python-emacs

Release 0.2.0

Jared Lumpe

Jan 07, 2024

CONTENTS:

CHAPTER

ONE

BASIC USAGE

Most of the functionality in this package is implemented in the *EmacsBatch* and *EmacsClient* classes, which allow you to evaluate Emacs Lisp code and get the result in Python. Both inherit from *EmacsBase* and share the same API.

1.1 EmacsBatch

EmacsBatch runs `emacs --batch` with each invocation. The `args` constructor argument is a list of additional command line arguments to add. It is often a good idea to use the `-Q` option to avoid loading personal configuration files each time, which can slow things down.

```
>>> from emacs import EmacsBatch
>>> emacs = EmacsBatch(args=['-Q'])
```

1.2 EmacsClient

EmacsClient uses the `emacsclient` command to connect to and execute code in a running Emacs server. A server can be started in a running Emacs process by calling `(server-start <server-name>)`. Alternatively you can start a daemon server with `emacs --daemon=<server-name>`.

```
>>> from emacs import EmacsClient
>>> emacs = EmacsClient(server="my-server")
```

1.3 Executing Emacs lisp code

The main job of the interface is to execute elisp code. You can do this using the *EmacsBase.eval()* method:

```
>>> emacs.eval('(+ 1 2)')
3
```

The source code can be passed in as a string, or you can build an Elisp expression using the *emacs.elisp* subpackage. This allows you to easily pass in data from Python:

```
>>> import emacs.elisp as el
>>> def emacs_add(a, b):
...     expr = el.funccall('+', a, b)
...     return emacs.eval(expr)
```

(continues on next page)

(continued from previous page)

```
>>> emacs_add(1, 2)
3
```

Note that it does this by converting the value to JSON in Emacs and then decoding it in Python, so the value must be json-encodable.

Errors in evaluating the expression are caught in Emacs (see the `catch_errors` argument to *EmacsBase.eval()*) and raised as an *ElispException* in Python:

```
>>> emacs_add(1, "foo")
Traceback (most recent call last):
ElispException: Wrong type argument: number-or-marker-p, "foo"
```

CHAPTER

TWO

MANIPULATING ELISP CODE IN PYTHON

The *emacs.elisp* module contains utilities for building and manipulating Emacs Lisp (Elisp) expressions in Python. These can then be passed to an *EmacsBatch* or *EmacsClient* instance to be executed.

2.1 Expr Objects

Elisp expressions are represented by subtypes of the *Expr* abstract base class:

- *Literal*(value) wraps Python ints, strs, and floats.
- *Symbol*(name: str) represents a symbol.
- *Cons*(car: Expr, cdr: Expr) represents a cons cell.
- *List*(items: Iterable[Expr]) represents a list.
- *Quote*(expr: Expr) represents a quoted expression.
- *Raw*(src: str) can be used to wrap a raw Elisp code string.

Generally you should use the functions detailed in the following section to build expressions rather than instantiating them directly.

You can use str(expr) to produce (hopefully) syntactically-correct Elisp code.

2.2 Building Elisp expressions

The *to_elisp()* function can be used to convert various Python values to Elisp expressions. Elements of composite data types (lists, tuples, dicts) are converted recursively. Most parts of this package's API will use *to_elisp()* to convert arguments that are not already instances of *Expr*, so it is often not necessary to use it directly.

2.2.1 Basic data types

to_elisp() converts numbers and strings to literals and bools and None to the correct symbols:

```
>>> import emacs.elisp as el

>>> el.to_elisp(123)
<el 123>

>>> el.to_elisp(1.23)
```

(continues on next page)

(continued from previous page)

```
<el 1.23>

>>> el.to_elisp('foo')
<el "foo">

>>> el.to_elisp(True)
<el t>

>>> el.to_elisp(False)
<el nil>

>>> el.to_elisp(None)
<el nil>
```

The `nill` and `t` symbols are also available as *`nil`* and *`el_true`*.

