pym Documentation

Release 0.3.3

jalanb (Alan Brogan)

Nov 02, 2018

Contents

CHAPTER 1

What is pym?

pym helps with editing code, not text (java_rst).

pym knows what code is, from a programmer's POV: controlled flow of data through controlled manipulations.

Code is expressed in a language, such as Python or Bash, etc.

Code is parsed to an NST, which is annotated with interesting properties of the code.

An NST is a "Normal Syntax Tree", which links known names to known Syntax Trees. A "Syntax Tree" is provided by a (language-specific) parser, and can catch any runtime problems, providing source for each stack frame.

pym was inspired by vim but it handles structured (not plain) text. At present pym is entirely vapourware, and consists of some ideas based around mindful manipulation of Abstract Syntax Trees, such as those provided by the ast module and grako . It would help if the parser could handle bash script too.

1.1 Parsing

Parsing structured text should be more boring, but remains interesting, at least as a means of transforming plain to structured text. Rendering is ! parsing.

pym treats parsing and rendering as separate operations, but ideally they should be a single, reversible process. They will provide a "back end" to pym, connecting stored text, which is plain, to structured text which can be editted.

1.2 Editing

Editting is the heart of pym, a means of transforming ideas into structured text.

An editor is an interface between a person and structured text. The person has a keyboard and sees a window, tapping keys changes the window. The person first needs to learn which keys lead to which changes on screen. As soon as changes are made the screen shows diff from original.

Screen shows one dir, file or function (aka package, module or function) at a time.

We are stopped on a breakpoint, showing a function, with stopped line underlined in red. If there is an active error (aka Exception) then all (data leading to) data on this line is marked red. All code which has been run without error is shown in green, and all other data is shown in blue.

18 Keys: j/k (up/down (+/- Y)), h/l (backward/forward (-/+ X)), g/; (out/in (-/+ Z)) u/i (keyboard/screen), y/o (), t/p (take/paste) m/, (mark/goto), n/. (type/continue), b// (bookmark/search)

More leftward keys (on a QWERTY keyboard) "more general", "pull", "back"

More rightward keys mean "more specific", "push", "forward

Warning to vim users - some keys may seem the opposite of what you're used to.

1.3 What are we all doing here anyway?

I have a screen, you have a keyboard.

I have codes, etc., that I could show you, you have the means to choose between them. Assuming your capacity for choosing is infinite, I need to know which code you'd like to look at. So we'll start with disk structures- trees with directories and files. And some recognisable patterns. Sounds like a similar structure to real code (modules are trees with blocks and lines), but simpler. But you didn't hear that from me.

Anyway, structured text is ever a snapshot from a flow of your ideas to the codes being witten. On a good day the ideas flow toward some runnable tree which works, but on a bad day they chase around random forests, crashing blindly into insects. It is important to store correct program text, more important to grasp ideas behind that text, and most important to grok the flow.

An editor should appear in a debugger: accepting **'changes <>'_** from the coder and passing them on to an **'evaluator< >'_**, and incidentally to a disk. An editor is rarely an end in itself, and should not get in the way of the larger cycle. Hence an editor should be quick, and more efficient of the coder's time than other factors.

pym should look for the flow of ideas in the iterations of the REPL, noting steps such as when tests start to pass, and so development moves on.

pym should be editable by itself. This is a **high priority** - I do not have a lot of time for coding personal projects such as pym, the sooner it is "good enough" to be usable daily for editing other programs, but *quickly* fixable, then the more development it will actually get. (Such has been my experience with my dot files, in particular since I added a function to edit functions).

1.4 Modality

pym is inherently a modal editor, and one is not directly editing plain text. Some parts of the program will look more like plain text than others, e.g. names. But each structure in the tree uses its own specialised sub-editor, e.g. there is a different editor for an else branch than for a function definition than for a function. pym should transition between such editors unnoticeably to the user, not needing any "start loop here" instructions, although they could be explicitly given.

