Pymacs version 0.25

Extending Emacs with Python

Author: François Pinard

Email: pinard@iro.umontreal.ca

Contents

1 Introduction **1**
- 1.1 What is Pymacs? . . . 1
- 1.2 Documentation and examples . . . 1
- 1.3 Other resources . . . 3

2 Installation **3**
- 2.1 Check the search paths . . . 3
- 2.2 Edit the configuration file . . . 4
- 2.3 Check if Pymacs would work . . . 4
- 2.4 Install the Pymacs proper . . . 5
- 2.5 Prepare your .emacs file . . . 6
- 2.6 Porting and caveats . . . 6

3 Emacs Lisp structures and Python objects **7**
- 3.1 Conversions . . . 7
- 3.2 Simple objects . . . 7
- 3.3 Sequences . . . 8
- 3.4 Opaque objects . . . 9
  - 3.4.1 Emacs Lisp handles . . . 9
  - 3.4.2 Python handles . . . 10

4 Usage on the Emacs Lisp side **10**
- 4.1 Special Emacs Lisp functions . . . 10
  - 4.1.1 pymacs-exec . . . 11
  - 4.1.2 pymacs-eval . . . 11
  - 4.1.3 pymacs-call . . . 11
  - 4.1.4 pymacs-apply . . . 11
  - 4.1.5 pymacs-load . . . 11
  - 4.1.6 pymacs-autoload . . . 12
- 4.2 Special Emacs Lisp variables . . . 13
  - 4.2.1 pymacs-python-command . . . 13

4.2.2 pymacs-load-path . . . 13
4.2.3 pymacs-trace-transit . . . 13
4.2.4 pymacs-forget-mutability . . . 14
4.2.5 pymacs-mutable-strings . . . 14
4.2.6 Timeout variables . . . 14
4.2.7 pymacs-auto-restart . . . 15
4.2.8 pymacs-dreadful-zombies . . . 15

5 Usage on the Python side **16**
5.1 Python setup . . . 16
5.2 Emacs Lisp symbols . . . 16
5.3 Dynamic bindings . . . 17
5.4 Raw Emacs Lisp expressions . . . 18
5.5 User interaction . . . 18
5.6 Key bindings . . . 19

6 Debugging **20**
6.1 The communication protocol . . . 20
6.2 The *Pymacs* buffer . . . 21
6.3 Debugging the Pymacs helper . . . 22
6.4 Emacs usual debugging . . . 23
6.5 Python usual debugging . . . 23
6.6 Auto-reloading on save . . . 23

7 Administrative miscellany **24**
7.1 Development history . . . 24
7.2 Should it come with Emacs? . . . 25
7.3 The future of Pymacs . . . 25

8 Technical miscellany **25**
8.1 Known bugs or limitations . . . 25
8.1.1 Needed control on stack unwinding . . . 26
8.1.2 Possible memory leak . . . 26
8.1.3 Death from a Ctrl-C . . . 26
8.2 Suggestions to ponder . . . 26
8.2.1 Python-driven Pymacs . . . 26
8.2.2 Autoloading interface . . . 27
8.2.3 Handling more special forms . . . 27
8.2.4 Support for Python dictionaries . . . 28
8.2.5 A nicer *Pymacs* buffer . . . 29
8.3 Speed issues . . . 29
8.4 Vim-related thoughts . . . 30
There exists a Romanian translation of this manual.

1 Introduction

1.1 What is Pymacs?

Pymacs is a powerful tool which, once started from Emacs, allows two-way communication between Emacs Lisp and Python. Pymacs aims to employ Python as an ex-

tension language for Emacs rather than the other way around, and this asymmetry is reflected in some design choices. Within Emacs Lisp code, one may load and use Python modules. Python functions may themselves use Emacs services, and handle Emacs Lisp objects kept in Emacs Lisp space.

The goals are to write *naturally* in both languages, debug with ease, fall back gracefully on errors, and allow full cross-recursion.

It is very easy to install Pymacs, as neither Emacs nor Python need to be compiled nor relinked. Emacs merely starts Python as a subprocess, and Pymacs implements a communication protocol between both processes.

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

- mailto:pinard@iro.umontreal.ca

1.2 Documentation and examples

The main Pymacs site conveys the Pymacs documentation (you are reading its Pymacs manual right now) and distributions:

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

I expect average Pymacs users to have a deeper knowledge of Python than Emacs Lisp. People have widely varying approaches in writing `.emacs` files, as far as Pymacs is concerned:

- Some can go and write almost no Emacs Lisp, yet a bit is still necessary for establishing a few loading hooks. For many simple needs, one can do a lot without having to learn much.
- On the other hand, for more sophisticated usages, people cannot really escape knowing the Emacs Lisp API to some extent, because they should be familiar, programming-wise, with what is a buffer, a point, a mark, etc. and what are the allowed operations on those.

While Pymacs examples are no substitute for a careful reading of the Pymacs manual, the contemplation and study of others' nice works may well enligthen and deepen your understanding. A few examples are included within the Pymacs distribution, each as a subdirectory of the `contrib/` directory, and each having its own `README` file. These are listed below, easiest examples first:

- Paul Winkler's example
  - http://pymacs.progiciels-bpi.ca/Winkler.html
- Fernando Pérez' examples
  - http://pymacs.progiciels-bpi.ca/Perez.html
  - http://pymacs.progiciels-bpi.ca/contrib/Perez/
- Giovanni Giorgi's files
  - http://pymacs.progiciels-bpi.ca/Giorgi.html
  - http://pymacs.progiciels-bpi.ca/contrib/Giorgi/
- A reformatter for boxed comments
  - http://pymacs.progiciels-bpi.ca/rebox.html
  - http://pymacs.progiciels-bpi.ca/contrib/rebox/

A few more substantial examples of Pymacs usage have been brought to my attention, and are available externally (listed here in no particular order):

- pymdev — A Python Emacs Development Module:
  - http://www.toolness.com/pymdev/
- Ropemacs — Features like refactoring and code-assists:
  - http://rope.sf.net/ropemacs.html
  - http://rope.sf.net/hg/ropemacs
- Bicycle Repair Man — A Refactoring Tool for Python:
  - http://bicyclerepair.sourceforge.net/
- Emacs Freex — A personal wiki on steroids:
  - http://www.princeton.edu/%7Egdetre/software/freex/docs/index.html
- PyJde — Java dev source code browsing features in Emacs using Python:
  - http://code.google.com/p/pyjde/

The QaTeX project was influenced by Pymacs, according to its author:

- http://qatex.sourceforge.net/
- http://www.pytex.org/doc/eurotex2005.pdf

1.3 Other resources

You are welcome writing to or joining the following mailing list, where there are a few people around likely to give you feedback:

- mailto:pymacs-devel@googlegroups.com
- https://groups.google.com/group/pymacs-devel/

If you have no fear of wider crowds :-), there still is:

- mailto:python-list@python.org

There are other Web sites specifically about Pymacs. Giovanni Giorgi has one of them:

- http://blog.objectsroot.com/projects/pymacs/

There is an entry for Pymacs on Freshmeat:

- http://freshmeat.net/projects/pymacs/

2 Installation

2.1 Check the search paths

You should make sure that both Emacs and Python are usable, whatever the directory happens to be the current one. This is particularly important at the time Emacs launches Python under the scene, as Python ought to be found then started. On most systems, this means setting the search path correctly.

The following notes, for MS Windows, have been provided by Greg Detre.

- After `Start / Run / Cmd`, type `python`. If this works wherever you are, then your Python installation directory is already in your system's **PATH** environment variable. If that's not the case, follow the instructions here to add it:

  http://www.computerhope.com/issues/ch000549.htm

- You may have to add the directory containing the Python scripts that you want to run through Pymacs to your **PYTHONPATH** variable, in the same fashion as above. You can test this by running Python, and then:

  ```
  import sys
  sys.path
  ```

  or just:

  ```
  import my_python_scripts
  ```

  from somewhere besides your scripts directory.

2.2 Edit the configuration file

In most cases, you may safely skip this step, as it is only needed in unusual, problematic circumstances. Merely check that none of the following applies to you.

- Under Aquamacs (which is a MacOS X native port of Emacs), it has been reported that one gets *Lisp nesting exceeds max-lisp-eval-depth* messages while interactively requesting the documentation for Lisp functions (we do not know why). If you have this problem, edit file `ppppconfig.py`, locate the line defining **DEFADVICE_OK**, make sure it gets the string `'nil'` as a value, instead of the string `'t'`, then save the edited file before proceeding further. This should work around the problem. The price to pay is that you will not get the Python docstring for modules imported through Pymacs.

2.3 Check if Pymacs would work

To know, before installing Pymacs, if it would work on your system, try the validation suite by running `make check`. The suite is fairly elementary, but nevertheless, it is able to detect some common show stoppers. To check a particular Emacs and Python combination, use `make check EMACS=some_Emacs PYTHON=some_Python`.

If `PYTHON` is left unset or empty, then the command for starting the Pymacs helper is `python`. Otherwise, it may be set to give the full path of the Python executable if it exists at some location outside the program search path. It may also be given when the interpreter name is different, for exemple when the Python version is part of the program name.

If `EMACS` is left unset or empty, then the command for starting the Emacs editor is `emacs`. For normal Pymacs usage, Emacs is launched by the user long before Pymacs is itself started, and consequently, there is absolutely no need to tell Pymacs which Emacs is needed. For the validation suite however, it may be set to give the full path of the executable if the Emacs program exists at some location outside the program search path. It may also be given when the editor name is different, for example when the Emacs version is part of the program name, or when this is a different editor. For

example, `make check EMACS=xemacs` runs the validation suite using `xemacs` for an editor.

