Could not open file %s" % fname)
else:
    with f:
        linegen = enumerate(f.read().split('\n'), 1)
        rootblock = Block(u' #@chunk(%s)' % nspace, fname, lineno = 0)
        rootblock.indent = -1
        registerblock(nspace, rootblock)
        parse.block(linegen, rootblock)

```

#### 4.1.3. *parse\_block*. Code of *parse.block*, used by *pnwimport*

```

def parse_block(linegen, node):
    lineno, text = linegen.next()
    line = Block(text, node.fname, lineno, node)
    while True:
        if line.tag == IMPORT:
            imp_fname, imp_nspace = line.args[0:2]
            if imp_fname in imports:
                raise PNWError(line, "pnw does not support multiple \
..... imports of the same file, file already imported")
            pnwimport(imp_fname, imp_nspace, line)
        elif (line.tag in keywords and line.indent < node.indent) or \
            (line.tag in [END, BLOCK, CHUNK] and line.indent <= node.indent):
            # our block ends here
            return (line, lineno)
        elif line.tag in [BLOCK, CHUNK]:
            registerblock(line.args[0], line)
            if REMOVE not in line.args:
                node.append(line) # we need to append this node
            line, lineno = parse_block(linegen, line)
            continue
        elif (line.tag in [EMPTY, TEXT]) and line.indent < node.indent:
            raise PNWError(line, u'indentation error')
        else:
            node.append(line)
    if line.tag in [INCLUDE]:
        # is here, because node must get appended, list here those directives which need nspace adjustment
        line.args[0] = format_nspace(line.nspace, line.args[0])

```

```

# if we are here, we need to get a new line or close up
try:
    lineno,text = linegen.next()
    line = Block(text,node.fname,lineno,node)
except StopIteration:
    # end of file reached, add a #@end with correct indentaton
    lineno+=1
    line = Block(u'#@end',node.fname,lineno,node)
    line.indent = -1
    return (line,lineno)

```

#### 4.1.4. Class Block. Code of Class Block

```

class Block():
    def __init__(self,text,fname,lineno,parent_block = None, parent_chunk = None):
        self.text = text
        self.name = u''
        self.fname = fname
        self.lineno = lineno
        self.parent_block = parent_block
        self.parent_chunk = parent_chunk
        self.children = []
        self.parse()

    @property
    def nspace(self):
        return imports[self.fname]

    @property
    def format(self):
        return self.kwargs.get('format')

    def append(self,block):
        block.parent_block = self
        self.children.append(block)

    def expand(self,indent=0,depth=0,toformat=None):
        # this is really ugly, find better solution that
        # fiddling with negative indents
        if self.indent < 0:
            selffind = 0
        else:
            selffind = self.indent
        res = []
        if self.children and depth > 0:
            for child in self.children:
                res.extend(child.expand(indent+child.indent-selffind,depth-1,toformat=self.format))

```

```

else:
    if self.tag == TEXT: # return verbatim
        res.append(u'␣' * indent + self.text[selfind:])
    elif self.tag == EMPTY:
        res.append(self.text)
    elif self.tag == INCLUDE:
        # ch gets expanded *instead* of self, thus depth is same
        ch = chunks[self.args[0]]
        res.extend(ch.expand(indent,self.depth or\
            depth,toformat=toformat))
    if self.tag in [BLOCK,CHUNK]:
        res.append(u'␣' * (indent) + '<' + self.name + '>\n')
tmp = u'\n'.join(res)
if self.tag in [BLOCK,CHUNK] and toformat and self.format:
    tmp = convert(tmp,toformat,self.format)
    res = tmp.split('\n')
return res

def render(self,indent=0,depth=0,toformat=None):
    return u'\n'.join(self.expand(indent,depth,toformat))

def parse(self):
    #TODO arbitrary non-word char instead of only % #
    m=re.match("(?P<indent>\s*)(%|#)@(P<tag>\w*)(\((?P<args>.*\))\){0,1})",self.text)
    if m:
        d = m.groupdict()
        tag = d['tag'] #m.group(2)
        if tag not in tags:
            logger.error(errmsg(self,"Unknown_Tag_%s"%tag))
        self.tag = tag
        if d['args']:
            args,kwargs = parseargs(d['args'])
        else:
            args = [u'',u'']
            kwargs = dict()
        self.indent = len(d['indent'])
        self.args = args
        self.kwargs = kwargs
        while len(self.args) < 2:
            self.args.append(u'')
        if self.tag in [BLOCK,CHUNK]:
            self.name = args[0]
            self.tag = BLOCK
        # tags we process just here on the fly
        if self.tag == PATH:
            paths.insert(0,args[0])
        if self.tag == INCLUDE:

```



```

        if self.args[1]:
            try:
                self.depth = int(self.args[1])
            except:
                raise PNWError(self, "optional argument of\
.....include tag must be a nonnegative integer")
        else:
            self.depth = None
        return (tag, args, self.indent)
    elif m is None:
        if self.text.strip() == u'':
            self.tag = EMPTY
            self.indent = 256
        else:
            self.tag = TEXT
            self.indent = len(self.text) - len(self.text.lstrip())
        return None

def _repr(self, depth=256, indent=0):
    res = []
    if self.tag not in [TEXT, EMPTY]:
        try:
            res.append(u'_' * indent + u'——<BLOCK_ %s: %s: %i: %s: %s_\'%\
                (self.name, self.fname, self.lineno, self.tag, \
                self.text.rstrip("\n").strip()) + u'>')
        except:
            #strace()
            pass
    if depth > 0:
        for child in self.children:
            res.extend(child._repr(depth-1, indent+4))
    return res

def __repr__(self):
    if self.tag in keywords:
        return u'\n'.join(self._repr())
    else:
        return u'——<TEXT_ %s: %i: %s: %s_\'%(self.fname, self.lineno, \
        self.tag, self.text.rstrip("\n").strip()) + u'>'

```

#### 4.1.5. Class Chunk. Bla

