Pymacs version 0.22

Extending Emacs with Python, updated 2003-06-30

Francois Pinard, pinard@iro.umontreal.ca

1 Presentation

1.1 What is Pymacs?

Pymacs is a powerful tool which, once started from Emacs, allows both-way communication between Emacs Lisp and Python. Pymacs aims Python as an extension language for Emacs rather than the other way around, and this assymetry 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-recursivity.

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.

`http://www.iro.umontreal.ca/~pinard/pymacs/` contains a copy of the Pymacs manual file in HTML form. The distribution holds the documentation sources, as well as PostScript and PDF renderings. The canonical Pymacs distribution is available as `http://www.iro.umontreal.ca/~pinard/pymacs/Pymacs.tar.gz`. Report problems and suggestions to `mailto:pinard@iro.umontreal.ca`.

1.2 Warning to Pymacs users.

I expect average Pymacs users to have a deeper knowledge of Python than Emacs Lisp. Some examples at the end of this file are meant for Python users having a limited experience with the Emacs API. Currently, there are only contains two examples, one is too small, the other is too big :-). As there is no dedicated mailing list nor discussion group for Pymacs, let's use `python-list@python.org` for asking questions or discussing Pymacs related matters.

This is beta status software : specifications are slightly frozen, yet changes may still happen that would require small adaptations in your code. Report problems to Francois Pinard at `pinard@iro.umontreal.ca`. For discussing specifications or making suggestions, please also copy the `python-list@python.org` mailing list, to help brain-storming! :-)

1.3 History and references.

I once starved for a Python-extensible editor, and pondered the idea of dropping Emacs for other avenues, but found nothing much convincing. Moreover, looking at all LISP extensions I wrote for myself, and considering all those superb tools written by others and that became 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 revisits previous Cedric Adjih's works about running Python as a process separate from Emacs. See `http://www.crepuscule.com/pyemacs/`, or write Cedric at `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.

Brian McErlean independently and simultaneously wrote a tool similar to this one, we decided to join our projects. 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 `brianmce@crosswinds.net`.

Another reference of interest is Doug Bagley’s shoot out project, which compares the relative speed of many popular languages. The first URL points to the original, the second points to a newer version oriented towards Win32 systems :

```
http://www.bagley.org/~doug/shootout/
http://dada.perl.it/shootout/index.html
```

2 Installation.

2.1 Install the Pymacs proper.

Currently, there are two installation scripts, and both should be run. If you prefer, you may use '`make install lispdir=`*lispdir*', where *lispdir* is some directory along the list kept in your Emacs `load-path`.

The first installation script installs the Python package, including the Pymacs examples, using the Python standard Distutils tool. Merely '`cd`' into the Pymacs distribution, then execute '`python setup.py install`'. To get an option reminder, do '`python setup.py install --help`'. Check the Distutils documentation if you need more information about this.

The second installation script installs the Emacs Lisp part only. (It used to do everything, but is now doomed to disappear completely.) Merely '`cd`' into the Pymacs distribution, then run '`python setup -ie`'. This will invite you to interactively confirm which Lisp installation directory, *if* the script discovers more than one reasonable possibility for it. Without '`-ie`', the Lisp part of Pymacs will be installed in some automatically guessed place. Use '`-n`' to known about the guess without proceeding to the actual installation. '`./setup -E xemacs ...`' may be useful to XEmacs lovers. See '`./setup -H`' for all options.

About Win32 systems, Syver Enstad says : *For Pymacs to operate correctly, one should create a batch file with '`pymacs-services.bat`' as a name, which runs the '`pymacs-services`' script. The '`.bat`' file could be placed along with '`pymacs-services`', wherever that maybe.*.

To check that '`pymacs.el`' is properly installed, start Emacs and give it the command '`M-x load-library RET pymacs`' : you should not receive any error. 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. To check that '`pymacs-services`' is properly installed, type '`pymacs-services </dev/null`' in a shell; you should then get a line ending with '`(pymacs-version` *version*`)`', and another saying : '`Protocol error : '>' expected.`'.

Currently, there is only one installed Pymacs example, which comes in two parts : a batch script '`rebox`' and a `Pymacs.rebox` module. To check that both are properly installed, type '`rebox </dev/null`' in a shell; you should not receive any output nor see any error.