The remaining of this section may be safely be skipped for mere Pymacs installation.

I did not base the validation suite on Junit (the Python unit testing framework is a re-implementation of it), but on Codespeak's pylib `py.test`, which is much simpler, and still very powerful. The **pylib** project is driven by Holge Kregel, but attracted some Python brains, like Armin Rigo (known for Psyco, among other things -- I think his **lsprof** has also been added to Python 2.5 under the name **cProfile**). This gang addresses overdone/heavy methods in Python, and do them better. Even `py.test` is a bit more complex that I would want, and has (or at least had) flaws on the Unicode side, so I rewrote my own, as a simple single file. I merely translated it from French to English, to make it more distributable within Pymacs.

I initially tried using Emacs stdin and stdout for communicating expressions to evaluate and getting back results, from within the validation suite. This did not prove useful so, so after some fight, I reluctantly put this avenue aside. Currently, the suite writes problems in files, for Emacs to read, and Emacs writes replies in files, for the suite to check. Busy waiting (with small sleep added in the loops) is used on both sides. This is all too heavy, and it slows down the suite. Hopefully, the suite is not run often, this is not a real problem.

2.4 Install the Pymacs proper

Pymacs is lean. Putting the documentation and administrative files aside, there is one Python file and one Emacs Lisp file to it, to be installed in turn. Always start with the Python file.

- For the Python part

  From the top-level of the Pymacs distribution, execute `make install`. If you do not have a Make program (Microsoft Windows?) read the `Makefile` file and emulate what `make install` does, maybe something like this:

  ```
  python pppp -C ppppconfig.py pppp.rst.in pymacs.el.in \
    pymacs.rst.in Pymacs.py.in contrib tests
  python setup.py install
  ```

  Without `make install`, you might also have to combine the two first lines above into a single longer one, without the backslash.

  If the Python interpreter has a non-standard name or location, rather do `make install PYTHON=Some_Python` (see the previous section for a discussion). First, the script copies a few source files while configuring them: it presets the version string and the name of the Python interpreter, it also adapts the Python source code which might differ, for example, between Python 2 and Python 3. Second, it installs the Python file through the Python standard Distutils tool. To get an option reminder, do `python setup.py install --help`. Consult the Distutils documentation if you need more information about this.

  That's normally all to it. To check that `Pymacs.py` is properly installed, start an interactive Python session and type `from Pymacs import lisp`: you should not receive any error.

A special difficulty arises when the particular Python you use does not have Distutils already installed. In such a case, `make install` prints a warning, leaving to you the task of figuring out where the `Pymacs/` directory is best copied, and making that copy.

- For the Emacs part

  This is usually done by hand now. First select some directory along the list kept in your Emacs **load-path**, for which you have write access, and copy file `pymacs.el` in that directory.

  If you want speed, you should ideally byte-compile this file. To do so, go to that directory, launch Emacs, then give the command `M-x byte-compile-file RET pymacs.el RET`. If for some reason you intend to such commands often, you could create a little script to do so. Here is an example of such a script, assuming here that you use Emacs and want to install in directory `~/share/emacs/lisp/`:

  ```
  #!/bin/bash
  cp pymacs.el ~/share/emacs/lisp/
  emacs -batch -eval '(byte-compile-file "~/share/emacs/lisp/pymacs.el")'
  ```

  You should be done now. To check that `pymacs.el` is properly installed, return to your usual directories, start Emacs and give it the command `M-x load-library RET pymacs RET`: you should not receive any error.

Some features from previous Pymacs releases have been dropped:

- Environment variable `PYMACS_EMACS` is gone, and environment variable `PYMACS_PYTHON` is usually not needed.
- There used to be a script for installing the Emacs Lisp file. As it was difficult to get it right in all circumstances; the script grew an interactive mode and lot of options. This is just not worth the complexity, so this script is now gone.
- Examples were all installed automatically, but at least for some of them, this was more pollution than help. You may browse the contents of the `contrib/` directory to learn about available examples.

2.5 Prepare your `.emacs` file

The `.emacs` file is not given in the distribution, you likely have one already in your home directory. You need to add these lines:

```
(autoload 'pymacs-apply "pymacs")
(autoload 'pymacs-call "pymacs")
(autoload 'pymacs-eval "pymacs" nil t)
(autoload 'pymacs-exec "pymacs" nil t)
(autoload 'pymacs-load "pymacs" nil t)
(autoload 'pymacs-autoload "pymacs")
;;(eval-after-load "pymacs"
;;  '(add-to-list 'pymacs-load-path YOUR-PYMACS-DIRECTORY"))
```

If you plan to use a special directory to hold your own Pymacs code in Python, which should be searched prior to the usual Python import search path, then uncomment the last two lines (by removing the semi-colons) and replace *YOUR-PYMACS-DIRECTORY* by the name of your special directory. If the file `~/.emacs` does not exist, merely create it with the above lines. You are now all set to use Pymacs.

To check this, start a fresh Emacs session, and type `M-x pymacs-eval RET`. Emacs should prompt you for a Python expression. Try `repr(2L**111) RET` (rather use `repr(2**111) RET` if you are using Python 3). The mini buffer should display *"259614842926741381426524816461004 8L"* (yet there is no `L` suffix in Python 3).

Let's do a second test. Whether in the same Emacs session or not, `M-x pymacs-load RET` should prompt you for a Python module name. Reply `os RET RET` (the second `RET` is for accepting the default prefix). This should have the effect of importing the Python **os** module within Emacs. Typing `M-: (os-getcwd) RET` should echo the current directory in the message buffer, as returned by the **os.getcwd** Python function.

2.6 Porting and caveats

Pymacs has been initially developed on Linux, Python 1.5.2, and Emacs 20, and is currently developed using Python 2.6, Python 3.1, Emacs 23.1 and XEmacs 21.4. It is expected to work out of the box on many flavours of Unix, MS Windows and Mac OSX, and also on many version of Python, Emacs and XEmacs.

From Pymacs 0.23 and upwards, Python 2.2 or better is likely needed, and for the Pymacs proper, I rely on testers or users for portability issues. However, the validation suite itself requires Python 2.6 or better, someone might choose to contribute the back porting. Python 3.1 support has been added for Pymacs 0.24.

Pymacs uses Emacs weak hash tables. It can run without them, but then, complex Python objects transmitted to Emacs will tie Python memory forever. It should not be a practical problem in most simple cases. Some later versions of Emacs 20 silently create ordinary tables when asked for weak hash tables. Older Emacses do not have hash tables.

In earlier versions, Pymacs was installing a `Pymacs` Python package holding a single `pymacs.py` file (besides the mandatory `__init__.py`). This is now replaced by a single `Pymacs.py` file, and because of the capitalisation, the API did not need to change.

3 Emacs Lisp structures and Python objects

3.1 Conversions

Whenever Emacs Lisp calls Python functions giving them arguments, these arguments are Emacs Lisp structures that should be converted into Python objects in some way. Conversely, whenever Python calls Emacs Lisp functions, the arguments are Python objects that should be received as Emacs Lisp structures. We need some conventions for doing such conversions.

Conversions generally transmit mutable Emacs Lisp structures as mutable objects on the Python side, in such a way that transforming the object in Python will effectively transform the structure on the Emacs Lisp side (strings are handled a bit specially

however, see below). The other way around, Python objects transmitted to Emacs Lisp often loose their mutability, so transforming the Emacs Lisp structure is not reflected on the Python side.

Pymacs sticks to standard Emacs Lisp, it explicitly avoids various Emacs Lisp extensions. One goal for many Pymacs users is taking some distance from Emacs Lisp, so Pymacs is not overly pushing users deeper into it.

3.2 Simple objects

Emacs Lisp **nil** and the equivalent Emacs Lisp `()` yield Python **None**. Python **None**, Python **False** and the Python empty list `[]` are returned as **nil** in Emacs Lisp. Notice the assymetry, in that three different Python objects are mapped into a single Emacs Lisp object. So, neither **False** nor `[]` are likely produced by automatic conversions from Emacs Lisp to Python.

Emacs Lisp **t** yields Python **True**. Python **True** is returned as **t** in Emacs Lisp.

Emacs Lisp numbers, either integer or floating, are converted in equivalent Python numbers. Emacs Lisp characters are really numbers and yield Python numbers. In the other direction, Python numbers are converted into Emacs Lisp numbers, with the exception of long Python integers and complex numbers.

Emacs Lisp strings are usually converted into equivalent Python strings. As Python strings do not have text properties, these are not reflected. This may be changed by setting the **pymacs-mutable-strings** option: if this variable is not **nil**, Emacs Lisp strings are then transmitted opaquely. Python strings are always converted into Emacs Lisp strings. Python releases before version 3 make a distinction between Unicode and narrow strings: Unicode strings are then produced on the Python side for Emacs Lisp multi-byte strings, but only when they do not fit in ASCII, otherwise Python narrow strings are produced. Conversely, Emacs Lisp multi-byte strings are produced for Python strings, but only when they do not fit ASCII, otherwise Emacs Lisp uni-byte strings are produced. Currently, Pymacs behaviour is undefined for users wandering outside the limits of Emacs' **utf-8** coding system.

Emacs Lisp symbols yield `lisp[STRING]` notations on the Python side, where *STRING* names the symbol. In the other direction, Python `lisp[STRING]` corresponds to an Emacs Lisp symbol printed with that *STRING* which, of course, should then be a valid Emacs Lisp symbol name. As a convenience, `lisp.SYMBOL` on the Python side yields an Emacs Lisp symbol with underscores replaced with hyphens; this convention is welcome, as Emacs Lisp programmers commonly prefer using dashes, where Python programmers use underlines. Of course, this `lisp.SYMBOL` notation is only usable when the *SYMBOL* is a valid Python identifier, while not being a Python keyword.