Command/insert mode might toggle on the CAPS LOCK key. pym is UI agnostic, capable of presentation between pipes, on a console, GUI, web page, or directly from Python. Development shall concentrate on Python first, console second, with others trailing. Full use should be made of available visual cues, such as colour, position, movement, ...

The user should be presented with program objects (e.g. projects, methods, operators) first and incidentals (e.g. files) second, if at all.

CHAPTER 2

Programs

A program is a history of a flow of ideas into a structured text representing a working algorithm.

Those ideas are *intentions* - what the coder wants to happen at run time.

CHAPTER 3

Usage

```
$ pym main.py
```

or

```
$ pym main.sh
```

You are shown a line of code in a function. You can move around: (A subset of) vim keys work, with the slight catch the ‘j’/’k’ moves down/up the call stack, ‘h’/’l’ moves left/right along the block of statements, and ‘g’/’;’ move out/in (through links to other code, e.g. function calls).

You can edit the code of the function, which will change the availability of run-time data.

Colours:

Errors shown in red, good code shown in green, links shown in blue. Untried code shown in white.

CHAPTER 4

Indices and tables

Parts of the documentation:

4.1 Parsing structured text

Parsing in general is of interest. One reason for delay in starting this project has been the postponing of the choice between using Python's built-in parsing modules, or more general parsing components. Although I have now decided to start with Python's built in parser I remain open to use of other parsers as well. Not much point in having a "structured text editor" which cannot handle structured text such as other langages, marked-up texts, config files, log files, and so on.

Parsing is more than merely interesting, it is required as long as storage is plain.

I also intend to rely as heavily as possible on others' code in developing pym, and can only expect that to arrive hither as plain text.

4.1.1 Parsing Python

Parsing produces Abstract Syntax Trees, which have a reasonable amount of related documentation. Green Tree Snakes was a proximate spur to this project, and the other links in this section were found thence.

- Green Tree Snakes - the missing Python AST docs (and some Reddit comments on them).
- The Design of CPython's Compiler lays out rationales behind some choices in the design of the compiler, and introduces a developer to the code supporting it. For pym it is particularly relevant in introducing ASDL and SPARK.
- Python internals: Working with Python ASTs

4.1.2 Parsing non-Python

Other parsers, written in Python but parsing other structures:

- parsley_page, which follows on from his Pymeta, and which he introduced at Pycon 2013.
- Ned Batchelder's Python parsing tools is a page covering many other parsers.
- Juancarlo Añez' Grako is "a tool that takes grammars in a variation of EBNF as input, and outputs memoizing (Packrat) PEG parsers in Python".

4.1.3 Meta-Compiling

HackerNews told me about META II, which is able to "reproduce its own code from a description":

At the end of these steps you should be able to:

1. Define a compiler description that ejects your favorite language code
2. Paste it into the Input window
3. Have one of the compilers on these web pages generate the translation to your favorite language
4. Cut the code generated in your favorite language from the Output window
5. Continue compiler and metacompiler development in your favorite language

Unfortunately I have other work to get back to now, but the tutorial is well-written, enthusiastic about the meta-compiler, and when the tutorials don't work, it's because they are meant not to, in the next paragraph.

Excellent site, I hope to get more time on that tool.

4.1.4 ASTs

Some ideas around ASTs

- AST Transformation Hooks for Domain Specific Languages
- possible meta coding in Python
- Reddit on pydit
- Reddit on ASTs (in /r/python)
- NuPIC implements cortical algorithms

4.1.5 Higher level structures

If the text to be editted is a program then parsing can be extended to include information on symbol tables, scope, control flow, tests, patterns, clichés, ...

If the text to be editted has meaning then parsing can be extended to include context, connections, importance, definition, relevance, ...

4.2 Rendering code

In order to be more easily read AST trees can pretty printed (a.k.a. rendered). Some other code which may be helpful in rendering AST trees as Python text are

4.2.1 Python's Unparser

The source for python includes an unparser, which can render ASTs to source.