```

class Chunk(Block):
    def __init__(self, name, block):
        Block.__init__(self, block.text, block.fname, block.lineno)
        self.__id__ = CHUNK
        self.indent = block.indent # cannot be parsed from block.text,

```

```

# since we play around with indent -1
self.name = name
self.tag = CHUNK
self.append(block)

def append(self,block):
    if block.indent != self.indent:
        raise PNWError(block,"All blocks in a chunk must have the same\
            indentation level")
    block.parent_chunk = self
    self.children.append(block)

def _repr(self,depth=256):
    res = []
    res.append(u'<Chunk_%s:%s:%i:%s:_%s'%(self.name,self.fname,\
        self.lineno,self.tag,self.text.rstrip('\n'))+u'>')
    for child in self.children:
        res.extend(child._repr(depth = depth))
    return res

def _repr__(self):
    return u'\n'.join(self._repr())

```

#### 4.2. The CLI interface. Bla

```

def CLI():
    import argparse
    parser = argparse.ArgumentParser(description =cli.description,
    usage="pnw_options_[Filename]")
    parser.add_argument("nw",help="The pnw file")
    parser.add_argument("-R","--chunk",help="Extract and render chunk",type = str, default = '')
    parser.add_argument("-F","--file", help="Render whole file",action = 'store_true')
    parser.add_argument("-t","--tree", help='Show Doc Tree',action = 'store_true')
    parser.add_argument('-f','--format', help = u'Output format',
        type = str, default = None)
    parser.add_argument("-d", help='depth of tree',type = int, default = 256)
    parser.add_argument('-a','--debug', help = 'show arguments and do\
nothing', action='store_true')
    args=parser.parse_args()
    pnwimport(args.nw,u'')
    if args.debug:
        print args
        raise SystemExit('Exit')
    elif args.tree:
        print showchunks(depth = args.d)
    else:
        print chunks[args.chunk].render(depth = args.d, toformat = args.format)

```

```
if __name__ == "__main__":
    CLI()
```

#### 4.3. Pandoc Interface. Bla

```
def dummyconverter(source,to,format):
    "dummyconverter"
    return source
```

```
formats = {'md': 'markdown',
           'tex': 'latex'}
```

```
def _convert(source,to,format):
    """pandoc converter"""
    format = formats.get(format,format)
    to = formats.get(to,to)
    p = subprocess.Popen(
        ['pandoc','--to='+to,'--from='+format,'--listings'],
        stdin = subprocess.PIPE,
        stdout = subprocess.PIPE)
    return p.communicate(source)[0]
# check for pandoc
try:
    p = subprocess.Popen(
        ['pandoc','-h'],
        stdin = subprocess.PIPE,
        stdout = subprocess.PIPE)
    pandoc = True
except OSError:
    logger.error("pandoc not available")
    pandoc = False
```

```
if pandoc:
    convert = _convert
else:
    convert = dummyconverter
```

#### 4.4. Helper and convenience functions. Bla

```
def format_nspace(nspace,name):
    """ if name contains a dot it is returned verbatim, otherwise
    returns nspace.name. """
    if u'.' in name or not nspace:
        return name
    else:
        return nspace+u'.'+name
```

```

def findroot(b):
    """ Traverses the doc tree upwards and returns the root node
    corresponding to 'b'."""
    while b.parent_block is not None:
        b = b.parent_block
    return b

def findrootchunk(b):
    """ Traverses the parent chunks upwards and returns the root chunk
    node corresponding to 'b'."""
    while b.parent_chunk is not None:
        b = b.parent_chunk
    return b

def findroots():
    """ Fills the sets 'roots' and 'rootchunks'."""
    for b in chunks.values():
        roots.add(findroot(b))
        b = findrootchunk(b)
        if isinstance(b,Chunk):
            rootchunks.add(b)

def parseargs(x):
    """ 'x' is a string of the form

        arg1,...,argn,kwarg1=value1,...,kwargm=valuem

    '(kw)argj' may *not* contain comma or '='. Positional args are
    collected in list 'args', keyword args are collected in dict
    'kwargs'. Returns '(args,kwargs)'."""
    rawargs=[item.strip() for item in x.split(',')]
    args=list()
    kwargs=dict()
    for item in rawargs:
        if not '=' in item:
            args.append(item)
        else:
            key,value = item.split('=')
            key = key.strip()
            value = value.strip()
            kwargs[key] = value
    return (args,kwargs)

```

## References

MATHEMATISCHES INSTITUT, UNIVERSITÄT BONN, ENDENICHER ALLEE 60, 53115 BONN, GERMANY  
*E-mail address:* `ml@matthiaslesch.de`, `lesch@math.uni-bonn.de`  
*URL:* `www.matthiaslesch.de`, `www.math.uni-bonn.de/people/lesch`