3.3 Sequences

The case of strings has been discussed in the previous section.

Proper Emacs Lisp lists, those for which the **cdr** of last cell is **nil**, are normally transmitted opaquely to Python. If **pymacs-forget-mutability** is set, or if Python later asks for these to be expanded, proper Emacs Lisp lists get converted into Python lists, if we except the empty list, which is always converted as Python **None**. In the other direction, Python lists are always converted into proper Emacs Lisp lists.

Emacs Lisp vectors are normally transmitted opaquely to Python. However, if **pymacs-forget-mutability** is set, or if Python later asks for these to be expanded,

Emacs Lisp vectors get converted into Python tuples. In the other direction, Python tuples are always converted into Emacs Lisp vectors.

Remember the rule: *Round parentheses correspond to square brackets!*. It works for lists, vectors, tuples, seen from either Emacs Lisp or Python.

The above choices were debatable. Since Emacs Lisp proper lists and Python lists are the bread-and-butter of algorithms modifying structures, at least in my experience, I guess they are more naturally mapped into one another, this spares many casts in practice. While in Python, the most usual idiom for growing lists is appending to their end, the most usual idiom in Emacs Lisp to grow a list is by cons'ing new items at its beginning:

```
(setq accumulator (cons 'new-item accumulator))
```

or more simply:

```
(push 'new-item accumulator)
```

So, in case speed is especially important and many modifications happen in a row on the same side, while order of elements ought to be preserved, some `(nreverse ...)` on the Emacs Lisp side or `.reverse()` on the Python side might be needed. Surely, proper lists in Emacs Lisp and lists in Python are the normal structure for which length is easily modified.

We cannot so easily change the size of a vector, the same as it is a bit more of a stunt to *modify* a tuple. The shape of these objects is fixed. Mapping vectors to tuples, which is admittedly strange, will only be done if the Python side requests an expanded copy, otherwise an opaque Emacs Lisp object is seen in Python. In the other direction, whenever an Emacs Lisp vector is needed, one has to write `tuple(python_list)` while transmitting the object. Such transmissions are most probably to be unusual, as people are not going to blindly transmit whole big structures back and forth between Emacs and Python, they would rather do it once in a while only, and do only local modifications afterwards. The infrequent casting to **tuple** for getting an Emacs Lisp vector seems to suggest that we did a reasonable compromise.

In Python, both tuples and lists have O(1) access, so there is no real speed consideration there. Emacs Lisp is different: vectors have O(1) access while lists have O(N) access. The rigidity of Emacs Lisp vectors is such that people do not resort to vectors unless there is a speed issue, so in real Emacs Lisp practice, vectors are used rather parsimoniously. So much, in fact, that Emacs Lisp vectors are overloaded for what they are not meant: for example, very small vectors are used to represent X events in key-maps, programmers only want to test vectors for their type, or users just like bracketed syntax. The speed of access is hardly an issue then.

3.4 Opaque objects

3.4.1 Emacs Lisp handles

When a Python function is called from Emacs Lisp, the function arguments have already been converted to Python types from Emacs Lisp types and the function result is going to be converted back to Emacs Lisp.

Several Emacs Lisp objects do not have Python equivalents, like for Emacs windows, buffers, markers, overlays, etc. It is nevertheless useful to pass them to Python functions, hoping that these Python functions will *operate* on these Emacs Lisp objects.

Of course, the Python side may not itself modify such objects, it has to call for Emacs services to do so. Emacs Lisp handles are a mean to ease this communication.

Whenever an Emacs Lisp object may not be converted to a Python object, an Emacs Lisp handle is created and used instead. Whenever that Emacs Lisp handle is returned into Emacs Lisp from a Python function, or is used as an argument to an Emacs Lisp function from Python, the original Emacs Lisp object behind the Emacs Lisp handle is automatically retrieved.

Emacs Lisp handles are either instances of the internal **Lisp** class, or of one of its subclasses. If *OBJECT* is an Emacs Lisp handle, and if the underlying Emacs Lisp object is an Emacs Lisp sequence, then whenever `OBJECT[INDEX]`, `OBJECT[INDEX] = VALUE` and `len(OBJECT)` are meaningful, these may be used to fetch or alter an element of the sequence directly in Emacs Lisp space. Also, if *OBJECT* corresponds to an Emacs Lisp function, `OBJECT(ARGUMENTS)` may be used to apply the Emacs Lisp function over the given arguments. Since arguments have been evaluated the Python way on the Python side, it would be conceptual overkill evaluating them again the Emacs Lisp way on the Emacs Lisp side, so Pymacs manage to quote arguments for defeating Emacs Lisp evaluation. The same logic applies the other way around.

Emacs Lisp handles have a `value()` method, which merely returns self. They also have a `copy()` method, which tries to *open the box* if possible. Emacs Lisp proper lists are turned into Python lists, Emacs Lisp vectors are turned into Python tuples. Then, modifying the structure of the copy on the Python side has no effect on the Emacs Lisp side.

For Emacs Lisp handles, `str()` returns an Emacs Lisp representation of the handle which should be **eq** to the original object if read back and evaluated in Emacs Lisp. `repr()` returns a Python representation of the expanded Emacs Lisp object. If that Emacs Lisp object has an Emacs Lisp representation which Emacs Lisp could read back, then `repr()` value is such that it could be read back and evaluated in Python as well, this would result in another object which is **equal** to the original, but not necessarily **eq**.

3.4.2 Python handles

The same as Emacs Lisp handles are useful for handling Emacs Lisp objects on the Python side, Python handles are useful for handling Python objects on the Emacs Lisp side.

Many Python objects do not have direct Emacs Lisp equivalents, including long integers, complex numbers, modules, classes, instances and surely a lot of others. When such are being transmitted to the Emacs Lisp side, Pymacs use Python handles. These are automatically recovered into the original Python objects whenever transmitted back to Python, either as arguments to a Python function, as the Python function itself, or as the return value of an Emacs Lisp function called from Python.

The objects represented by these Python handles may be inspected or modified using the basic library of Python functions. For example, in:

```
(pymacs-exec "import re")
(setq matcher (pymacs-eval "re.compile('PATTERN').match"))
(pymacs-call matcher ARGUMENT)
```

the **setq** line above could be decomposed into:

```
(setq compiled (pymacs-eval "re.compile('PATTERN')")
```

```
matcher (pymacs-call "getattr" compiled "match"))
```

This example shows that one may use **pymacs-call** with **getattr** as the function, to get a wanted attribute for a Python object.

4 Usage on the Emacs Lisp side

4.1 Special Emacs Lisp functions

Pymacs is mainly launched and used through a few special functions, among all those added by Pymacs for Emacs Lisp. These few imported functions are listed and detailed in the following subsections. They really are the preferred way to call Python services with Pymacs.

Even then, we do not expect that **pymacs-exec**, **pymacs-eval**, **pymacs-call** or **pymacs-apply** will be much used, if ever, in most Pymacs applications. In practice, the Emacs Lisp side of a Pymacs application might call either **pymacs-autoload** or **pymacs-load** a few times for linking into the Python modules, with the indirect effect of defining trampoline functions for these modules on the Emacs Lisp side, which can later be called like usual Emacs Lisp functions.

4.1.1 pymacs-exec

Function `(pymacs-exec TEXT)` gets *TEXT* executed as a Python statement, and its value is always **nil**. So, this function may only be useful because of its possible side effects on the Python side.

This function may also be called interactively:

```
M-x pymacs-exec RET TEXT RET
```

4.1.2 pymacs-eval

Function `(pymacs-eval TEXT)` gets *TEXT* evaluated as a Python expression, and returns the value of that expression converted back to Emacs Lisp.

This function may also be called interactively:

```
M-x pymacs-eval RET TEXT RET
```

4.1.3 pymacs-call

Function `(pymacs-call FUNCTION ARGUMENT...)` will get Python to apply the given *FUNCTION* over zero or more *ARGUMENT*. *FUNCTION* is either a string holding Python source code for a function (like a mere name, or even an expression), or else, a Python handle previously received from Python, and hopefully holding a callable Python object. Each *ARGUMENT* gets separately converted to Python before the function is called. **pymacs-call** returns the resulting value of the function call, converted back to Emacs Lisp.

4.1.4 pymacs-apply

Function `(pymacs-apply FUNCTION ARGUMENTS)` will get Python to apply the given *FUNCTION* over the given *ARGUMENTS*. *ARGUMENTS* is a list containing

all arguments, or **nil** if there is none. Besides arguments being bundled together instead of given separately, the function acts pretty much like **pymacs-call**.

4.1.5 pymacs-load

Function `(pymacs-load MODULE PREFIX)` imports the Python *MODULE* into Emacs Lisp space. *MODULE* is the name of the file containing the module, without any `.py` or `.pyc` extension. If the directory part is omitted in *MODULE*, the module will be looked into the current Python search path. Dot notation may be used when the module is part of a package. Each top-level function in the module produces a trampoline function in Emacs Lisp having the same name, except that underlines in Python names are turned into dashes in Emacs Lisp, and that *PREFIX* is uniformly added before the Emacs Lisp name (as a way to avoid name clashes). *PREFIX* may be omitted, in which case it defaults to base name of *MODULE* with underlines turned into dashes, and followed by a dash.

Note that **pymacs-load** has the effect of declaring the module variables and methods on the Emacs Lisp side, but it does *not* declare anything on the Python side. Of course, Python imports the module before making it available for Emacs, but there is no Pymacs ready variable on the Python side holding that module. If you need to import *MODULE* in a variable on the Python side, the proper incantation is `(pymacs-exec "import MODULE")`. And of course, this latter statement does not declare anything on the Emacs Lisp side.

