neovim-prompt Documentation

Release 0.1.0

lambdalisue

October 30, 2016

Contents

CHAPTER 1

prompt package

1.1 Submodules

1.2 prompt.action module

Action module.

class prompt.action.**Action**
Bases: object

Action class which holds action callbacks.

registry
dict

An action dictionary.

call(*prompt*, *name*)
Call a callback of specified action.

Parameters

- **prompt** (Prompt) – A prompt.prompt.Prompt instance.
- **name** (*str*) – An action name.

Example

```
>>> from unittest.mock import MagicMock
>>> from .prompt import STATUS_ACCEPT, STATUS_CANCEL
>>> prompt = MagicMock()
>>> action = Action()
>>> action.register_from_rules([
...     ('prompt:accept', lambda prompt: STATUS_ACCEPT),
...     ('prompt:cancel', lambda prompt: STATUS_CANCEL),
... ])
>>> action.call(prompt, 'prompt:accept')
1
>>> action.call(prompt, 'unknown:accept')
1
>>> action.call(prompt, 'unknown:unknown')
Traceback (most recent call last):
```

```
    ...
AttributeError: No action "unknown:unknown" has registered.
```

Returns None or int which represent the prompt status.

Return type None or int

classmethod **from_rules**(*rules*)

Create a new action instance from rules.

Parameters **rules** (*Iterable*) – An iterator which returns rules. A rule is a (name, callback) tuple.

Example

```
>>> from .prompt import STATUS_ACCEPT, STATUS_CANCEL
>>> Action.from_rules([
...     ('prompt:accept', lambda prompt: STATUS_ACCEPT),
...     ('prompt:cancel', lambda prompt: STATUS_CANCEL),
... ])
<....action.Action object at ...>
```

Returns An action instance.

Return type *Action*

register(*name*, *callback*)

Register action callback to a specified name.

Parameters

- **name** (*str*) – An action name which follow {namespace}:{action name}
- **callback** (*Callable*) – An action callback which take a `prompt.prompt.Prompt` instance and return None or int.

Example

```
>>> from .prompt import STATUS_ACCEPT
>>> action = Action()
>>> action.register('prompt:accept', lambda prompt: STATUS_ACCEPT)
```

register_from_rules(*rules*) → None

Register action callbacks from rules.

Parameters **rules** (*Iterable*) – An iterator which returns rules. A rule is a (name, callback) tuple.

Example

```
>>> from .prompt import STATUS_ACCEPT, STATUS_CANCEL
>>> action = Action()
>>> action.register_from_rules([
...     ('prompt:accept', lambda prompt: STATUS_ACCEPT),
```

```
...     ('prompt:cancel', lambda prompt: STATUS_CANCEL),
... ])
```

registry

1.3 prompt.caret module

Caret module.

class prompt.caret.**Caret**(*context*)

Bases: object

Caret (cursor) class which indicate the cursor locus in a prompt.

Note: This class defines __slots__ attribute so sub-class must override the attribute to extend available attributes.

---

context

Context

The prompt.context.Context instance.

context

get_backward_text()

A backward text from the caret.

Returns A backward part of the text from the caret

Return type str

Examples

```
>>> from .context import Context
>>> context = Context()
>>> # Caret:                 |
>>> context.text = '    Hello world!'
>>> context.caret_locus = 8
>>> caret = Caret(context)
>>> caret.get_backward_text()
'    Hell'
```

get_forward_text()

A forward text from the caret.

Returns A forward part of the text from the caret

Return type str

Examples

```
>>> from .context import Context
>>> context = Context()
>>> # Caret:                 |
>>> context.text = '    Hello world!'
```

```
>>> context.caret_locus = 8
>>> caret = Caret(context)
>>> caret.get_forward_text()
' world!'
```

get_selected_text()
A selected text under the caret.

Returns A part of the text under the caret

Return type str

Examples

```
>>> from .context import Context
>>> context = Context()
>>> # Caret:                |
>>> context.text = '    Hello world!'
>>> context.caret_locus = 8
>>> caret = Caret(context)
>>> caret.get_selected_text()
'o'
```

head
int: Readonly head locus index of the caret in the prompt.