2.2 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-load "pymacs" nil t)
(autoload 'pymacs-eval "pymacs" nil t)
(autoload 'pymacs-apply "pymacs")
(autoload 'pymacs-call "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 '`$HOME/.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`'. Emacs should prompt you for a Python expression. Try '`` `2L**111` ``' (type the internal backquotes, but not the external quotes). The minibuffer should display '`2596148429267413814265248164610048L`'. '`M-x pymacs-load`' should prompt you for a Python module name. Reply '`os`'. After Emacs prompts you for a prefix, merely hit Enter to accept the default prefix. This should have the effect of importing the Python `os` module within Emacs. Typing '`M- : (os-getcwd)`' should echo the current directory in the message buffer, as returned by the `os.getcwd` Python function.

2.3 Porting Pymacs.

Pymacs has been developped on Linux, Python 1.5.2, and Emacs (20 and 21), it is expected to work out of the box on most other Unices, newer Python and Emacs releases, and also with XEmacs. While I tried to maintain Python 1.5.2 compatibility for the Pymacs proper, I am not testing it anymore, and now rely on testers for reporting portability bugs, if any.

Syver Enstad reports that Pymacs could be made to work on Windows-2000 (win2k), he suspects it should equally work with NT and XP. However, little shell stunts may be required, I hope to later document them here.

The distribution also contains a manual in Allout format, and an Allout format processor which depends on Python 2.2 features. However, since ready Postscript and PDF renderings of the manual are also provided, you should not need Python 2.2 for getting an unaltered manual.

There might be Python 2.2 dependencies for the `Nn` example, while I believe the `Rebox` is still 1.5.2 compatible. This might change.

2.4 Caveats.

Some later versions of Emacs 20 silently ignore the request for creating weak hash tables, they create an ordinary table instead. Older Emacses just do not have hash tables. Pymacs should run on all, yet for these, memory will leak on the Python side whenever complex objects get transmitted to Emacs, as these objects will not be reclaimed on the Python side once Emacs is finished with them. It should not be a practical problem in most simple cases.

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` and the Python empty list '`[]`' are returned as `nil` 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 narrow 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, except Unicode, are always converted into Emacs Lisp strings.

Emacs Lisp symbols yield the special '`lisp.`*symbol*' or '`lisp[`*string*`]`' notations on the Python side. The first notation is used when the Emacs Lisp symbol starts with a letter, and contains only letters, digits and hyphens, in which case Emacs Lisp hyphens get replaced by Python underscores. This convention is welcome, as Emacs Lisp programmers commonly prefer using dashes, where Python programmers use underlines. Otherwise, the second notation is used. Conversely, '`lisp.`*symbol*' on the Python side yields an Emacs Lisp symbol with underscores replaced with hyphens, while '`lisp[`*string*`]`' corresponds to an Emacs Lisp symbol printed with that *string* which, of course, should then be a valid Emacs Lisp symbol name.

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 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 neccessarily `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, Unicode strings, 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 :

```
(setq matcher (pymacs-eval "re.compile('pattern').match"))
(pymacs-call matcher argument)
```

the initial `setq` 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 `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.

4.2 `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.

4.3 `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`.

4.4 `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.

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.

4.5 Expected usage.

We do not expect that `pymacs-eval`, `pymacs-call` or `pymacs-apply` will be much used, if ever. In practice, the Emacs Lisp side of a Pymacs application might call `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.

These imported functions are usually those which are of interest for the user, and the preferred way to call Python services with Pymacs.

4.6 Special Emacs Lisp variables.

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

4.6.1 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, `pymacs-services` is called, 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`.

4.6.2 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)`’.

4.6.3 pymacs-forget-mutability

The default behaviour of Pymacs is to transmit Emacs Lisp objects to Python in such a way thay 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 overriden 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 assymetry 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.

4.6.4 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.

4.6.5 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 minibuffer.

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.

5 Usage on the Python side.

5.1 Python setup.

Pymacs requires little or no setup in the Python modules which are meant to be used from Emacs, for the simple situations where these modules receive nothing but Emacs `nil`, numbers or strings, or return nothing but Python `None`, numbers or strings.