Whenever **pymacs_load_hook** is defined in the loaded Python module, **pymacs-load** calls it without arguments, but before creating the Emacs view for that module. So, the **pymacs_load_hook** function may create new definitions or even add **interaction** attributes to functions.

The return value of a successful **pymacs-load** is the module object. An optional third argument, *noerror*, when given and not **nil**, will have **pymacs-load** to return **nil** instead of raising an error, if the Python module could not be found.

When later calling one of these trampoline functions, all provided arguments are converted to Python and transmitted, and the function return value is later converted back to Emacs Lisp. It is left to the Python side to check for argument consistency. However, for an interactive function, the interaction specification drives some checking on the Emacs Lisp side. Currently, there is no provision for collecting keyword arguments in Emacs Lisp.

This function may also be called interactively:

```
M-x pymacs-load RET MODULE RET PREFIX RET
```

If you find yourself using **pymacs-call** a lot for builtin Python functions, you might rather elect to import all Python builtin functions and definitions directly into Emacs Lisp space, and call them directly afterwards. Here is a recipe (use the first line for Python 2, or the second line for Python 3):

```
M-x pymacs-load RET __builtin__ RET py- RET
M-x pymacs-load RET builtins RET py- RET
```

After such a command, calling the function `py-getattr`, say, with an opaque Python object and with a string naming an attribute, returns the value of that attribute for that object.

4.1.6 pymacs-autoload

Function `(pymacs-autoload FUNCTION MODULE PREFIX DOCSTRING INTERACTIVE)` is meant to mimic the functionality of the standard Emacs **autoload** function.

It declares *FUNCTION* to be autoloaded from the specified Python *MODULE*. The **pymacs-load** for this module is delayed until *FUNCTION* is actually called. Of course, if there are many such functions declared as autoloading the module, calling any of them will then load the module and resolve the autoloading for all of them at once. For the meaning of the optional *PREFIX* argument, see the documentation for the **pymacs-load** function above.

Before the function gets loaded for real, Emacs may still provide a documentation for it, which the user gives through the contents of the optional *DOCSTRING*. Emacs also needs to know if the function may be called interactively and, when this is the case, the arguments it may accept. If the *INTERACTIVE* argument is not provided, or when it is nil, the function is not known to be interactive. A value of **t** for *INTERACTIVE* means that the function is interactive, but has no arguments. Otherwise, *INTERACTIVE* receives a description of the interaction to interactively get the function arguments. See the Emacs documentation for function **autoload** and **interactive** for more information.

If, at the moment of the **pymacs-autoload** call, *FUNCTION* is already related to a loaded Python function, the autoloading declaration is ignored.

Here are examples of usage for the **pymacs-autoload** function:

```
(pymacs-autoload 'os-getenv "os" nil nil "sEnv name: ")
(pymacs-autoload 'posix-getenv "os" "posix-" nil
    '(list (read-string "Env name: ")))
```

The second example could be written more simply as in the first example. Moreover, both examples of an :var:INTERACTIVE argument are merely given here for illustration, as the real **os-getenv** function is *not* interactive.

4.2 Special Emacs Lisp variables

Users could alter the inner working of Pymacs through a few variables, these are all documented here. Except for **pymacs-python-command** and **pymacs-load-path**, which should be set before calling any Pymacs function, the value of these variables can be changed at any time.

4.2.1 pymacs-python-command

This variable is initialized with the Python executable that was used at installation time. It tells Emacs about the Python interpreter to launch far starting the Pymacs helper. The value of this variable may be overridden by setting the `PYMACS_PYTHON` environment variable, yet in practice, for newer versions of Pymacs, this is rarely needed.

While the Python part of Pymacs is pre-processed and yields different sources for Python 2 and Python 3 (among other possibilities), the Emacs part of Pymacs is mostly configured at run time for various Emacs versions, so the same Emacs source is likely to work unaltered, would it be for different versions of Emacs and for different versions of Python. So it makes sense, at least in some special circumstances, giving the capability of selecting a specific Python interpreter by programmatical means within Emacs.

4.2.2 pymacs-load-path

Users might want to use special directories for holding their Python modules, when these modules are meant to be used from Emacs. Best is to preset **pymacs-load-path**, **nil** by default, to a list of these directory names. (Tilde expansions and such occur automatically.)

Here is how it works. The first time Pymacs is needed from Emacs, a Pymacs helper is automatically started as an Emacs subprocess, and given as arguments all strings in the **pymacs-load-path** list. These arguments are added at the beginning of **sys.path**, or moved at the beginning if they were already on **sys.path**. So in practice, nothing is removed from **sys.path**.

4.2.3 pymacs-trace-transit

The ***Pymacs*** buffer, within Emacs, holds a trace of transactions between Emacs and Python. When **pymacs-trace-transit** is **nil**, the buffer only holds the last bi-directional transaction (a request and a reply). In this case, it gets erased before each and every transaction. If that variable is **t**, all transactions are kept. This could be useful for debugging, but the drawback is that this buffer could grow big over time, to the point of diminishing Emacs performance. As a compromise, that variable may also be a cons cell of integers `(KEEP . LIMIT)`, in which case the buffer is reduced to approximately *KEEP* bytes whenever its size exceeds *LIMIT* bytes, by deleting an integral number of lines from its beginning. The default setting for **pymacs-trace-transit** is `(5000 . 30000)`.

4.2.4 pymacs-forget-mutability

The default behaviour of Pymacs is to transmit Emacs Lisp objects to Python in such a way that they are fully modifiable from the Python side, would it mean triggering Emacs Lisp functions to act on them. When **pymacs-forget-mutability** is not **nil**, the behaviour is changed, and the flexibility is lost. Pymacs then tries to expand proper lists and vectors as full copies when transmitting them on the Python side. This variable, seen as a user setting, is best left to **nil**. It may be temporarily overridden within some functions, when deemed useful.

There is no corresponding variable from objects transmitted to Emacs from Python. Pymacs automatically expands what gets transmitted. Mutability is preserved only as a side-effect of not having a natural Emacs Lisp representation for the Python object. This asymmetry is on purpose, yet debatable. Maybe Pymacs could have a variable telling that mutability is important for Python objects? That would give Pymacs users the capability of restoring the symmetry somewhat, yet so far, in our experience, this has never been needed.

4.2.5 pymacs-mutable-strings

Strictly speaking, Emacs Lisp strings are mutable. Yet, it does not come naturally to a Python programmer to modify a string *in-place*, as Python strings are never mutable. When **pymacs-mutable-strings** is **nil**, which is the default setting, Emacs Lisp strings are transmitted to Python as Python strings, and so, loose their mutability. Moreover, text properties are not reflected on the Python side. But if that variable is not **nil**, Emacs Lisp strings are rather passed as Emacs Lisp handles. This variable is ignored whenever **pymacs-forget-mutability** is set.

4.2.6 Timeout variables

Emacs needs to protect itself a bit, in case the Pymacs service program, which handles the Python side of requests, would not start correctly, or maybe later die unexpectedly. So, whenever Emacs reads data coming from that program, it sets a time limit, and take some action whenever that time limit expires. All times are expressed in seconds.

The **pymacs-timeout-at-start** variable defaults to 30 seconds, this time should only be increased if a given machine is so heavily loaded that the Pymacs service program has not enough of 30 seconds to start, in which case Pymacs refuses to work, with an appropriate message in the mini buffer.

The two remaining timeout variables almost never need to be changed in practice. When Emacs is expecting a reply from Python, it might repeatedly check the status of the Pymacs service program when that reply is not received fast enough, just to make sure that this program did not die. The **pymacs-timeout-at-reply** variable, which defaults to 5, says how many seconds to wait without checking, while expecting the first line of a reply. The **pymacs-timeout-at-line** variable, which defaults to 2, says how many seconds to wait without checking, while expecting a line of the reply after the first.

4.2.7 pymacs-auto-restart

The Pymacs helper process is started as soon as it is needed, and gets associated with the ***Pymacs*** buffer. When that buffer is killed, as it occurs automatically whenever the Emacs session is ending, the Pymacs helper process is killed as well. Any other disappearance of the helper is unexpected, and might be the consequence of some error in the Python side of the user application (or a Pymacs bug, maybe!).

When the Pymacs helper dies, all useful Python objects it might contain also die with it. So, after an unexpected death, there might now exist dangling references in Emacs Lisp space towards vanished Python objects, and using these references may be fatal to the application. When the Pymacs helper dies, the safest thing to do is stopping all Pymacs functionality and even exiting Emacs. On the other hand, it is not always practical having to restart everything in such cases: the user knows best, and is the one who ultimately decides.

The Pymacs helper death is detected at the time a new Pymacs request gets initiated from the Emacs side. Pymacs could not do much without a Pymacs helper, so it has either to restart a new Pymacs helper, or abort the Pymacs request. The variable **pymacs-auto-restart** controls how this is done. The possible values are:

- `nil` — the Pymacs request is unconditionally aborted,
- `t` — a new Pymacs helper is silently launched, and the previous helper death might well go unnoticed,
- `'ask` — the user interactively decides whether to restart the Pymacs helper or not. This is the default value.

4.2.8 pymacs-dreadful-zombies

When a Pymacs helper gets restarted in a given Emacs session, brand new Python objects may be created within that new helper. There is not enough information kept on the Emacs Lisp side for the new Pymacs helper to recreate the useful Python objects

which disappeared. However, there is enough machinery to recover all their slot numbers (all references to opaque Python objects from Emacs Lisp space are transmitted in form of object slot numbers).