Example

```
>>> from .context import Context
>>> context = Context()
>>> context.text = "Hello"
>>> caret = Caret(context)
>>> caret.head
0
```

lead
int: Readonly lead locus index of the caret in the prompt.

The lead indicate a minimum index for a first printable character.

Examples

For example, the `lead` become `4` in the following case.

```
>>> from .context import Context
>>> context = Context()
>>> context.text = '    Hello world!'
>>> caret = Caret(context)
>>> caret.lead
4
```

locus
int: Read and write current locus index of the caret in the prompt.

When a value is smaller than the `head` attribute or larger than the `tail` attribute, the value is regualted to `head` or `tail`.

Example

```
>>> from .context import Context
>>> context = Context()
>>> context.text = "Hello"
>>> caret = Caret(context)
>>> caret.locus
0
>>> caret.locus = 3
>>> caret.locus
3
>>> caret.locus = -1
>>> caret.locus
0
>>> caret.locus = 100    # beyond text length
>>> caret.locus
5
```

tail
int: Readonly tail locus index of the caret in the prompt.

Example

```
>>> from .context import Context
>>> context = Context()
>>> context.text = "Hello"
>>> caret = Caret(context)
>>> caret.tail
5
```

1.4 prompt.context module

Context module.

class prompt.context.**Context**
Bases: object

Context class which used to store/restore data.

Note: This class defines __slots__ attribute so sub-class must override the attribute to extend available attributes.

---

nvim
Nvim

The neovim.Nvim instance.

text
str

A user input text of the prompt.

caret_locus
int

A locus index of the caret in the prompt.

caret_locus

extend(*d*)

Extend a context instance from a dictionary.

Use `context.to_dict()` to create a corresponding dictionary. Keys which is not in `__slots__` will be ignored.

Parameters **d** (*dict*) – A dictionary.

Example

```
>>> context = Context.from_dict({
...     'text': 'Hello',
...     'caret_locus': 3,
... })
>>> context.text
'Hello'
>>> context.caret_locus
3
>>> context.extend({
...     'text': 'Bye',
...     'caret_locus': 1,
... })
>>> context.text
'Bye'
>>> context.caret_locus
1
```

classmethod from_dict(*d*)

Create a new context instance from a dictionary.

Use `context.to_dict()` to create a corresponding dictionary.

Parameters **d** (*dict*) – A corresponding dictionary.

Example

```
>>> context = Context.from_dict({
...     'text': 'Hello',
...     'caret_locus': 3,
... })
>>> context.text
'Hello'
>>> context.caret_locus
3
```

Returns A context instance.

Return type *Context*

text

to_dict()

Convert a context instance into a dictionary.

Use `Context.from_dict(d)` to restore a context instance from a dictionary.

Example

```
>>> context = Context()
>>> context.text = 'Hello'
>>> context.caret_locus = 3
>>> d = context.to_dict()
>>> d['text']
'Hello'
>>> d['caret_locus']
3
```

Returns A context dictionary.

Return type dict

1.5 prompt.digraph module

Digraph module.

class prompt.digraph.**Digraph**
Bases: object

A digraph registry class.

find(*nvim*, *char1*, *char2*)
Find a digraph of char1/char2.

registry

retrieve(*nvim*)
Retrieve char1/char2 and return a corresponding digraph.

1.6 prompt.history module

Command-line history module.

class prompt.history.**History**(*prompt*)
Bases: object

History class which manage a Vim's command-line history for input.

current()
Current command-line history value of input.

Returns A current command-line history value of input which an internal index points to. It returns a cached value when the internal index points to 0.

Return type str

next()
Get next command-line history value of input.

It decreases an internal index and points to a next command-line history value.

Returns A next command-line history value of input.

Return type str

next_match()
Get next matched command-line history value of input.

The initial query text is a text before the cursor when an internal index was 0 (like a cached value but only before the cursor.) It decreases an internal index until a next command-line history value matches to an initial query text and points to the matched next command-line history value. This behaviour is to mimic a Vim's builtin command-line history behaviour.