Otherwise, use '`from Pymacs import lisp`'. If you need more Pymacs features, like the `Let` class, write '`from Pymacs import lisp, Let`'.

5.2 Response mode.

When Python receives a request from Emacs in the context of Pymacs, and until it returns the reply, Emacs keeps listening to serve Python requests. Emacs is not listening otherwise. Other Python threads, if any, may not call Emacs without *very* careful synchronisation.

5.3 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).

Except for '`lisp.nil`' or '`lisp["nil"]`', which are the same as `None`, 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 '`lisp.t`'.

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.

5.4 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(lisp.t)`' 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.

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, all remaining '`let.pop()`' get automatically executed, 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, 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 would 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.

5.5 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.

5.6 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 programmatically 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' 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 fallback 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 preceeding 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.

5.7 Keybindings.

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 'ESC', that is 0x1b.

So '`\C-x`' in Emacs is '\x18' in Python. This is easily found, using an interactive Python session, by givin 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.

6.1 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*`. 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*`)`'. As it is sometimes helpful to understand the communication protocol, it is briefly explained here, using an artificially complex example to do so. Consider :

```
(pymacs-eval "lisp('(pymacs-eval \"`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 backticks, so Python returns a string representation of the result, instead of the result itself. Here is a trace for this example. The '`<`' character flags a message going from Python to Emacs and is followed by an expression written in Emacs Lisp. The '`>`' character flags a message going from Emacs to Python and is followed by a expression written in Python. The number gives the length of the message.

```
<22   (pymacs-version "0.3")
>49   eval("lisp('(pymacs-eval \"`2L**111`\")')")
<25   (pymacs-eval "`2L**111`")
>18   eval("`2L**111`")
<47   (pymacs-reply "2596148429267413814265248164610048L")
>45   reply("2596148429267413814265248164610048L")
<47   (pymacs-reply "2596148429267413814265248164610048L")
```

Python evaluation is done in the context of the `Pymacs.pymacs` module, so for example a mere `reply` really means '`Pymacs.pymacs.reply`'. On the Emacs Lisp side, there is no concept of module namespaces, so we use the '`pymacs-`' prefix as an attempt to stay clean. Users should ideally refrain from naming their Emacs Lisp objects with a '`pymacs-`' prefix.

`reply` and `pymacs-reply` are special functions meant to indicate that an expected result is finally transmitted. `error` and `pymacs-error` are special functions that introduce a string which explains an exception which recently occurred. `pymacs-expand` is a special function implementing the '`copy()`' methods of Emacs Lisp handles or symbols. In all other cases, the expression is a request for the other side, that request stacks until a corresponding reply is received.

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.

6.2 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 backtrace 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 backtrace gets displayed in the mini-buffer, and the Emacs Lisp backtrace 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 backtrace, 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 backtrace 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 backtrace, and '`C-x C-f RET`' will quickly open that source for you.

6.3 Auto-reloading on save.

I found useful to automatically `pymacs-load` some Python files whenever they get saved from Emacs. Here is how I do it. 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 Examples.

7.1 Example 1 — Paul Winkler's.

7.1.1 Example 1 — The problem.

Let's say I have a a module, call it 'manglers.py', containing this simple python function :

```
        def break_on_whitespace(some_string) :
    words = some_string.split()
    return '\n'.join(words)
```

The goal is telling Emacs about this function so that I can call it on a region of text and replace the region with the result of the call. And bind this action to a key, of course, let's say [f7].

The Emacs buffer ought to be handled in some way. If this is not on the Emacs Lisp side, it has to be on the Python side, but we cannot escape handling the buffer. So, there is an equilibrium in the work to do for the user, that could be displaced towards Emacs Lisp or towards Python.

7.1.2 Example 1 — Python side.

Here is a first draft for the Python side of the problem :

```
        from Pymacs import lisp

        def break_on_whitespace() :
    start = lisp.point()
    end = lisp.mark(lisp.t)
    if start > end :
        start, end = end, start
    text = lisp.buffer_substring(start, end)
    words = text.split()
    replacement = '\n'.join(words)
    lisp.delete_region(start, end)
    lisp.insert(replacement)

        interactions = {break_on_whitespace : ''}
```

For various stylistic reasons, this could be rewritten into :