The new Pymacs helper is given the list of all previous slot numbers still referenced from the Emacs side, and is then careful at never allocating a new Python object using an old slot number, as this might possibly create fatal confusion. All the previous slots are initialized with so-called *zombies* on the Python side. If Emacs later calls a vanished Python object, this merely awakes its zombie, which will then make some noise, then fall asleep again. The noise has the form of a diagnostic within the `*Messages*` buffer, sometimes visible in the mini-buffer too, at least when the mini-buffer is not simultaneously used for some other purpose.

Zombies get more dreadful if **pymacs-dreadful-zombies** is set to a non-**nil** value. In this case, calling a vanished Python object raises an error that will eventually interrupt the current computation. Such a behaviour might be useful for debugging purposes, or for making sure that no call to a vanished Python object goes unnoticed.

In previous Pymacs releases, zombies were always dreadful, under the assumption that calling a vanished object is a real error. However, it could cause irritation in some circumstances, like when associated with frequently triggered Emacs Lisp hook functions. That's why that, by default, zombies have been finally turned into more innocuous beings!

5 Usage on the Python side

5.1 Python setup

For Python modules meant to be used from Emacs and which receive nothing but Emacs **nil**, numbers or strings, or return nothing but Python **None**, numbers or strings, then Pymacs requires little or no setup. Otherwise, use `from Pymacs import lisp` at the start of your module. If you need more Pymacs features, like the **Let** class, then write `from Pymacs import lisp, Let`.

The Pymacs helper runs Python code to serve the Emacs side, and it is blocked waiting until Emacs sends a request. Until the Pymacs helper returns a reply, Emacs is blocked in turn, yet fully listening to serve eventual Python sub-requests, etc. So, either Emacs or the Pymacs helper is active at a given instant, but never both at once.

Unless Emacs has sent a request to the Pymacs helper and is expecting a reply, it is just not listening to receive Python requests. So, any other Python thread may not asynchronously use Pymacs to get Emacs services. The design of the Python application should be such that the communication is always be channelled from the main Python thread.

When Pymacs starts, all process signals are inhibited on the Python side. Yet, **SIGINT** gets re-enabled while running user functions. If the user elects to reactivate some other signal in her Python code, she should do so as to not damage or severe the communication protocol.

5.2 Emacs Lisp symbols

lisp is a special object which has useful built-in magic. Its attributes do nothing but represent Emacs Lisp symbols, created on the fly as needed (symbols also have their built-in magic).

As special cases, `lisp.nil` or `lisp["nil"]` are the same as **None**, and `lisp.t` or `lisp["t"]` are the same as **True**. Otherwise, both `lisp.SYMBOL` and `lisp[STRING]` yield objects of the internal **Symbol** type. These are genuine Python objects, that could be referred to by simple Python variables. One may write `quote = lisp.quote`, for example, and use `quote` afterwards to mean that Emacs Lisp symbol. If a Python function received an Emacs Lisp symbol as an argument, it can check with == if that argument is `lisp.never` or `lisp.ask`, say. A Python function may well choose to return some symbol, like `lisp.always`.

In Python, writing `lisp.SYMBOL = VALUE` or `lisp[STRING] = VALUE` does assign *VALUE* to the corresponding symbol in Emacs Lisp space. Beware that in such cases, the `lisp.` prefix may not be spared. After `result = lisp.result`, one cannot hope that a later `result = 3` will have any effect in the Emacs Lisp space: this would merely change the Python variable `result`, which was a reference to a **Symbol** instance, so it is now a reference to the number 3.

The **Symbol** class has `value()` and `copy()` methods. One can use either `lisp.SYMBOL.value()` or `lisp.SYMBOL.copy()` to access the Emacs Lisp value of a symbol, after conversion to some Python object, of course. However, if `value()` would have given an Emacs Lisp handle, `lisp.SYMBOL.copy()` has the effect of `lisp.SYMBOL.value().copy()`, that is, it returns the value of the symbol as opened as possible.

A symbol may also be used as if it was a Python function, in which case it really names an Emacs Lisp function that should be applied over the following function arguments. The result of the Emacs Lisp function becomes the value of the call, with all due conversions of course.

5.3 Dynamic bindings

As Emacs Lisp uses dynamic bindings, it is common that Emacs Lisp programs use **let** for temporarily setting new values for some Emacs Lisp variables having global scope. These variables recover their previous value automatically when the **let** gets completed, even if an error occurs which interrupts the normal flow of execution.

Pymacs has a **Let** class to represent such temporary settings. Suppose for example that you want to recover the value of `lisp.mark()` when the transient mark mode is active on the Emacs Lisp side. One could surely use `lisp.mark(True)` to *force* reading the mark in such cases, but for the sake of illustration, let's ignore that, and temporarily deactivate transient mark mode instead. This could be done this way:

```
try:
    let = Let()
    let.push(transient_mark_mode=None)
    ... USER CODE ...
finally:
    let.pop()
```

`let.push()` accepts any number of keywords arguments. Each keyword name is interpreted as an Emacs Lisp symbol written the Pymacs way, with underlines. The value of that Emacs Lisp symbol is saved on the Python side, and the value of the keyword becomes the new temporary value for this Emacs Lisp symbol. A later `let.pop()` restores the previous value for all symbols which were saved together at the time of the corresponding `let.push()`. There may be more than one `let.push()` call for a single **Let** instance, they stack within that instance. Each

`let.pop()` will undo one and only one `let.push()` from the stack, in the reverse order or the pushes.

A single call to `let.pops()` automatically does all pending `let.pop()` at once, in the correct reverse order. When the **Let** instance disappears, either because the programmer does `del let` or `let = None`, or just because the Python **let** variable goes out of scope, `let.pops()` gets executed under the scene, so the **try/finally** statement may be omitted in practice. For this omission to work flawlessly, the programmer should be careful at not keeping extra references to the **Let** instance.

The constructor call `let = Let()` also has an implied initial `.push()` over all given arguments, given there is any, so the explicit `let.push()` may be omitted as well. In practice, this sums up and the above code could be reduced to a mere:

```
let = Let(transient_mark_mode=None)
... USER CODE ...
```

Be careful at assigning the result of the constructor to some Python variable. Otherwise, the instance might disappear immediately after having been created, restoring the Emacs Lisp variable much too soon.

Any variable to be bound with **Let** should have been bound in advance on the Emacs Lisp side. This restriction usually does no kind of harm. Yet, it will likely be lifted in some later version of Pymacs.

The **Let** class has other methods meant for some macros which are common in Emacs Lisp programming, in the spirit of **let** bindings. These method names look like `push_*` or `pop_*`, where Emacs Lisp macros are `save-*`. One has to use the matching `pop_*` for undoing the effect of a given `push_*` rather than a mere `.pop()`: the Python code is clearer, this also ensures that things are undone in the proper order. The same **Let** instance may use many `push_*` methods, their effects nest.

`push_excursion()` and `pop_excursion()` save and restore the current buffer, point and mark. `push_match_data()` and `pop_match_data()` save and restore the state of the last regular expression match. `push_restriction()` and `pop_restriction()` save and restore the current narrowing limits. `push_selected_window()` and `pop_selected_window()` save and restore the fact that a window holds the cursor. `push_window_excursion()` and `pop_window_excursion()` save and restore the current window configuration in the Emacs display.

As a convenience, `let.push()` and all other `push_*` methods return the **Let** instance. This helps chaining various `push_*` right after the instance generation. For example, one may write:

```
let = Let().push_excursion()
if True:
    ... USER CODE ...
del let
```

The `if True:` (use `if 1:` with older Python releases, some people might prefer writing `if let:` anyway), has the only goal of indenting *USER CODE*, so the scope of the **let** variable is made very explicit. This is purely stylistic, and not at all necessary. The last `del let` might be omitted in a few circumstances, for example if the excursion lasts until the end of the Python function.

5.4 Raw Emacs Lisp expressions

Pymacs offers a device for evaluating a raw Emacs Lisp expression, or a sequence of such, expressed as a string. One merely uses **lisp** as a function, like this:

```
lisp('''
...
POSSIBLY-LONG-SEQUENCE-OF-LISP-EXPRESSIONS
...
''')
```

The Emacs Lisp value of the last or only expression in the sequence becomes the value of the **lisp** call, after conversion back to Python.

5.5 User interaction

Emacs functions have the concept of user interaction for completing the specification of their arguments while being called. This happens only when a function is interactively called by the user, it does not happen when a function is directly called by another. As Python does not have a corresponding facility, a bit of trickery was needed to retrofit that facility on the Python side.

After loading a Python module but prior to creating an Emacs view for this module, Pymacs decides whether loaded functions will be interactively callable from Emacs, or not. Whenever a function has an **interaction** attribute, this attribute holds the Emacs interaction specification for this function. The specification is either another Python function or a string. In the former case, that other function is called without arguments and should, maybe after having consulted the user, return a list of the actual arguments to be used for the original function. In the latter case, the specification string is used verbatim as the argument to the `(interactive ...)` function on the Emacs side. To get a short reminder about how this string is interpreted on the Emacs side, try `C-h f interactive RET` within Emacs. Here is an example where an empty string is used to specify that an interactive has no arguments:

```
from Pymacs import lisp

def hello_world():
    "`Hello world' from Python."
    lisp.insert("Hello from Python!")
hello_world.interaction = ''
```

Versions of Python released before the integration of PEP 232 do not allow users to add attributes to functions, so there is a fall-back mechanism. Let's presume that a given function does not have an **interaction** attribute as explained above. If the Python module contains an **interactions** global variable which is a dictionary, if that dictionary has an entry for the given function with a value other than **None**, that function is going to be interactive on the Emacs side. Here is how the preceding example should be written for an older version of Python, or when portability is at premium:

```
from Pymacs import lisp
interactions = {}

def hello_world():
```

```
        "`Hello world' from Python."
        lisp.insert("Hello from Python!")
    interactions[hello_world] = ''
```

One might wonder why we do not merely use `lisp.interactive(...)` from within Python. There is some magic in the Emacs Lisp interpreter itself, looking for that call *before* the function is actually entered, this explains why `(interactive ...)` has to appear first in an Emacs Lisp **defun**. Pymacs could try to scan the already compiled form of the Python code, seeking for `lisp.interactive`, but as the evaluation of **lisp.interactive** arguments could get arbitrarily complex, it would a real challenge un-compiling that evaluation into Emacs Lisp.