Links to the unparser

- The unparser is a script in the Tools/parser directory called unparse.py.
- This script is a continuation from the unparse.py file in the older (2.7) Demo/parser directory, and has been intermittently maintained for a few years
- There is a test script in the same directory, and an equivalent test script for the 2.7 branch
- There are some issues about it, such as this one claiming that it is out of date.

Rendering

The code provides an Unparser class, which contains a method for each type of node. The methods render their own node's code, and call sub-nodes's methods sending in the respective nodes.

Running the unpraser on itself

Ideally the unparser should produce itself as output, given itself as input. Let's see how it does.

Running

Run the unparser, with the commands:

```
$ python2.7 unparse.py unparse.py > unparse.out.py
$ diff unparse.py unparse.out.py
```

Some differences are apparent

1. **Comments are stripped** This is expected. To my mind the problem here is that unparse.py includes any comments (but that may be a less common opinion).
2. **Docstrings are changed from double-quotes to single quotes.** Some docstrings in unparse.py use single double-quotes, others use the (PEP 8 recommendation of) triple double-quotes. In the output both have been reduced to single-quotes

   This feature is also apparent in other strings.
3. Blank lines are not maintained
4. **Extraneous parentheses are introduced** Some of these seem improvident, such as those introduced around conditions of (some) if statements. Others seem harmless, or even helpful, such as introducing parentheses around a literal tuple. Introducing parentheses around expressions after a print statement is dubious, given that I used python 2, not 3.
5. **Extra indentation is introduced at control structures** In particular this is noticeable at if/else statements, where the original uses compound statements for both if and else, but the unparsed version splits them over two lines. Once again the unparsed version seems more correct.
6. Re-formatting of a multi-line dictionary to a single line
7. Re-formatting expressions to introduce spaces around operators

Most of the above issues seem to be matters of opinion and should be deferred to PEP 8. The unparser's output does seem closer to PEP8 than is its code.

Pepping

To check that the output can be run through the pep8 tool. I've already noticed that blank lines and spaces around some operators aren't handled optimally, so I'll ask pep8 to ignore errors 302 and 203:

```
$ pep8 --ignore=E302,E203 unparse.out.py
unparse.out.py:22:80: E501 line too long (150 > 79 characters)
unparse.out.py:25:80: E501 line too long (95 > 79 characters)
unparse.out.py:255:80: E501 line too long (86 > 79 characters)
unparse.out.py:383:80: E501 line too long (82 > 79 characters)
unparse.out.py:408:80: E501 line too long (180 > 79 characters)
unparse.out.py:416:80: E501 line too long (150 > 79 characters)
unparse.out.py:573:23: W292 no newline at end of file
```

This shows up two more problems

1. Some lines are too long
2. The final line is not correctly terminated

Conclusion

This is not a test of the unparser, just an indication of where some of its strengths/weaknesses may lie. It did not show up any fatal flaws, but does suggest the unparser needs some (small) work to get it's own code to meet PEP 8 standards.

This quick run through also shows up the fact that "what code should look like" is a very opinionated subject. Python is lucky to have PEP 8 as a standard to refer to, but even that allows for variations (only "new code" need use spaces, for example). Which shows up one further bug in the unparser: all its formatting choices are hard-coded, whereas we may need greater flexibility in such choices.

4.2.2 Malthe Borch's sourcecodegen

sourcecodegen is a project by Malthe Borch which can generate source code from ASTs.

Links to sourcecodegen

- The source codegen project is stored on github
- It is available in the Cheeseshop

Summary

sourcecodegen is designed to work with older versions of Python, specifically "2.4 and below", and is not compatible with the new *ast* module in Python 2.6.

It is released under the BSD license, so can be copied for investigation.

Rendering

The code provides an ASTVisitor class, which contains a method for each type of node. The methods render their own node's code, and visit sub-nodes's methods sending in the respective nodes.

This concept of a visitor class which visits each node of a tree is a classic way of handling ASTs.