```
        from Pymacs import lisp
        interactions = {}

        def break_on_whitespace() :
    start, end = lisp.point(), lisp.mark(lisp.t)
    words = lisp.buffer_substring(start, end).split()
    lisp.delete_region(start, end)
    lisp.insert('\n'.join(words))
```

```
        interactions[break_on_whitespace] = ''
```

The above relies, in particular, on the fact that for those Emacs Lisp functions used here, '`start`' and '`end`' may be given in any order.

7.1.3 Example 1 — Emacs side.

On the Emacs side, one would do :

```
(pymacs-load "manglers")
(global-set-key [f7] 'manglers-break-on-whitespace)
```

7.2 Example 2 — Yet another Gnus backend.

Note. This example is not fully documented yet. As it stands, it is merely a collection of random remarks from other sources.

7.2.1 Example 2 — The problem.

I've been reading, saving and otherwise handling electronic mail from within Emacs for a lot of years, even before Gnus existed. The preferred Emacs archiving disk format for email is Babyl storage, and the special `Rmail` mode in Emacs handles Babyl files. With years passing, I got dozens, then hundreds, then thousands of such Babyl files, each of which holds from as little as only one to maybe a few hundreds individual messages. I tried to taylor `Rmail` mode in various ways to MIME, foreign charsets, and many other nitty-gritty habits. One of these habits was to progressively eradicate paragraphs in messages I was visiting many times, as users were often using a single message to report many problems or suggestions all at once, while I was often addressing issues one at a time.

When I took maintenance of some popular packages, like GNU `tar`, my volume of daily email raised drastically, and I choose Gnus as a way to sustain the heavy load. I thought about converting all my Babyl files to `nnml` format, but this would mean loosing many tools I wrote for Babyl files, consuming a lot of i-nodes, and also much polluting my `*Group*` buffer. I rather chose to select and read Babyl files as ephemeral mail groups (and for doing so, developed Emacs user machinery so selection could be done very efficiently). Gnus surely gave me for free nice MIME and cryptographic features, and a flurry of handsome and useful commands, compared to previous `Rmail` mode. On the other hand, Gnus did not allow me to modify invidual messages in Babyl files, so for a good while, I had to give up on some special handling, like eradicating paragraphs as I used to do.

This pushed me into writing my own Gnus backend for Babyl files : making sure I correctly implement the article editing and modification support of the backend API. I chose Python to do so because I already had various Python tools for handling Babyl files, because I wanted to connect other Python scripts to the common mechanics, and of course because Pymacs was making this project feasable. Nowadays, Babyl file support does not go very far beyond Emacs itself, while many non-Emacs tools for handling Unix mailbox folders are available. Spam fighting concerns brought me to revisit the idea of massively transforming all my Babyl files to Unix mailbox format, and I discovered that it would be a breeze to do, if I only adapted the Python backend to handle Unix mailbox files as well as Babyl, transparently.

7.2.2 Example 2 — Python side.

I started by taking the Info nodes of the Gnus manual which were describing the back end interface, and turning them all into a long Python comment. I then split that comment into one dummy function per back end interface function, meant to write some debugging information when called, and then return failure to Gnus. This was enough to explore what functions were needed, and in which circumstances. I then implemented enough of them so ephemeral Babyl groups work, while solid groups might require more such functions. The unimplemented functions are still sitting in the module, with their included comments and debugging code.

7.2.3 Example 2 — Emacs side.

One difficulty is ensuring that `Nn` contents ('`nncourrier.py`' and '`folder.py`') have to be on the Python or Pymacs search path. The '`__init__.py`' and package nature are not essential.

7.3 Example 3 — The `rebox` tool.

7.3.1 Example 3 — The problem.

For comments held within boxes, it is painful to fill paragraphs, while stretching or shrinking the surrounding box *by hand*, as needed. This piece of Python code eases my life on this. It may be used interactively from within Emacs through the Pymacs interface, or in batch as a script which filters a single region to be reformatted.

In batch, the reboxing is driven by command options and arguments and expects a complete, self-contained boxed comment from a file. Emacs function `rebox-region` also presumes that the region encloses a single boxed comment. Emacs `rebox-comment` is different, as it has to chase itself the extent of the surrounding boxed comment.