5.6 Key bindings

An interactive function may be bound to a key sequence.

To translate bindings like `C-x w`, say, one might have to know a bit more how Emacs Lisp processes string escapes like `\C-x` or `\M-\C-x` in Emacs Lisp, and emulate it within Python strings, since Python does not have such escapes. `\C-L`, where L is an upper case letter, produces a character which ordinal is the result of subtracting 0x40 from ordinal of `L`. `\M-` has the ordinal one gets by adding 0x80 to the ordinal of following described character. So people can use self-inserting non-ASCII characters, `\M-` is given another representation, which is to replace the addition of 0x80 by prefixing with Escape, that is 0x1b. So `\C-x` in Emacs is `\x18` in Python. This is easily found, using an interactive Python session, by giving it: `chr(ord('X') - ord('A') + 1)`.

An easier way would be using the **kbd** function on the Emacs Lisp side, like with `lisp.kbd('C-x w')` or `lisp.kbd('M-<f2>')`.

To bind the F1 key to the **helper** function in some **module**:

```
lisp.global_set_key((lisp.f1,), lisp.module_helper)
```

`(item,)` is a Python tuple yielding an Emacs Lisp vector. `lisp.f1` translates to the Emacs Lisp symbol **f1**. So, Python `(lisp.f1,)` is Emacs Lisp `[f1]`. Keys like `[M-f2]` might require some more ingenuity, one may write either `(lisp['M-f2'],)` or `(lisp.M_f2,)` on the Python side.

6 Debugging

Finding bugs in a program is an art, which may be difficult enough already when there is a single process and a single language. Pymacs involves a part (usually short) written in Emacs Lisp and another part (usually more substantial) written in Python, each running in their own process. Both processes communicate which each other. Moreover, to get debugging hints, Emacs is often the necessary door by which the programming user may catch glimpses on what is happening on both sides.

To effectively debug Pymacs code, one benefits from having some familiarity with the communication protocol, and also from knowing how to observe both sides of this protocol at once. The usual way is through the ***Pymacs*** buffer within Emacs, which shows an Emacs view the whole protocol. One may also view by forcing the Pymacs helper to save a trace file, which shows a Python view the whole protocol — unless there are communication errors, this should tell the same story as with the ***Pymacs***

buffer. These few topics are developed in the three following sections. The remaining sections address more specific issues about Emacs Lisp or Python debugging.

6.1 The communication protocol

The Pymacs communication protocol is rather simple deep down, merely using evaluation on arrival on both sides. All the rest is recursion trickery over that simple idea.

- It is more easy to generate than to parse. Moreover, Emacs has a Lisp parser and Python has a Python parser. So, when preparing a message to the Pymacs helper, Emacs generates Python code for Python to parse, and when preparing a message for Emacs, Python generates Emacs Lisp expressions for Emacs to parse.
- Messages are exchanged in strictly alternating directions (from Python to Emacs, from Emacs to Python, etc.), the first message being sent by the Pymacs helper (from Python to Emacs) just after it started, identifying the current Pymacs version.
- Messages in both directions have a similar envelope. Each physical message has a prefix, the message contents, and a newline. The prefix starts with either < or > to mark the directionality, is followed by the decimal expression of the contents length counted in characters, and terminates with a single horizontal tab. The count excludes the prefix, but includes the newline.
- In each direction, messages are made up of two elements: an action keyword and a single argument (yet the argument may sometimes be complex). As a special case, memory cleanup messages from Python to Emacs use four elements: the atom **free**, a list of slot numbers to free, and then the real action and argument. This is because the cleanup is delayed and piggy-backed over some other message.
- For Emacs originated messages, the action and the argument are separated by a space. For Python originated messages, the action and the argument are made into a Lisp list.
- Most actions in the following table are available in both directions, unless noted. The first three actions *start* a new level of Pymacs evaluation, the two remaining actions end the current level.
  - **eval** requests the evaluation of its expression argument.
  - **exec** requests the execution of its statement argument (this may only be received on the Python side).
  - **expand** requests the opening of an Emacs Lisp structure (this may only be received on the Emacs side).
  - **return** represents the normal reply to a request, the argument holds the value to be returned (**nil** in case of **exec**).
  - **raise** represents the error reply to a request, the argument then holds a diagnostic string.

Python evaluation is done in the context of the **Pymacs.pymacs** module. On the Emacs Lisp side, there is no concept of module name spaces, so we internally use the `pymacs-` prefix as an attempt to stay clean. Users should ideally refrain from naming their Emacs Lisp objects with a `pymacs-` prefix.

The protocol may be fragile to interruption requests, so it tries to recognize each message action before evaluation is attempted. The idea (not fully implemented yet) is to make the protocol part immune to interruptions, but to allow evaluations themselves to be interrupted.

6.2 The *Pymacs* buffer

Emacs and Python are two separate processes (well, each may use more than one process). Pymacs implements a simple communication protocol between both, and does whatever needed so the programmers do not have to worry about details. The main debugging tool is the communication buffer between Emacs and Python, which is named ***Pymacs***.

As it is sometimes helpful to understand the communication protocol, it is briefly explained here, using an artificially complex example to do so. Consider (this example assumes Python 2):

```
(pymacs-eval "lisp('(pymacs-eval \"repr(2L**111)\")')")
"2596148429267413814265248164610048L"
```

Here, Emacs asks Python to ask Emacs to ask Python for a simple bignum computation. Note that Emacs does not natively know how to handle big integers, nor has an internal representation for them. This is why I use the **repr** function, so Python returns a string representation of the result, instead of the result itself. Here is a trace for this example. Imagine that Emacs stands on the left and that Python stands on the right. The < character flags a message going from Python to Emacs, while the > character flags a message going from Emacs to Python. The number gives the length of the message, including the end of line. (Acute readers may notice that the first number is incorrect, as the version number gets replaced in the example while this manual is being produced.)

```
<22     (version "0.25")
>43     eval lisp('(pymacs-eval "repr(2L**111)")')
<45     (eval (progn (pymacs-eval "repr(2L**111)")))
>19     eval repr(2L**111)
<47     (return "2596148429267413814265248164610048L")
>45     return "2596148429267413814265248164610048L"
<47     (return "2596148429267413814265248164610048L")
```

Part of the protocol manages memory, and this management generates some extra-noise in the ***Pymacs*** buffer. Whenever Emacs passes a structure to Python, an extra pointer is generated on the Emacs side to inhibit garbage collection by Emacs. Python garbage collector detects when the received structure is no longer needed on the Python side, at which time the next communication will tell Emacs to remove the extra pointer. It works symmetrically as well, that is, whenever Python passes a structure to Emacs, an extra Python reference is generated to inhibit garbage collection on the Python side. Emacs garbage collector detects when the received structure is no longer needed on

the Emacs side, after which Python will be told to remove the extra reference. For efficiency, those allocation-related messages are delayed, merged and batched together within the next communication having another purpose.

Variable **pymacs-trace-transit** may be modified for controlling how and when the ***Pymacs*** buffer, or parts thereof, get erased. By default, this buffer gets erased before each transaction. To make good debugging use of it, first set **pymacs-trace-transit** to either **t** or to some `(KEEP . LIMIT)`.

6.3 Debugging the Pymacs helper

The Pymacs helper is a Python program which accepts options and arguments. The available options, which are only meant for debugging, are:

-d FILE	Debug the protocol to FILE
-s FILE	Trace received signals to FILE

- The `-d` option saves a copy of the communication protocol in the given file, as seen from the Pymacs helper. The file should be fairly identical to the contents of the ***Pymacs*** buffer within Emacs.
- The `-s` option monitors most signals received by the Pymacs helper and logs them in the given file. Each log line merely contains a signal number, possibly followed by a star if the interruption was allowed in. Besides logging, signals are usually ignored.

The arguments list directories to be added at the beginning of the Python module search path, and whenever Emacs launches the Pymacs helper, the contents of the Emacs Lisp **pymacs-load-path** variable is turned into this argument list.

The Pymacs helper options may be set through the **PYMACS_OPTIONS** environment variable. For example, one could execute something like:

```
export PYMACS_OPTIONS='-d /tmp/pymacs-debug -s /tmp/pymacs-signals'
```

in a shell (presuming **bash** here) and start Emacs from that shell. Then, when Emacs launches the Pymacs helper, the above options are transmitted to it.

6.4 Emacs usual debugging

If cross-calls between Emacs Lisp and Python nest deeply, an error will raise successive exceptions alternatively on both sides as requests unstack, and the diagnostic gets transmitted back and forth, slightly growing as we go. So, errors will eventually be reported by Emacs. I made no kind of effort to transmit the Emacs Lisp back trace on the Python side, as I do not see a purpose for it: all debugging is done within Emacs windows anyway.

On recent Emacses, the Python back trace gets displayed in the mini-buffer, and the Emacs Lisp back trace is simultaneously shown in the ***Backtrace*** window. One useful thing is to allow to mini-buffer to grow big, so it has more chance to fully contain the Python back trace, the last lines of which are often especially useful. Here, I use:

```
(setq resize-mini-windows t
      max-mini-window-height .85)
```

in my `.emacs` file, so the mini-buffer may use 85% of the screen, and quickly shrinks when fewer lines are needed. The mini-buffer contents disappear at the next keystroke, but you can recover the Python back trace by looking at the end of the ***Messages*** buffer. In which case the **ffap** package in Emacs may be yet another friend! From the ***Messages*** buffer, once **ffap** activated, merely put the cursor on the file name of a Python module from the back trace, and `C-x C-f RET` will quickly open that source for you.