2.2.2 Symbols

Create a symbol with the *`symbol()`* function:

```
>>> el.symbol('foo')
<el foo>
```

The *`symbols()`* function can be used to create a list of symbols:

```
>>> el.symbols('a', 'b', 'c')
<el (a b c)>
```

2.2.3 Lists

`el_list()` converts any iterable to a list expression:

```
>>> el.el_list(range(1, 5))
<el (1 2 3 4)>
```

`to_elisp()` converts Python lists to quoted Elisp lists, while tuples are left unquoted:

```
>>> el.to_elisp([1, 2, 3])
<el '(1 2 3)>

>>> el.to_elisp(('a', 'b', 'c'))
<el ("a" "b" "c")>
```

2.2.4 Function calls

Function call expressions can be created with *funccall()*, or by calling a *Symbol* instance. Keyword arguments are converted to `kebab-case` and prefixed with a ":" character.

```
>>> el.funccall('+', 1, 2)
<el (+ 1 2)>

>>> foo = el.symbol('foo')
>>> foo(el.symbol('x'), el.symbol('y'), kw_arg=123)
<el (foo x y :kw-arg 123)>
```

2.2.5 Quoting

The *quote()* method produces a quoted version of an expression:

```
>>> s = el.symbol('foo')
>>> s.quote()
<el 'foo>

>>> el.symbols('a', 'b', 'c').quote()
<el '(a b c)>
```

The *q* property acts as a shortcut:

```
>>> s.q
<el 'foo>
```

2.2.6 Cons cells

An expression that must be constructed directly because it has no Python equivalent is the cons cell, represented with the class *Cons*:

```
>>> c = el.cons(el.symbol('a'), 1)
>>> c
<el (cons a 1)>

>>> c.q
<el '(a . 1)>
```

2.2.7 Mapping formats (alists and plists)

You can use *make_alist()* or *make_plist()* to convert mapping types like `dict`s to their Elisp equivalents. These functions will always treat string keys as symbols:

```
>>> el.make_alist({'a': 1, 'b': 2}).q
<el '((a . 1) (b . 2))>

>>> el.make_plist({':x': 1, ':y': 2}).q
<el '(:x 1 :y 2)>
```

to_elisp() converts mapping types like dicts to plists or alists, depending on the value of the `dict_format` argument (defaults to `"alist"`.

2.2.8 Raw code strings

Finally, use *Raw* to wrap a raw Elisp code string to be inserted verbatim in the given location:

```
>>> el.Raw('(print "hi")')
<el (print "hi")>

>>> el.el_list([1, 2, el.Raw('(+ a b)')])
<el (1 2 (+ a b))>
```

2.3 Elisp DSL

This package also includes an unholy abomination of a DSL that lets you write Elisp code in Python. It is implemented through the singleton object *emacs.elisp.E*.

Calling the singleton as a function converts a Python object into an Elisp object using *to_elisp()*:

```
>>> from emacs.elisp import E
>>> E(3)
<el 3>

>>> E('foo')
<el "foo">

>>> E(['a', 'b', 'c'])
<el '("a" "b" "c")>
```

Attribute access produces Elisp symbols, converting `snake_case` to `kebab-case`. The same can be done by indexing with a string (without the case conversion):

```
>>> E.abc
<el abc>

>>> E.foo_bar
<el foo-bar>

>>> E[':baz']
<el :baz>
```

Symbols can be called as functions, generating Elisp function calls:

```
>>> E.message("Hello from %s", E('python-emacs'))
<el (message "Hello from %s" "python-emacs")>

>>> E['='](E.a, E.b)
<el (= a b)>
```

Additionally, the `C`, `S`, and `R` methods are aliases for *cons*, *symbols()*, and *Raw*, respectively.

Using just the `E` object, it is possible to write complex Elisp expressions:

```
>>> E.defun(E.my_elisp_function, E.S('a', 'b'),
...   E.message("I am a crime against God."),
...   E['+'](E.a, E.b))
<el (defun my-elisp-function (a b) (message "I am a crime against God.") (+ a b))>
```

CHAPTER

THREE

PYTHON API

3.1 Emacs interface

Interface with Emacs and run commands.

exception emacs.emacs.**ElispException**

Bases: Exception

An exception caught in Emacs while evaluating an expression.

message

Error message, from `error-message-string`.

Type str

symbol

Error symbol, the `car` of the caught error object.

Type str

data

Error data, the `cdr` of the caught error object.

Type Any

expr

The expression that was being evaluated.

Type *emacs.elisp.ast.Expr*

proc

Completed process object.

Type subprocess.CompletedProcess

__init__(*message*, *symbol*, *data*, *expr*, *proc*)

Parameters

- **message** (*str*) –
- **symbol** (*str*) –
- **expr** (emacs.elisp.ast.Expr) –
- **proc** (*subprocess.CompletedProcess*) –

class emacs.emacs.**EmacsBase**

Bases: abc.ABC

Abstract base class for an interface to GNU Emacs.

cmd
Command to run with each invocation.

Type Tuple[str]

logger
Logger instance (not yet implemented).