7.3.2 Example 3 — Python side.

The Python code is too big to be inserted in this documentation : see file '`Pymacs/rebox.py`' in the Pymacs distribution. You will observe in the code that Pymacs specific features are used exclusively from within the `pymacs_load_hook` function and the `Emacs_Rebox` class. In batch mode, `Pymacs` is not even imported. Here, we mean to discuss some of the design choices in the context of Pymacs.

In batch mode, as well as with `rebox-region`, the text to handle is turned over to Python, and fully processed in Python, with practically no Pymacs interaction while the work gets done. On the other hand, `rebox-comment` is rather Pymacs intensive : the comment boundaries are chased right from the Emacs buffer, as directed by the function `Emacs_Rebox.find_comment`. Once the boundaries are found, the remainder of the work is essentially done on the Python side.

Once the boxed comment has been reformatted in Python, the old comment is removed in a single delete operation, the new comment is inserted in a second operation, this occurs in `Emacs_Rebox.process_emacs_region`. But by doing so, if point was within the boxed comment before the reformatting, its precise position is lost. To well preserve point, Python might have driven all reformatting details directly in the Emacs buffer. We really preferred doing it all on the Python side : as we gain legibility by expressing the algorithms in pure Python, the same Python code may be used in batch or interactively, and we avoid the slowdown that would result from heavy use of Emacs services.

To avoid completely loosing point, I kludged a `Marker` class, which goal is to estimate the new value of point from the old. Reformatting may change the amount of white space, and either delete or insert an arbitrary number characters meant to draw the box. The idea is to initially count the number of characters between the beginning of the region and point, while ignoring any problematic character. Once the comment has been reboxed, point is advanced from the beginning of the region until we get the same count of characters, skipping all problematic characters. This `Marker` class works fully on the Python side, it does not involve Pymacs at all, but it does solve a problem that resulted from my choice of keeping the data on the Python side instead of handling it directly in the Emacs buffer.

We want a comment reformatting to appear as a single operation, in the context of Emacs Undo. The method `Emacs_Rebox.clean_undo_after` handles the general case for this. Not that we do so much in practice : a reformatting implies one `delete-region` and one `insert`, and maybe some other little adjustements at `Emacs_Rebox.find_comment` time. Even if this method scans and mofifies an Emacs Lisp list directly in the Emacs memory, the code doing this stays neat and legible. However, I found out that the undo list may grow quickly when the Emacs buffer use markers, with the consequence of making this routine so Pymacs intensive that most of the CPU is spent there. I rewrote that routine in Emacs Lisp so it executes in a single Pymacs interaction.

Function `Emacs_Rebox.remainder_of_line` could have been written in Python, but it was probably not worth going away from this one-liner in Emacs Lisp. Also, given this routine is often called by `find_comment`, a few Pymacs protocol interactions are spared this way. This function is useful when there is a need to apply a regexp already compiled on the Python side, it is probably better fetching the line from Emacs and do the pattern match on the Python side, than transmitting the source of the regexp to Emacs for it to compile and apply it.

For refilling, I could have either used the refill algorithm built within in Emacs, programmed a new one in Python, or relied on Ross Paterson's `fmt`, distributed by GNU and available on most Linuxes. In fact, `refill_lines` prefers the latter. My own Emacs setup is such that the built-in refill algorithm is *already* overridden by GNU `fmt`, and it really does a much better job. Experience taught me that calling an external program is fast enough to be very bearable, even interactively. If Python called Emacs to do the refilling, Emacs would itself call GNU `fmt` in my case, I preferred that Python calls GNU `fmt` directly. I could have reprogrammed GNU `fmt` in Python. Despite interesting, this is an uneasy project : `fmt` implements the Knuth refilling algorithm, which depends on dynamic programming techniques; Ross did carefully fine tune them, and took care of many details. If GNU `fmt` fails, for not being available, say, `refill_lines` falls back on a dumb refilling algorithm, which is better than none.

7.3.3 Example 3 — Emacs side.

The Emacs recipe appears under the '`Emacs usage`' section, near the beginning of '`Pymacs/rebox.py`', so I do not repeat it here.

Table of Contents