6.5 Python usual debugging

A common way to debug a Python script is to spread it with **print** commands. When such a Python script is executed under Pymacs control, these **print** statements display the results right within the ***Pymacs*** buffer, and may be observed there.

As such output gets intermixed with the Pymacs protocol itself, never ever print the symbol <, immediately followed by the expression of a decimal number, immediately followed by a horizontal tab (`\t`). If you were doing so, the communication protocol would get pretty mixed up, and Pymacs would break. But you do not have to worry much about this: the forbidden sequence is unlikely in practice, would it be only because people do not often use horizontal tabs anymore — oh, tabs were once undoubtedly popular, but this was many years ago...

6.6 Auto-reloading on save

I found useful to automatically **pymacs-load** some Python files whenever they get saved from Emacs. This can be decided on a per-file or per-directory basis. To get a particular Python file to be reloaded automatically on save, add the following lines at the end:

```
# Local Variables:
# pymacs-auto-reload: t
# End:
```

Here is an example of automatic reloading on a per-directory basis. The code below assumes that Python files meant for Pymacs are kept in `~/share/emacs/python`:

```
(defun fp-maybe-pymacs-reload ()
  (let ((pymacsdir (expand-file-name "~/share/emacs/python/")))
    (when (and (string-equal (file-name-directory buffer-file-name)
                             pymacsdir)
               (string-match "\\.py\\'" buffer-file-name))
      (pymacs-load (substring buffer-file-name 0 -3)))))
(add-hook 'after-save-hook 'fp-maybe-pymacs-reload)
```

7 Administrative miscellany

7.1 Development history

I once hungered for a Python-extensible editor, so much so that I pondered the idea of dropping Emacs for other avenues, but found nothing much convincing. Moreover, looking at all Lisp extensions I'd made for myself, and considering all those superb tools written by others, all of which are now part of my computer life, it would have

been a huge undertaking for me to reprogram these all in Python. So, when I began to see that something like Pymacs was possible, I felt strongly motivated! :-)

Pymacs draws on previous work of Cedric Adjih that enabled the running of Python as a process separate from Emacs. See http://www.crepuscule.com/pyemacs/, or write Cedric at mailto:adjih-pam@crepuscule.com. Cedric presented **pyemacs** to me as a proof of concept. As I simplified that concept a bit, I dropped the `e` in `pyemacs` :-). Cedric also previously wrote patches for linking Python right into XEmacs, but abandoned the idea, as he found out that his patches were unmaintainable over the evolution of both Python and XEmacs.

As Brian McErlean independently and simultaneously wrote a tool similar to this one, we decided to merge our projects. In an amusing coincidence, he even chose **pymacs** as a name. Brian paid good attention to complex details that escaped my courage, so his help and collaboration have been beneficial. You may reach Brian at mailto:brianmce@crosswinds.net.

The initial throw at Pymacs has been written on 2001-09-05, and releases in the 0.x series followed in a rapid pace for a few months, and Pymacs soon became stable. Reported bugs or suggestions were minor, and the feature set was fairly usable from the start. For a long while, there was not enough new material to warrant other releases.

Later, someone begged me to consider Vim, and not only Emacs, for some tools I was then writing (in the area of musical scores). Looking at Vim more closely, I discovered that it is a worth editor, with Python nicely integrated, enough for me to switch. In a Web article (which many enjoyed, as they told me), I detailed my feelings on these matters.

I switched from Emacs to Vim in my day-to-day habits, and because of this, felt that Pymacs needed a more credible maintainer than me. Syver Enstad, who was an enthusiastic user and competent contributor, was kind enough to accept the duty (2003-10). Syver then became unavailable, to the point I could not contact him in years. I would loathe to see myself interfering with an official maintainer, but after I decided to return to some moderate Emacs usage, and because of the long silence, I considered resuming Pymacs maintenance (2007-11), and did it (2008-01).

Giovanni Giorgi once (2007-03) wanted to expand on Pymacs and publish it on his own, and later felt like maintaining it whole (late 2007-12). I rather suggested an attempt at collaborative maintenance, and this experiment is still going on...

7.2 Should it come with Emacs?

Gerd Möllman, who was maintaining Emacs at the time of Pymacs birth and development, retrofitted (2001-09) the idea of a **post-gc-hook** from XEmacs, as a way to facilitate memory management within Pymacs.

Richard Stallman once suggested (2001-10) that Pymacs be distributed within Emacs, and while discussing the details of this, I underlined small technical difficulties about Emacs installing the Python parts, and the need of a convention about where to install Python files meant for Pymacs. As Richard felt, at the time, very overwhelmed with other duties, no decision was taken and the integration went nowhere.

After Gerd resigned as an Emacs maintainer, someone from the Emacs development team wrote again (2002-01) asking information about how to integrate Pymacs. It was easy for me to write a good and thorough summary, after all these discussions with Richard. And that's the end of the story: I never heard of it again. :-)

7.3 The future of Pymacs

Some people suggested important internal Pymacs changes. In my opinion, new bigger features are better implemented in a careful way, first as examples or contributions, and moved closer to internal integration depending on how users use or appreciate them. For now, Pymacs should concentrate at doing its own humble job well, and resist bloat.

Before Pymacs closes to some version 1.0, some specifications should be revisited, user suggestions pondered, porting matters documented. The test suite should grow up, we should collect more examples. Pymacs should aim seamless integration with `.el` files and with transparent **autoload** (my little tries were not so successful). On the Python side, Pymacs *might* fake primitives like **getindex** and **putindex**, and better support iterators and some newer Python features.

Pymacs is not much geared towards Python threads. It is not clear yet if it would be reasonably tractable to better support them.

8 Technical miscellany

8.1 Known bugs or limitations

What is the difference between a bug and a limitation? *Limitations* are either bugs not worth repairing, or else, bugs that we do not know yet how to repair. While documenting a bug is indeed a way to postpone its solution, it does not necessarily turns it into a limitation.

On a mailing list I once closely followed, a few maintainers were getting very, very upset whenever the word *bug* happened to be used in any message, especially if the bug was documented. A distinguished member on this list (William N. Venable) coined the wonderful word *unfelicity*, as a way to discuss problems while avoiding human damage.

Such delicacies are surely unneeded for Pymacs. A bug is a bug!

8.1.1 Needed control on stack unwinding

As Ali Gholami Rudi nicely summarized it (2008-02-12):

> *Lisp programmers could use* **inhibit-quit** *at various levels of recursion, and use Pymacs at these various levels. As an Emacs* **quit** *might propagate out of the stack, but stopping at various levels of it when the Lisp programmers took measures for it, I think there is no choice that finding some mechanism by which Python will unstack in parallel with Emacs, that is, no more and no less, so if Emacs resumes processing at some intermediate level, Python should be ready at the exact corresponding level on its side.*

By doing `pymacs-eval "(time.sleep(10))"`, and quitting, I once saw that:

- Emacs does not interrupt at once, and if **inhibit-quit** remains set while Emacs waits for the Pymacs helper, this is surely not user friendly!
- At the end of the wait, I get a spurious IO error (I do not know where it comes from).

8.1.2 Possible memory leak

Memory may leak in some theoretical circumstances (I say theoretical, because no one ever reported this as being an actual problem). As Richard Stallman once put it (2002-08):

> *I wonder, though, can this* [memory management] *technique fully handle cycles that run between Lisp and Python? Suppose Lisp object A refers to Python object B, which refers to Lisp object A, and suppose nothing else refers to either one of them. Will you succeed in recognizing these two objects as garbage?*

8.1.3 Death from a Ctrl-C

Ali Gholami Rudi notices (2008-02-20) that Pymacs dies over:

```
M-x pymacs-eval RET lisp.kbd('C-c r r') RET
```

as there is a `Ctrl-C` in the value returned from Emacs.

8.2 Suggestions to ponder

8.2.1 Python-driven Pymacs

I guess the most important improvement we could think to Pymacs would be some machinery by which Python programs, started outside Emacs, could access Pymacs, once it started. That could be useful at least for testing or debugging, and maybe for more serious work as well. These are mere thoughts, I do not plan working at this soon, unless I have an actual need. But if the challenge interests someone, please go ahead!

Here is how it could go. Pymacs has a Python interpreter running as a sub-process of Emacs. In fact, Emacs loads `pymacs.el`, which in turn gets Python to execute `Pymacs.py`, and both communicate afterwards. `Pymacs.py` is only active whenever `pymacs.el` calls it, otherwise it is blocked. `Pymacs.py` could, under some option, start another thread within itself. The initial thread would block waiting for Emacs, as usual. The second thread would block waiting to serve any Python client wanting to access Emacs. When this occurs, the second thread would queue a request for the first thread, and then send a signal to Emacs so it triggers a Pymacs communication. At each communication opportunity, the first thread on the Python side might fully service the queue from the second thread.

8.2.2 Autoloading interface

I once tried better interfacing to **autoload**, and failed. It got more intricate that I thought it would be. I might revisit this, but in low priority.

In the meantime, one may use a small `.el` file, like this one, on the Emacs load path:

```
# File zorglub.el -- just load zorglub.py.
(pymacs-load "zorglub")
(provide 'zorglub)
```

and then use either one of:

```
(require 'zorglub)              ; in Lisp
lisp.require(lisp.zorglub)      # in Python
```

at the beginning of body for any function needing functions from `zorglub.py`. One may also write one or many:

```
(autoload 'FUNCTION-NAME "zorglub" nil t)
```

to indirectly autoload `zorglub.py` as needed.