Running sourcecodegen on itself

Ideally the unparser should produce itself as output, given itself as input. Let's see how it does.

Running

Run the unparser, with the commands:

```
$ python2.5
>>> from compiler import parse
>>> tree = parse(file('visitor.py').read())
>>> from sourcecodegen import ModuleSourceCodeGenerator
>>> generator = ModuleSourceCodeGenerator(tree)
>>> visitor_out = file('visitor.py', 'w')
>>> print >> visitor_out, generator.getSourceCode()
>>> visitor_out.close()
>>> exit()
$ git commit "run sourcecodegen on itself" visitor.py
```

Some differences are apparent in the commit.

1. **Blank lines are not maintained** In at least one case this has resulted in incorrect code, with two lines being incorrectly joined.
2. Indentation is changed from spaces to tabs
3. **Extraneous parentheses are introduced** Some of these seem improvident, such as those introduced around conditions of (some) if statements. Some return statements also get extra parentheses, but not others. Some introduced parentheses seem harmless, or even helpful, such as introduing parentheses around a tuple.
4. Re-formatting of multi-line arguments for method calls to single lines
5. Re-formatting of strings to remove redundant escaping.

Most of the above issues seem to be matters of opinion and should be deferred to PEP 8. The unparser's output does seem closer to PEP8 than is its code.

Pepping

To check that the output can be run through the pep8 tool. I've already noticed that blank lines are removed, so we can ignore errors E301 and E302. And ignore W191 because we know indentation uses tabs:

```
$ pep8 --ignore=W191,E301,E302 visitor.py
visitor.py:8:80: E501 line too long (114 > 79 characters)
visitor.py:84:26: E712 comparison to False should be 'if cond is False:' or 'if not␣
↪cond:'
visitor.py:108:80: E501 line too long (84 > 79 characters)
visitor.py:469:80: E501 line too long (85 > 79 characters)
```

(continues on next page)

(continued from previous page)

```
visitor.py:479:80: E501 line too long (98 > 79 characters)
visitor.py:480:3: E112 expected an indented block
visitor.py:529:1: W391 blank line at end of file
```

This shows up some more problems

1. Some lines are too long
2. The final line is not correctly terminated
3. The code uses " == False", but that is not introduced by sourcecodegen itself, as it was in the original.
4. Some lines are joined, in particular pep8 notices a problem resulting from the joining of the lines before line 480.

Conclusion

This is not a test of sourcecodegen, just an indication of where some of its strengths/weaknesses may lie. It has one flaw (badly joined lines) which can result in unusable code, but otherwise seems robust

4.2.3 Eduardo Schettino's pyregurgitator

pyregurgitator is a project by Eduardo Schettino which provides tools for analysing Python code.

Links to pyregurgitator

- The pyRegurgitator project is stored on bitbucket
- It is available in the Cheeseshop

The project page does show a TODO list, but has had no commits in a few years.

Summary

pyregurgitator is intended to offer two primary tools

- asdl2html which provides a reference table of python's ASDL
- ast2html which provides a super pretty print of python source-code's AST

As of version 0.1 I have found three tools available

- ast2html which provides a super pretty print of python source-code's AST
- ast2txt which dumps out the AST, similarly to ast.dump
- ast2map which prints out the AST as a list of items, with minimal information about each

It is released under the MIT license, so can be copied for investigation.

Running pyregurgitator on itself

Run the ast2html tool, with the commands:

```
$ python2.5 bin/ast2html bin/ast2html > bin/ast2html.html
```

This produces output showing the detail as the AST of the original source

Conclusion

pyregurgitator is a helpful tool for anyone working with ASTs (such as myself). But it has proved not to be directly relevant to pym, as it concentrates on printing the AST, not the python. The methods it uses to pretty print the AST could yet prove informative, but details deserve a bit more investigation. Looks like he's using a dictionary.