Returns A matched next command-line history value of input.

Return type str

nvim
A `neovim.Nvim` instance.

previous()
Get previous command-line history value of input.

It increases an internal index and points to a previous command-line history value.

Note that it cahces a `prompt.text` when the internal index was 0 (an initial value) and the cached value is used when the internal index points to 0. This behaviour is to mimic a Vim's builtin command-line history behaviour.

Returns A previous command-line history value of input.

Return type str

previous_match()
Get previous matched command-line history value of input.

The initial query text is a text before the cursor when an internal index was 0 (like a cached value but only before the cursor.) It increases an internal index until a previous command-line history value matches to an initial query text and points to the matched previous history value. This behaviour is to mimic a Vim's builtin command-line history behaviour.

Returns A matched previous command-line history value of input.

Return type str

prompt

1.7 prompt.key module

Key module.

class `prompt.key.`**Key**
Bases: *prompt.key.KeyBase*

Key class which indicate a single key.

code
int or bytes

A code of the key. A bytes is used when the key is a special key in Vim (a key which starts from 0x80 in getchar()).

char
str

A printable represantation of the key. It might be an empty string when the key is not printable.

classmethod **parse** (*nvim*, *expr*)
Parse a key expression and return a Key instance.

It returns a Key instance of a key expression. The instance is cached to individual expression so that the instance is exactly equal when same expression is spcified.

Parameters **expr** (*int, bytes, or str*) – A key expression.

Example

```
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> Key.parse(nvim, ord('a'))
Key(code=97, char='a')
>>> Key.parse(nvim, '<Insert>')
Key(code=b'kI', char='')
```

Returns A Key instance.

Return type *Key*

classmethod **represent** (*nvim*, *code*)
Return a string representation of a Keycode.

class prompt.key.**KeyBase** (*code*, *char*)
Bases: tuple

__getnewargs__ ()
Return self as a plain tuple. Used by copy and pickle.

__getstate__ ()
Exclude the OrderedDict from pickling

static __new__ (*_cls*, *code*, *char*)
Create new instance of KeyBase(code, char)

__repr__ ()
Return a nicely formatted representation string

char
Alias for field number 1

code
Alias for field number 0

1.8 prompt.keymap module

Keymap.

class prompt.keymap.**Definition**
Bases: *prompt.keymap.DefinitionBase*

An individual keymap definition.

classmethod **parse** (*nvim*, *rule*)
Parse a rule (list) and return a definition instance.

class prompt.keymap.**DefinitionBase**(*lhs*, *rhs*, *noremap*, *nowait*, *expr*)

Bases: tuple

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

__getstate__()

Exclude the OrderedDict from pickling

static __new__(*_cls*, *lhs*, *rhs*, *noremap*, *nowait*, *expr*)

Create new instance of DefinitionBase(lhs, rhs, noremap, nowait, expr)

__repr__()

Return a nicely formatted representation string

expr

Alias for field number 4

lhs

Alias for field number 0

noremap

Alias for field number 2

nowait

Alias for field number 3

rhs

Alias for field number 1

class prompt.keymap.**Keymap**

Bases: object

Keymap.

filter(*lhs*)

Filter keymaps by lhs Keystroke and return a sorted candidates.

Parameters **lhs** (Keystroke) – A left hand side Keystroke instance.

Example

```
>>> from .keystroke import Keystroke
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> k = lambda x: Keystroke.parse(nvim, x)
>>> keymap = Keymap()
>>> keymap.register_from_rules(nvim, [
...     ('<C-A><C-A>', '<prompt:A>'),
...     ('<C-A><C-B>', '<prompt:B>'),
...     ('<C-B><C-A>', '<prompt:C>'),
... ])
>>> candidates = keymap.filter(k(''))
>>> len(candidates)
3
>>> candidates[0]
Definition(..., rhs=(Key(code=b'<prompt:A>', ...)
>>> candidates[1]
Definition(..., rhs=(Key(code=b'<prompt:B>', ...)
>>> candidates[2]
```

```
        Definition(..., rhs=(Key(code=b'<prompt:C>', ...)
        >>> candidates = keymap.filter(k('<C-A>'))
        >>> len(candidates)
        2
        >>> candidates[0]
        Definition(..., rhs=(Key(code=b'<prompt:A>', ...)
        >>> candidates[1]
        Definition(..., rhs=(Key(code=b'<prompt:B>', ...)
        >>> candidates = keymap.filter(k('<C-A><C-A>'))
        >>> len(candidates)
        1
        >>> candidates[0]
        Definition(..., rhs=(Key(code=b'<prompt:A>', ...)
```