8.2.3 Handling more special forms

The discussion started about the lack of specific Pymacs support, on the Python side, for the Emacs Lisp **setq-default** function. People also mentioned **defvar** and **defcustom**, but there are really many other special forms in Emacs Lisp. (A special form is any expression form in which all arguments are not all blindly evaluated before the function actually enters. The function then receives the arguments unevaluated, and it is its responsibility to choose which arguments should be evaluated, and when.)

The fact is that, besides **setq** and some forms of **defun**, functions, few special forms are supported in Pymacs. One may think of **let**, functions like **save-excursion**, etc. But that's all, and maybe debatable as too much already. The real problem to solve is supporting special forms (and macros) at Pymacs level. If we create special cases in Pymacs for each special form we happen to stumble upon, Pymacs might loose its elegance, and so, we have to stay a bit careful.

All special forms require that the user somehow defeat the fact that Pymacs evaluate all function arguments before calling a Lisp function. I realise it might be a subtle point for people unfamiliar with Lisp. **apply** on the Lisp side applies a function on a list of arguments, so the trick is to evaluate on the Python side something yielding a list, the contents of which are to be actual arguments. I'm not fully sure this is the good direction to take, even if easy — I mean here, that the real problem to solve is something else.

On a related matter, Ali Gholami Rudi suggested that Pymacs supports Emacs so-called *keyword arguments*, and even provide a simple patch to do so:

```
diff --git a/Pymacs.py b/Pymacs.py
--- a/Pymacs.py
+++ b/Pymacs.py
@@ -453,13 +453,16 @@
             write(') nil)')
             lisp._eval(''.join(fragments))

-    def __call__(self, *arguments):
+    def __call__(self, *arguments, **keywords):
         fragments = []
         write = fragments.append
         write('(%s' % self.text)
         for argument in arguments:
             write(' ')
             print_lisp(argument, write, True)
+        for kwd, value in keywords.items():
+            write(' :%s ' % kwd)
+            print_lisp(value, write, True)
```

```
        write(')')
        return lisp._eval(''.join(fragments))
```

So far that I understand, there are just no keyword arguments in Emacs. Keywords might be nothing but a mirage created by **defcustom** only (maybe through **define-minor-mode**) and **defstruct** -- is there any other usage for keywords? So I wonder if this unusual trickery, not even a real part of Emacs Lisp, is important enough to warrant modifying something as fundamental as **__call__** in Pymacs. Part of my reluctance might also come from my (unsubstanciated) fear that the above change would slow down the hearth of Pymacs.

For now at least, users are invited to use `lisp(...)` for all other special forms. It's simple, it's rather safe. Things like:

```
lisp('(setq-default %s %s)' % (name, value))
```

are not so horrible... :-) Deep down, `lisp()` calls are what Pymacs do all the time under the table, all the rest are bits of sugar. What would be needed is a visit to this special form support with wider eyes and mind, come up with a general unifying solution, rather than multiplying special cases.

8.2.4 Support for Python dictionaries

While Pymacs mirrors Python tuples and lists into Emacs Lisp vectors and lists, it has nothing currently to reflect Python dictionaries.

It has been suggested to use Emacs Lisp alists to do so, but this does not seem adequate to me. Pymacs 0.0 and 0.1 did convert Python dicts to Emacs Lisp alists. This was a mere toy to get experience with the Pymacs mechanics, not a serious idea. Despite I wanted *something* for Python dicts, this choice was not very satisfying:

- Dicts access speed are O(1); alists are O(N).
- Dicts have no intrinsic order; alists are really a sequence.
- Dicts have no duplicate keys; alists may have shadows.

The last two points, in particular, have the consequence that one cannot convert back and forth from Lisp and have results which compare with `(equal ...)`. This makes the equivalence especially ugly. Proper lists and vectors in Lisp can be converted back and forth to Python and be `(equal ...)`, so those equivalences are bearable. The dict conversion was withdrawn in Pymacs 0.2; I thought I should better postpone until a better idea pops up, than let users develop habits with something wrong and doomed to be replaced.

Emacs Lisp hash tables (as in Emacs 21) could be an acceptable equivalent for Python dicts. This is what Brian McErlean did, and suggests. My only reservation is about the Python need for non-mutable keys, something which Emacs does not guarantee. As by default, from Lisp to Python, references are transmitted instead of contents, this would be a possible problem only when an expanded copy is requested from the Python side. This would never be a problem going from Python to Emacs, so far as I understand things now.

8.2.5 A nicer *Pymacs* buffer

We might improve how the ***Pymacs*** communication buffer looks. Let's sketch this quickly, in any case, I'm not sure how worth this is. The buffer might be turned into a

more fully featured Emacs mode, so it can benefit from highlighting and colourisation, and other goodies. The first thing would be to install font-lock definitions. The second thing would be to use indenting to show the proper nesting of calls between Emacs and Python, in both directions. I would prefer this to be done as a display feature, not as part of the communication protocol. A third thing would be to automatically interpret object numbers on both sides, replacing them with clearer text whenever possible — this information may often be deduced from earlier communications. Finally, that mode could allow for some inspection on Pymacs object and status, and maybe also to control the external Python server described in another suggestion in this series, if it ever gets implemented.

8.3 Speed issues

Shoot out projects compare the relative speed of many popular languages, and the relative merits of Lisp and Python might interest Pymacs users. The first URL points to a version oriented towards Win32 systems, the second is more recent but Debian-oriented:

- http://dada.perl.it/shootout/index.html
- http://shootout.alioth.debian.org/

I've not heard of any Python to Lisp compiler. Lisp may be slow or fast depending on how one uses it, and how much one uses declarations. Some Lisp systems have really excellent compilers, that give very fast code when properly hinted.

Python itself may be slow or fast, once again depending on how one uses it. With the proper bend, one can develop the habit of writing Python which shows honest speed. And there is always Pyrex (and the very similar Cython), which is Python complemented with explicit declarations (a bit like some Lisp implementations), and which can buy a lot of speed.

This is quite likely that one can have fast programs while using Python, or a mix of Python and either Pyrex or Cython (or even Psyco sometimes), that is, within Python paradigms, without feeling any need of resorting to Lisp.

If Python looks like being slow while being used with Emacs, the problem probably lies in Emacs-Python communication which Pymacs implements. One has to learn how to do the proper compromises for having less communications. (In that regard, Vim and Python are really linked together, so Python in Vim is likely faster than Pymacs for someone who does not pay special attention to such matters.)

Ali Gholami Rudi also writes (2008-02):

> *Well, there seems to be lots of overhead when transferring large strings. Transferring them requires:*
>
> 1. *escaping characters in the strings*
> 2. *putting them in* ****Pymacs**** *buffer*
> 3. *sending the region to Python process*
> 4. *evaluating the Python string in Python-side (involves compiling)*
>
> *In my experiments, transferring a ~5k-line file takes more than a second on a relatively new computer (data from* **rope-dev***). Improving that probably requires a new protocol that does not use Python eval and has an optional*

debug buffer. Probably few applications need to transfer large strings to Python but if they do, it is quite slow.

All in all, speed may sometimes become a real issue for Pymacs. I once wrote within http://pinard.progiciels-bpi.ca/opinions/editors.html :

> *While Pymacs is elegant in my opinion, one cannot effectively use Pymacs (the Python part) without knowing at least the specification of many Lisp functions, and I found that it requires some doing for a Pymacs developer to decouple the Emacs interaction part from the purer algorithmic part in applications. Moreover, if you do not consider speed issues, they bite you.*

8.4 Vim-related thoughts

Emacs Lisp is deeply soldered into Emacs internals. Vim has its own language, which people sometimes call Vimscript, similarly tightened into Vim. My feeling is that Emacs Lisp allows for a more intimate handling of edit buffers and external processes than Vimscript does, yet this intimacy has a price in complexity, so all totalled, they may be perceived as comparable for most practical purposes.

Pymacs allows customising Emacs with Python instead of Emacs Lisp, and then runs Python as a process external to Emacs, with a communication protocol between both processes. Python may be built into Vim, and then both Python and Vim use a single process. The same as Pymacs gives access to almost all of Emacs Lisp, Python within Vim gives access to almost all of Vimscript, but with a much smaller overhead than Pymacs.

Pymacs is not Emacs Lisp, and Python in Vim is not Vimscript either, tweaks are needed in both cases for accessing some of the underlying scripting facilities. Pymacs is rather elegant, Python in Vim is rather clean. Python itself is both elegant and clean, but one strong point of Python for me is the legibility, which builds deeper roots on the clean side than on the elegant side. All in all, despite I know how debatable it can be, I guess I now have a prejudice towards Python in Vim.

I figured out a simple way to have the same Python source usable both within Pymacs or Vim. However, Emacs is byte oriented, while Vim is line oriented. In a few Pymacs applications of mine, I internally toggle between line orientation and byte orientation, keeping both for speed most probably, while I see things would be a bit simpler (and maybe slower) if I was pushing myself on the line-oriented side. Each of Emacs and Vim have their own logic and elegance, and it is probable that we loose overall if we try to emulate one with the other.

The idea traversed me to convert all the few Pymacs examples so they work both for Pymacs and Vim, and through the documentation, publicise how people writing Python extensions could write them for both editors at once. Yet, while doing so, one has to stretch either towards Emacs or Vim, and I guess I would favour Vim over Emacs when the time comes to evaluate efficiency-related choices.

I also thought about writing a Pymacs module for running Python scripts already written for Vim, by offering a compatibility layer. The complexity of this might be unbounded, I should study actual Python scripts for Vim before knowing better if this is thinkable or not.