Type logging.Logger

__init__(*cmd*, *logger=None*)

Parameters

- **cmd** (*Sequence[str]*) –
- **logger** (*Optional[logging.Logger]*) –

eval(*src*, ***, *catch_errors=True*, *ret='value'*, *is_json=False*, *extra_args=None*, *tmpfile=False*, ***kw*)
Evaluate Elisp source code.

Parameters

- **src** (*Union[str, emacs.elisp.ast.Expr, Sequence[Union[str, emacs.elisp.ast.Expr]]]*) – Elisp code. If a list of strings/expressions will be enclosed in `progn`.
- **catch_errors** (*bool*) – Catch errors evaluating the expression in Emacs and raise an *ElispException* in Python.
- **ret** (*Optional[str]*) – What to return. `'value'` returns the value of the evaluated expression (must be something that can be JSON-encoded using `(json-encode)`. `'subprocess'` returns the `subprocess.CompletedProcess`` of the command that was run (can be used to get the raw output). `'both'` returns a tuple `(value, process)`. `'none'` or `None` returns nothing. Use `subprocess` or `none` to avoid processing a potentially large amount of output you don't care about.
- **extra_args** (*Optional[Iterable[str]]*) – Additional arguments to pass to command.
- **tmpfile** (*bool*) – Read result through temporary file instead of stdout. This may avoid certain issues `emacsclient` has when printing large amounts of output, or if the expression also has the side effect of printing to stdout.
- **is_json** (*bool*) – If the result of evaluating `src` is already a json-encoded string that should be decoded.
- **kw** – Passed to *run()*.

Raises

- ***ElispException*** – If `catch_errors=True` and there is an error in Emacs when evaluating the expression.
- **subprocess.CalledProcessError** – If `check=True` and return code is nonzero.

Returns See the `ret` argument.

Return type Any

run(*args*, ***, *check=True*, *run_kw=None*)
Run the Emacs command with a list of arguments.

Parameters

- **args** (*Sequence[str]*) – Arguments to run command with.
- **check** (*bool*) – Check the return code is zero.
- **run_kw** – Keyword arguments to pass to `subprocess.run()`.

Raises `subprocess.CalledProcessError` – If `check=True` and return code is nonzero.

Returns Completed process object.

Return type subprocess.CompletedProcess

class emacs.emacs.**EmacsBatch**

Bases: *emacs.emacs.EmacsBase*

Interface to Emacs program using `emacs --batch`.

Parameters

- **cmd** (*Tuple[str]*) – Base command to run. Name or path of emacs executable.
- **args** – Additional arguments to add to `cmd`.

__init__(*cmd='emacs'*, ***, *args=None*, *logger=None*)

Parameters

- **cmd** (*str*) –
- **args** (*Optional[Sequence[str]]*) –
- **logger** (*Optional[logging.Logger]*) –

class emacs.emacs.**EmacsClient**

Bases: *emacs.emacs.EmacsBase*

Interface to running Emacs server using `emacsclient`.

Parameters

- **cmd** (*Tuple[str]*) – Base command to run. Name or path of emacsclient executable.
- **args** – Additional arguments to add to `cmd`.
- **server** – Name of server to connect to.

__init__(*cmd='emacsclient'*, ***, *args=None*, *server=None*, *logger=None*)

Parameters

- **cmd** (*str*) –
- **args** (*Optional[Sequence[str]]*) –
- **server** (*Optional[str]*) –
- **logger** (*Optional[logging.Logger]*) –

emacs.emacs.**el_catch_err_json**(*expr*, *encoded=False*)

Elisp snippet to catch errors and return the error message as encoded JSON.

Parameters

- **expr** (emacs.elisp.ast.Expr) –
- **encoded** (*bool*) –

Return type *emacs.elisp.ast.Expr*

emacs.emacs.**el_encode_json**(*expr*)
Elisp snippet to encode value as JSON.

Parameters **expr** (emacs.elisp.ast.Expr) –

Return type *emacs.elisp.ast.Expr*

emacs.emacs.**make_cmd**(**parts*)
Concatenate arguments or lists of arguments to make command.

Parameters **parts** (*Optional[Union[str, Sequence[str]]]*) –

Return type *List*[str]

3.2 Elisp Expressions

Build and print Emacs Lisp abstract syntax trees in Python.

3.2.1 AST Classes

Base classes for Emacs Lisp abstract syntax trees.

class emacs.elisp.ast.**Cons**
Bases: *emacs.elisp.ast.Expr*

A cons cell.

__init__(*car*, *cdr*)

Parameters

- **car** (emacs.elisp.ast.Expr) –
- **cdr** (emacs.elisp.ast.Expr) –

class emacs.elisp.ast.**Expr**
Bases: object

Base for classes which represent Elisp expressions.

quote()
Return a quoted form of this expression.