4.2.4 Chris Warburton's Python-Decompiler

Python-Decompiler is a project by Chris Warburton which can generate source code from ASTs.

Links to Python-Decompiler

- The most recent form of the project is stored on gitorious.
- A previous version was stored on Github (whence I forked it while working with PyMeta).

Summary

The Python-Decompiler project includes a directory called python_rewriter which defines an OMeta grammar to parse Python's ASTs and thence render Python code. it is based on the ASTs of the 2.x compiler module

It is released into the Public Domain, so can be copied for investigation.

Rendering

Running Python-Decompiler on itself

Running

Run the unparser, with the commands:

```
$ python
>>> from python_rewriter import base
>>> source_text = file('base.py').read()
>>> parsed_source = base.parse(source_text)
>>> parsed_sources = [parsed_source]
>>> grammared_sources = base.grammar(parsed_sources)
>>> pythoned_sources = grammared_sources.apply('python', 0)
>>> text = pythoned_sources[0]
>>> file('base-out.py','w').write(text)
```

Some differences are apparent

1. **Comments are stripped** This is expected.
2. **Leading tabs are converted to spaces.** This is unsurprising, but not in the spirit of PEP 8's distinction about "new code"
3. Some extra blank lines are added

4. **Extraneous parentheses are introduced** Some of these seem improvident, such as those introduced around conditions of (some) if statements. Others seem harmless, or even helpful, such as introducing parentheses around a literal tuple. Introducing parentheses around expressions after a print statement is dubious, given that I used python 2, not 3.
5. Re-formatting expressions to introduce spaces around operators
6. **Reduction of a multi-line, triple-quoted string to a single-quoted string containing 'n' characters.** This is unfortunate as this string is a few hundred lines long and contains the OMeta grammar. The reduction renders the grammar effectively unreadable.

Most of the above issues seem to be matters of opinion and should be deferred to PEP 8. This unparser's effects seem very similar to *those of Python's unparser*.

Pepping

Conclusion

This is not a test of Python-Decompiler, just an indication of where some of its strengths/weaknesses may lie.

4.2.5 Alex Michael's approach

approach is a project by Alex Michael as a "toy" example of an HTML template engine.

Links to approach

- The approach project is explained at Alex's website only.

Summary

approach is more a teaching tool than intended for production use. As such it is a good exemplar for the many Python tools which render HTML from templates, which include some embedded Python.

> It parses templates, which are html with included python snippets, to produce an AST, and then renders the AST to HTML. The AST is represented by an object per node, with the objects being instantiations of Node subclass. Each subclass has its own version of a `render()` method, to which is provided a context. Such a class can also include supporting code to help with the rendering of the HTML, and so can be more than a simple rendering.

It is released without any license, so cannot be copied.

4.2.6 Jorge Monforte's heracles

heracles is a project by Jorge Monforte which can parse and render configuration files. It is based on augeas.

Links to heracles

- heracles is documented on Read The Docs.
- The source of the project is stored on github.
- It is available in the Cheeseshop.

The code is licensed under the LGPL, and so can be copied for investigation

Summary

heracles is designed to work with Python2, having been specifically tested with Python2.6 and 2.7.

Like augeas, heracles uses "lenses" which allow parsing and rendering of a variety of common Linux configuration files. Augeas is a c-based library which also provides Python bindings, whereas heracles goes further in allowing straight access to the parser functions from Python

The idea of a "lens parser" is derived from Boomerang, and is a "well-behaved bidirectional transformation" beween text and tree, which allows re-writing parts of a file while leaving the rest of the file untouched. As such a lens parser normally has two siginificant methods: get and put.

Rendering

The code provides an

Running

Run the unparser, with the commands:

```
$ python2.6
>>> from heracles import Heracles
```

Conclusion

4.2.7 Cenk Alti's pyhtml

pyhtml is a project by Cenk Alti which can render html from python sequences. Such sequences can look very similar to the generated html. Which is far more easily seen in his example, than in my explanation.