Returns Sorted Definition instances which starts from *lhs* Keystroke instance

Return type Iterator[Definition]

classmethod **from_rules**(*nvim*, *rules*)

Create a keymap instance from a rule tuple.

Parameters

- **nvim** (*neovim.Nvim*) – A `neovim.Nvim` instance.
- **rules** (*tuple*) – A tuple of rules.

Example

```
        >>> from .keystroke import Keystroke
        >>> from unittest.mock import MagicMock
        >>> nvim = MagicMock()
        >>> nvim.options = {'encoding': 'utf-8'}
        >>> lhs1 = Keystroke.parse(nvim, '<C-H>')
        >>> lhs2 = Keystroke.parse(nvim, '<C-D>')
        >>> lhs3 = Keystroke.parse(nvim, '<C-M>')
        >>> rhs1 = Keystroke.parse(nvim, '<BS>')
        >>> rhs2 = Keystroke.parse(nvim, '<DEL>')
        >>> rhs3 = Keystroke.parse(nvim, '<CR>')
        >>> keymap = Keymap.from_rules(nvim, [
        ...     (lhs1, rhs1),
        ...     (lhs2, rhs2, 'noremap'),
        ...     (lhs3, rhs3, 'nowait'),
        ... ])
```

Returns A keymap instance

Return type *Keymap*

harvest(*nvim*, *timeoutlen*)

Harvest a keystroke from getchar in Vim and return resolved.

It reads 'timeout' and 'timeoutlen' options in Vim and harvest a keystroke as Vim does. For example, if there is a key mapping for <C-X><C-F>, it waits 'timeoutlen' milliseconds after user hit <C-X>. If user continue <C-F> within timeout, it returns <C-X><C-F>. Otherwise it returns <C-X> before user continue <C-F>. If 'timeout' options is 0, it wait the next hit forever.

Note that it returns a key immediately if the key is not a part of the registered mappings.

Parameters **nvim** (*neovim.Nvim*) – A `neovim.Nvim` instance.

Returns A resolved keystroke.

Return type *Keystroke*

register(*definition*)

Register a keymap.

Parameters **definition** (`Definition`) – A definition instance.

Example

```
>>> from .keystroke import Keystroke
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> keymap = Keymap()
>>> keymap.register(Definition(
...     Keystroke.parse(nvim, '<C-H>'),
...     Keystroke.parse(nvim, '<BS>'),
... ))
>>> keymap.register(Definition(
...     Keystroke.parse(nvim, '<C-H>'),
...     Keystroke.parse(nvim, '<BS>'),
...     noremap=True,
... ))
>>> keymap.register(Definition(
...     Keystroke.parse(nvim, '<C-H>'),
...     Keystroke.parse(nvim, '<BS>'),
...     nowait=True,
... ))
>>> keymap.register(Definition(
...     Keystroke.parse(nvim, '<C-H>'),
...     Keystroke.parse(nvim, '<BS>'),
...     noremap=True,
...     nowait=True,
... ))
```

register_from_rule(*nvim*, *rule*)

Register a keymap from a rule.

Parameters

- **nvim** (*neovim.Nvim*) – A `neovim.Nvim` instance.
- **rule** (*tuple*) – A rule tuple.

Example

```
>>> from .keystroke import Keystroke
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> keymap = Keymap()
>>> keymap.register_from_rule(nvim, ['<C-H>', '<BS>'])
>>> keymap.register_from_rule(nvim, [
...     '<C-H>',
```

```
...     '<BS>',
...     'noremap',
... ])
>>> keymap.register_from_rule(nvim, [
...     '<C-H>',
...     '<BS>',
...     'noremap nowait',
... ])
```

register_from_rules(*nvim*, *rules*)

Register keymaps from raw rule tuple.