Return type *emacs.elisp.ast.Expr*

property q
Shortcut for `self.quote()`.

class emacs.elisp.ast.**List**
Bases: *emacs.elisp.ast.Expr*

An Elisp list expression.

items
Items in the list.

Type Tuple[*emacs.elisp.ast.Expr*, ...]

__init__(*items*)

Parameters **items** (*Iterable[emacs.elisp.ast.Expr]*) –

class emacs.elisp.ast.**Literal**

Bases: *emacs.elisp.ast.Expr*

Basic self-evaluating expressions like strings, numbers, etc.

pyvalue

The Python value of the literal.

Type Union[str, int, float]

__init__(*pyvalue*)

Parameters **pyvalue** (*Union[str, int, float]*) –

class emacs.elisp.ast.**Quote**

Bases: *emacs.elisp.ast.Expr*

A quoted Elisp expression.

expr

The quoted Elisp expression.

__init__(*expr*)

Parameters **expr** (*emacs.elisp.ast.Expr*) –

class emacs.elisp.ast.**Raw**

Bases: *emacs.elisp.ast.Expr*

Just raw Elisp code to be pasted in at this point.

src

Raw Elisp source code.

Type str

__init__(*src*)

Parameters **src** (*str*) –

class emacs.elisp.ast.**Symbol**

Bases: *emacs.elisp.ast.Expr*

An Elisp symbol.

__init__(*name*)

Parameters **name** (*str*) –

3.2.2 Creating expressions

Functions to build Elisp expressions (more) easily.

emacs.elisp.exprs.**cons**(*car*, *cdr*, *, *convert_kw=None*)
Create a cons cell expression, converting both arguments first.

Parameters

- **car** –
- **cdr** –
- **convert_kw** – Keyword arguments to *to_elisp()*.

Return type *emacs.elisp.ast.Cons*

emacs.elisp.exprs.**el_bool**(*value*)
Convert a Python boolean to the standard Elisp representation.

Parameters **value** (*bool*) –

Return type *emacs.elisp.ast.Expr*

emacs.elisp.exprs.**el_list**(*items*, *, *convert_kw=None*)
Create an Elisp list expression, converting items to Elisp expressions if needed.

Parameters

- **items** (*Iterable*) – Contents of list.
- **convert_kw** – Keyword arguments to *to_elisp()*.

Return type *emacs.elisp.ast.List*

emacs.elisp.exprs.**funccall**(*f*, **args*, ***kw*)
Create a function call expression.

Parameters

- **f** (*Union[str, emacs.elisp.ast.Symbol]*) – Function name or symbol
- **args** – Function arguments. Will be converted to Elisp expressions if necessary.
- **kw** – Keyword arguments. Argument names are converted like `my_num=1` -> `:my-num 1`.

emacs.elisp.exprs.**get_src**(*src*)
Get Elisp source code as *Expr* instance.

Parameters **src** (*Union[str, emacs.elisp.ast.Expr, Sequence[Union[str, emacs.elisp.ast.Expr]]]*) – Elisp expression(s) as either a string containing raw Elisp code, a single `Expr`, or a list of these.

Returns Source code as single expression. If the input was a list it will be enclosed in a `progn` block.

Return type *Expr*

emacs.elisp.exprs.**let**(*assignments*, **body*)
Make a "let" expression.

Parameters

- **assignments** – Mapping from variable names (as symbols or strings) to values.
- **body** – Expressions to add to body.

Return type *emacs.elisp.ast.List*

emacs.elisp.exprs.**make_alist**(*pairs*, ***kw*)
Create an alist expression from a set of key-value pairs.

Parameters

- **pairs** (*Union[collections.abc.Mapping, Iterable[Tuple]]*) – Key-value pairs as a dict or collections of 2-tuples.
- **quote** – Quote the resulting expression.
- **kw** – Keyword arguments passed to *to_elisp()* to convert mapping values.

Return type *emacs.elisp.ast.Expr*

emacs.elisp.exprs.**make_plist**(*pairs*, ***kw*)
Create a plist expression from a set of key-value pairs.

Parameters

- **pairs** (*Union[collections.abc.Mapping, Tuple[Any, Any]]*) – Key-value pairs as a dict or collections of 2-tuples.
- **kw** – Keyword arguments passed to *to_elisp()* to convert mapping values.

Return type *emacs.elisp.ast.Expr*

emacs.elisp.exprs.**symbol**(*name*, *kebab=False*, *keyword=False*)
Convert argument to symbol.