Links to pyhtml

- The source of the project is stored on github.
- It is available in the Cheeseshop.

The code is licensed under the Apache License, Version 2.0, and so can be copied for investigation.

Summary

pyhtml has been tested by Cenk with Python 2.7 only.

Rendering

Running

I have run pythtml's tests with Pythons 2.5, 2.6 and 2.7. It passed with the latter only, but fails were only due to use of assertIs and assertIn, both of which were introduced to the unittest module in 2.7. Replacing them with the older assertTrue led to the test suite passing in all three Python versions.

Conclusion

4.2.8 Python renderers of Python ASTs

- *Python's Unparser*
- *Malthe Borch's sourcecodegen*

4.2.9 Python renderers of other trees

- *Eduardo Schettino's pyregurgitator*
- *Alex Michael's approach*
- *Jorge Monforte's heracles*
- Guillaume Bour's Reblok builds ASTs from bytecode
- Cenk Alti's pythml renders html from method calls

4.2.10 Other renderers of other trees

- Augeas

4.2.11 To Do

A significant initial task shall be to evaluate these renderers:

- find their commonalities
- find their idiosyncracies
- decide on my own rendering strategy.

4.3 Miscellaneous

This page is mainly a grab-bag of interesting links which have given me some of the ideas behind pym.

4.3.1 Other stuff I have read

- In Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing (pdf) Tillmann Rendel & Klaus Ostermann propose a method for writing parsers such that pretty printers can be written in the same manner. I do not have enough knowledge of their academic context to appreciate it; but the idea of "invertible syntax" descriptions strikes a chord. It does seem reasonable that if one can parse from "A" into "B", then one should be able to render from "B" to "A".
- LL and LR in Context: Why Parsing Tools Are Hard introduced me to the notion of undecidably ambiguous grammars and the common strategies for getting around them.
- Martin Fowler distinguishes between Concrete and Abstract syntax trees when discussing language workbenches.
- Laurence Tratt rememebers that SDEs are Syntax directed editing tools and that when seasoned programmers were asked to use SDE tools, they revolted against the huge decline in their productivity, but that does not discouarge him from introducing Eco, an editor for language composition. (I liked /u/chrisdoner's contribution to the reddit discussion).

4.3.2 Other stuff I need to read

- C parser and AST generator written in Python: pycparser.
- PyPlus.
- CodeTalker.
- Reddit always has much to say, so /r/parsing will too.
- Python uses Zephyr Abstract Syntax Description Language to specify ASTs.
- parsley_page uses TermL to specify ASTs.
- Python uses SPARK, which is a Scanning, Parsing, and Rewriting Kit.
- There is a version of Ometa in Javascript.
- Static Modification of Python With Python: The AST Module.
- Sounds like fun: Coding in a debugger.
- Ira Baxter often turns up on forums I read, usually claiming that this is going to be harder than I think. He may have a point! (Alternatively, as an actress asked: Have you ever noticed how "What the hell!" is always the right decision to make?)
- Recognition can be harder than parsing.
- KOD has an interesting approach to text syntax trees (TST).
- The Program Transformation Wiki.
- The AST Tool Box looks like it is under active development

4.3.3 Related tools

- Autopep8 is a tool which re-writes Python source code (to increase compliance with PEP8), but does not use ASTs to do so.
- py.code provides higher level python code and introspection objects.
- Lamdu aims to "create a next-generation live programming environment". Their ambitions are wider than mine, but we start from similar problems.

Version 2.0 +

- ACE was A Cliche-based Program Structure Editor (pdf). Clichés are higher branches of the (AS) tree, or collections of branches, and are closer to what editors are actually thinking about. Ultimately one would prefer to "replace the switch with inheritance", not twiddle about with nodes and branches.
- One should be prepared to move beyond the standard flat represenation of text, to 3D Treemaps (pdf), especially Interactive Rendering of 3D Treemaps (pdf).
- (Like many others) I think Bret Victor is right about Learnable Programming.
- genindex
- search