Parameters

- **nvim** (*neovim.Nvim*) – A `neovim.Nvim` instance.
- **rules** (*tuple*) – A tuple of rules.

Example

```
>>> from .keystroke import Keystroke
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> lhs1 = Keystroke.parse(nvim, '<C-H>')
>>> lhs2 = Keystroke.parse(nvim, '<C-D>')
>>> lhs3 = Keystroke.parse(nvim, '<C-M>')
>>> rhs1 = Keystroke.parse(nvim, '<BS>')
>>> rhs2 = Keystroke.parse(nvim, '<DEL>')
>>> rhs3 = Keystroke.parse(nvim, '<CR>')
>>> keymap = Keymap()
>>> keymap.register_from_rules(nvim, [
...     (lhs1, rhs1),
...     (lhs2, rhs2, 'noremap'),
...     (lhs3, rhs3, 'nowait'),
... ])
```

registry

resolve(*nvim*, *lhs*, *nowait=False*)

Resolve `lhs` Keystroke instance and return resolved keystroke.

Parameters

- **nvim** (*neovim.Nvim*) – A `neovim.Nvim` instance.
- **lhs** (*Keystroke*) – A left hand side Keystroke instance.
- **nowait** (*bool*) – Return a first exact matched keystroke even there are multiple keystroke instances are matched.

Example

```
>>> from .keystroke import Keystroke
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> k = lambda x: Keystroke.parse(nvim, x)
```

```
>>> keymap = Keymap()
>>> keymap.register_from_rules(nvim, [
...     ('<C-A><C-A>', '<prompt:A>'),
...     ('<C-A><C-B>', '<prompt:B>'),
...     ('<C-B><C-A>', '<C-A><C-A>', ''),
...     ('<C-B><C-B>', '<C-A><C-B>', 'noremap'),
...     ('<C-C>', '<prompt:C>', ''),
...     ('<C-C><C-A>', '<prompt:C1>'),
...     ('<C-C><C-B>', '<prompt:C2>'),
...     ('<C-D>', '<prompt:D>', 'nowait'),
...     ('<C-D><C-A>', '<prompt:D1>'),
...     ('<C-D><C-B>', '<prompt:D2>'),
... ])
>>> # No mapping starts from <C-C> so <C-C> is returned
>>> keymap.resolve(nvim, k('<C-Z>'))
(Key(code=26, ...),)
>>> # No single keystroke is resolved in the following case so None
>>> # will be returned.
>>> keymap.resolve(nvim, k('')) is None
True
>>> keymap.resolve(nvim, k('<C-A>')) is None
True
>>> # A single keystroke is resolved so rhs is returned.
>>> # will be returned.
>>> keymap.resolve(nvim, k('<C-A><C-A>'))
(Key(code=b'<prompt:A>', ...),)
>>> keymap.resolve(nvim, k('<C-A><C-B>'))
(Key(code=b'<prompt:B>', ...),)
>>> # noremap = False so recursively resolved
>>> keymap.resolve(nvim, k('<C-B><C-A>'))
(Key(code=b'<prompt:A>', ...),)
>>> # noremap = True so resolved only once
>>> keymap.resolve(nvim, k('<C-B><C-B>'))
(Key(code=1, ...), Key(code=2, ...))
>>> # nowait = False so no single keystroke could be resolved.
>>> keymap.resolve(nvim, k('<C-C>')) is None
True
>>> # nowait = True so the first matched candidate is returned.
>>> keymap.resolve(nvim, k('<C-D>'))
(Key(code=b'<prompt:D>', ...),)
```

Returns None if no single keystroke instance is resolved. Otherwise return a resolved keystroke instance or `lhs` itself if no mapping is available for `lhs` keystroke.