Parameters

- **name** (*Union[str, emacs.elisp.ast.Symbol]*) – Symbol name as string. Alternatively, an existing symbol instance which will be returned unchanged.
- **kebab** (*bool*) – Convert name from `snake_case` to `kebab-case`.
- **keyword** (*bool*) – Prefix name with ":" character if it doesn't start with it already.

Return type *emacs.elisp.ast.Symbol*

emacs.elisp.exprs.**symbols**(**names*)
Create an Elisp list of symbols.

Parameters **names** (*Union[str, emacs.elisp.ast.Symbol]*) – Symbol names.

Return type *emacs.elisp.ast.Expr*

emacs.elisp.exprs.**to_elisp**(*value*, ***kw*)
emacs.elisp.exprs.**to_elisp**(*v*, ***kw*)
emacs.elisp.exprs.**to_elisp**(*v*, ***kw*)
emacs.elisp.exprs.**to_elisp**(*v*, ***kw*)
emacs.elisp.exprs.**to_elisp**(*v*, ***kw*)
emacs.elisp.exprs.**to_elisp**(*v*, ***kw*)
emacs.elisp.exprs.**to_elisp**(*v*, ***kw*)
emacs.elisp.exprs.**to_elisp**(*v*, ***kw*)
emacs.elisp.exprs.**to_elisp**(*v*, ***kw*)
emacs.elisp.exprs.**to_elisp**(*value*, ***kw*)
Convert a Python value to an Elisp expression.

The following conversions are supported:

- `True` to `t` symbol.
- `False` and `None` to `nil` symbol.
- `int`, `float`, and `str` to literals.

- `tuple` to unquoted elisp list.
- `list` to quoted elisp list.
- `dict` and other mapping types to either alist or plist, see the `dict_format` argument.
- *Expr* instances are returned unchanged.

For compound types, their contents are recursively converted as well.

Parameters

- **value** – Python value to convert.
- **dict_format** (*str*) – Elisp format to convert dicts/mappings to. Either `'alist'` (default) or `'plist'`.

Returns

Return type *Expr*

emacs.elisp.exprs.**el_true = <el t>**
The standard Elisp representation of True

Return type *List*

emacs.elisp.exprs.**nil = <el nil>**
The `nil` symbol

Return type *List*

3.2.3 DSL

A DSL for writing Elisp in Python.

God help us all.

class emacs.elisp.dsl.**ElispDSL**
Bases: `object`

Implements the Elisp DSL.

R
alias of *emacs.elisp.ast.Raw*

static C(*car*, *cdr*, *, *convert_kw=None*)
Create a cons cell expression, converting both arguments first.

Parameters

- **car** –
- **cdr** –
- **convert_kw** – Keyword arguments to *to_elisp()*.

Return type *emacs.elisp.ast.Cons*

static S(**names*)
Create an Elisp list of symbols.

Parameters **names** (*Union[str, emacs.elisp.ast.Symbol]*) – Symbol names.

Return type *emacs.elisp.ast.Expr*

emacs.elisp.dsl.**E = <emacs.elisp.dsl.ElispDSL object>**
Instance of *ElispDSL* for easy importing.

Return type *emacs.elisp.ast.Expr*

3.2.4 Misc

Other utility code relating to Elisp.

emacs.elisp.util.**escape_emacs_char**(*c*)
Escape character for use in Elisp string literal.

Parameters **c** (*Union[int, str]*) –

Return type str

emacs.elisp.util.**escape_emacs_string**(*s*, *quotes=False*)
Escape non-printable characters in a way that can be read by Emacs.

If `quotes=True` this returns a valid Elisp string literal that evaluates to `s` and can be read by the `read` function.

Parameters

- **s** (*str*) – String to escape.
- **quotes** (*bool*) – Surround output with double quote characters.

Return type str

emacs.elisp.util.**snake_to_kebab**(*name*)
Convert a symbol name from `'snake_case'` to `'kebab-case'`.

Parameters **name** (*str*) –

Return type str

emacs.elisp.util.**unescape_emacs_string**(*s*, *quotes=False*)
Unescape the representation of a string printed by Emacs.

This can be used to parse string printed using `prin1`, for example.

Important: this requires the Emacs variables `print-escape-newlines` and `print-escape-control-characters` be set to `t` for certain control and whitespace characters to be escaped properly. See here for more information.

Parameters

- **s** (*str*) – String to escape.
- **quotes** (*bool*) – Expect contents of `s` to be surrounded by double quote characters.

Return type str

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

Symbols

C

D

E

F

G

I

L

M

N

P

Q

R

S

T

U