Return type None or Keystroke

1.9 prompt.keystroke module

Keystroke module.

class `prompt.keystroke.`**`Keystroke`**

Bases: `tuple`

Keystroke class which indicate multiple keys.

classmethod **`parse`**(*nvim*, *expr*)

Parse a keystroke expression and return a Keystroke instance.

Parameters

- **nvim** (*neovim.Nvim*) – A `neovim.Nvim` instance.
- **expr** (*tuple, bytes, str*) – A keystroke expression.

Example

```
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> Keystroke.parse(nvim, 'abc')
(Key(code=97, ...), Key(code=98, ...), Key(code=99, ...))
>>> Keystroke.parse(nvim, '<Insert>')
(Key(code=b'kI', char=''),)
```

Returns A Keystroke instance.

Return type *Keystroke*

startswith(*other*)
Check if the keystroke starts from `other`.

Parameters **other** (`Keystroke`) – A keystroke instance will be checked.

Returns True if the keystroke starts from `other`.

Return type bool

1.10 prompt.prompt module

Prompt module.

class `prompt.prompt.`**Prompt**(*nvim*, *context*)
Bases: `object`

Prompt class.

apply_custom_mappings_from_vim_variable(*varname*)
Apply custom key mappings from Vim variable.

Parameters **varname** (*str*) – A global Vim's variable name

insert_text(*text*)
Insert text after the caret.

Parameters **text** (*str*) – A text which will be inserted after the caret.

Example

```
>>> from .context import Context
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> context = Context()
>>> context.text = "Hello Goodbye"
>>> context.caret_locus = 3
```

```
>>> prompt = Prompt(nvim, context)
>>> prompt.insert_text('AA')
>>> prompt.text
'HelAAlo Goodbye'
```

on_init(*default*)
Initialize the prompt.

It calls 'inputsave' function in Vim and assign `default` text to the `self.text` to initialize the prompt text in default.

Parameters **default** (*None or str*) – A default text of the prompt. If omitted, a text in the context specified in the constructor is used.

Returns The return value will be used as a status of the prompt mainloop, indicating that if return value is not STATUS_PROGRESS, the prompt mainloop immediately terminated. Returning None is equal to returning STATUS_PROGRESS.

Return type None or int

on_keypress(*keystroke*)
Handle a pressed keystroke and return the status.

It is used to handle a pressed keystroke. Note that subclass should NOT override this method to perform actions. Register a new custom action instead. In default, it call action and return the result if the keystroke is <xxx:xxx>or call Vim function XXX and return the result if the keystroke is <call:XXX>.

Parameters **keystroke** (Keystroke) – A pressed keystroke instance. Note that this instance is a reslved keystroke instace by keymap.

Returns The return value will be used as a status of the prompt mainloop, indicating that if return value is not STATUS_PROGRESS, the prompt mainloop immediately terminated. Returning None is equal to returning STATUS_PROGRESS.

Return type None or int

on_redraw()
Redraw the prompt.

It is used to redraw the prompt. In default, it echos specified prefix the caret, and input text.

on_term(*status*)
Finalize the prompt.

It calls 'inputrestore' function in Vim to finalize the prompt in default. The return value is used as a return value of the prompt.

Parameters **status** (*int*) – A prompt status.

Returns A status which is used as a result value of the prompt.

Return type int

on_update(*status*)
Update the prompt status and return the status.

It is used to update the prompt status. In default, it does nothing and return the specified `status` directly.

Parameters **status** (*int*) – A prompt status which is updated by previous on_keypress call.

Returns The return value will be used as a status of the prompt mainloop, indicating that if return value is not STATUS_PROGRESS, the prompt mainloop immediately terminated. Returning None is equal to returning STATUS_PROGRESS.

Return type None or int

prefix = ''

redraw_prompt()

replace_text(*text*)

Replace text after the caret.

Parameters **text** (*str*) – A text which will be replaced after the caret.

Example

```
>>> from .context import Context
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> context = Context()
>>> context.text = "Hello Goodbye"
>>> context.caret_locus = 3
>>> prompt = Prompt(nvim, context)
>>> prompt.replace_text('AA')
>>> prompt.text
'HelAA Goodbye'
```

start(*default=None*)

Start prompt with `default` text and return value.

Parameters **default** (*None or str*) – A default text of the prompt. If omitted, a text in the context specified in the constructor is used.

Returns The status of the prompt.

Return type int

text

str: A current context text.

It automatically adjust the current caret locus to the tail of the text if any text is assigned.

It calls the following overridable methods in order of the appearance.

- on_init - Only once
- on_update
- on_redraw
- on_keypress
- on_term - Only once

Example

```
>>> from .context import Context
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> context = Context()
>>> context.text = "Hello"
>>> context.caret_locus = 3
```

```
>>> prompt = Prompt(nvim, context)
>>> prompt.text
'Hello'
>>> prompt.caret.locus
3
>>> prompt.text = "FooFooFoo"
>>> prompt.text
'FooFooFoo'
>>> prompt.caret.locus
9
```

update_text(*text*)
Insert or replace text after the caret.

Parameters **text** (*str*) – A text which will be replaced after the caret.

Example

```
>>> from .context import Context
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> context = Context()
>>> context.text = "Hello Goodbye"
>>> context.caret_locus = 3
>>> prompt = Prompt(nvim, context)
>>> prompt.insert_mode = INSERT_MODE_INSERT
>>> prompt.update_text('AA')
>>> prompt.text
'HelAAlo Goodbye'
>>> prompt.insert_mode = INSERT_MODE_REPLACE
>>> prompt.update_text('BB')
>>> prompt.text
'HelAABB Goodbye'
```

1.11 prompt.util module

Utility module.

class prompt.util.**Singleton**
Bases: type

A singleton metaclass.

instance = None

prompt.util.**ensure_bytes**(*nvim*, *seed*)
Encode *str* to *bytes* if necessary and return.

Parameters

- **nvim** (*neovim.Nvim*) – A neovim.Nvim instance.
- **seed** (*AnyStr*) – A str or bytes instance.

Example

```
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> ensure_bytes(nvim, b'a')
b'a'
>>> ensure_bytes(nvim, 'a')
b'a'
```

Returns A bytes represantation of `seed`.

Return type bytes

`prompt.util.`**`ensure_str`**(*nvim*, *seed*)

Decode *bytes* to *str* if necessary and return.

Parameters

- **nvim** (*neovim.Nvim*) – A `neovim.Nvim` instance.
- **seed** (*AnyStr*) – A str or bytes instance.

Example

```
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> ensure_str(nvim, b'a')
'a'
>>> ensure_str(nvim, 'a')
'a'
```

Returns A str represantation of `seed`.

Return type str

`prompt.util.`**`get_encoding`**(*nvim*)

Return a Vim's internal encoding.

The retrieve encoding is cached to the function instance while encoding options should not be changed in Vim's live session (see :h encoding) to enhance performance.

Parameters **nvim** (*neovim.Nvim*) – A `neovim.Nvim` instance.

Returns A Vim's internal encoding.

Return type str

`prompt.util.`**`getchar`**(*nvim*, **args*)

Call getchar and return int or bytes instance.

Parameters

- **nvim** (*neovim.Nvim*) – A `neovim.Nvim` instance.
- ***args** – Arguments passed to getchar function in Vim.

Returns A int or bytes.

Return type Union[int, bytes]

prompt.util.**int2char**(*nvim*, *code*)

Return a corresponding char of *code*.

It uses "nr2char()" in Vim script when 'encoding' option is not utf-8. Otherwise it uses "chr()" in Python to improve the performance.

Parameters

- **nvim** (*neovim.Nvim*) – A neovim.Nvim instance.
- **code** (*int*) – A int which represent a single character.

Example

```
>>> from unittest.mock import MagicMock
>>> nvim = MagicMock()
>>> nvim.options = {'encoding': 'utf-8'}
>>> int2char(nvim, 97)
'a'
```

Returns A str of code.

Return type str

prompt.util.**int2repr**(*nvim*, *code*)

prompt.util.**safeget**(*l*, *index*, *default=None*)

Return an index item of list or default.

1.12 Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

p

Symbols

A

C

D

E

F

G

H

I

K

L

N