tea Documentation

Release 0.0.5

Viktor Kerkez <alefnula@gmail.com>

Mar 10, 2017

Contents

Contents:

CHAPTER 1

API Documentation

tea api documentation

`tea.console` Module

`tea.console.color.`**`set_color`**(*fg='normal'*, *bg='normal'*, *fg_dark=False*, *bg_dark=False*, *underlined=False*)

Set the console color.

```
>>> set_color(Color.red, Color.blue)
>>> set_color('red', 'blue')
>>> set_color() # returns back to normal
```

`tea.console.color.`**`strip_colors`**(*text*)

Helper function used to strip out the color tags so other function can determine the real text line lengths.

Parameters **`text`** (*`str`*) – Text to strip color tags from

Return type `str`

Returns Stripped text.

`tea.console.color.`**`cprint`**(*text*, *fg='normal'*, *bg='normal'*, *fg_dark=False*, *bg_dark=False*, *underlined=False*, *parse=False*)

Prints string in to stdout using colored font.

See L{set_color} for more details about colors.

Parameters

- **`color`** (*`str or list[str]`*) – If color is a string than this is a color in which the text will appear. If color is a list of strings than all desired colors will be used as a mask (this is used when you wan to set foreground and background color).
- **`text`** (*`str`*) – Text that needs to be printed.

tea.console.color.**colorize_output**(*output*, *colors*, *indent=0*)
Prints output to console using provided color mappings.

Color mapping is dict with regular expressions as key and tuple of two as values. Key is used to match if line should be colorized and tuple contains color to be used and boolean value that indicates if dark foreground is used. For example:

```
>>> CLS = {
>>>     re.compile(r'^(--- .*)$'): (Color.red, False)
>>> }
```

will colorize lines that start with '—' to red.

If different parts of line needs to be in different color then dict must be supplied in colors with keys that are named group from regular expression and values that are tuples of color and boolean that indicates if dark foreground is used. For example:

```
>>> CLS = {
>>>     re.compile(r'^(?P<key>user:\s+)(?P<user>.*)$'): {
>>>         'key': (Color.yellow, True),
>>>         'user': (Color.cyan,   False)
>>>     }
>>> }
```

will colorize line 'user: Some user' so that 'user:' part is yellow with dark foreground and 'Some user' part is cyan without dark foreground.

tea.console.format.**format_page**(*text*)
Formats the text for output adding ASCII frame around the text.

Parameters **text** (*str*) – Text that needs to be formatted.

Return type string

Returns Formatted string.

tea.console.format.**table**(*text*)
Formats the text as a table

Text in format:

first | second row 2 col 1 | 4

Will be formatted as:

```
+-------------+--------+
| first       | second |
+-------------+--------+
| row 2 col 1 | 4      |
+-------------+--------+
```

Parameters **text** (*str*) – Text that needs to be formatted.

Return type str

Returns Formatted string.

tea.console.format.**hbar**(*width*)
Returns ASCII HBar +---+ with the specified width.

Parameters **width** (*int*) – Width of the central part of the bar.

Return type str

Returns ASCII HBar.

tea.console.format.**print_page**(*text*)
Formats the text and prints it on stdout.

Text is formatted by adding a ASCII frame around it and coloring the text. Colors can be added to text using color tags, for example:

My [FG_BLUE]blue[NORMAL] text. My [BG_BLUE]blue background[NORMAL] text.

tea.console.format.**wrap_text**(*text*, *width=80*)
Wraps text lines to maximum *width* characters.

Wrapped text is aligned against the left text border.

Parameters

- **text** (*str*) – Text to wrap.
- **width** (*int*) – Maximum number of characters per line.

Return type str

Returns Wrapped text.

tea.console.format.**rjust_text**(*text*, *width=80*, *indent=0*, *subsequent=None*)
Same as L{wrap_text} with the difference that the text is aligned against the right text border.

Parameters

- **text** (*str*) – Text to wrap and align.
- **width** (*int*) – Maximum number of characters per line.
- **indent** (*int*) – Indentation of the first line.
- **subsequent** (*int or None*) – Indentation of all other lines, if it is None, then the indentation will be same as for the first line.

tea.console.format.**center_text**(*text*, *width=80*)
Center all lines of the text.

It is assumed that all lines width is smaller then B{width}, because the line width will not be checked.

Parameters

- **text** (*str*) – Text to wrap.
- **width** (*int*) – Maximum number of characters per line.

Return type str

Returns Centered text.

tea.console.utils.**clear_screen**(*numlines=100*)
Clear the console.

Parameters **numlines** (*int*) – This is an optional argument used only as a fall-back if the operating system console doesn't have clear screen function.

Return type None

tea.console.utils.**getch**()
Cross-platform getch() function.

Same as the getch function from msvcrt library, but works on all platforms. :rtype: str :return: One character got from standard input.

tea.ctx Module

Context library - providing usefull context managers

tea.ctx.**suppress**(**args*, ***kwds*)
Ignores an exception or exception list

Usage:

```
with suppress(OSError):
    os.remove('filename.txt')
```

tea.ds Module

Data structures module

class tea.ds.config.**Config**(*filename=None*, *data=None*, *fmt=None*, *encoding=u'utf-8'*, *auto_save=True*)
Configuration class

keys(**args*, ***kwargs*)
Returns a set of top level keys in this configuration

get(**args*, ***kwargs*)
Safe version which always returns a default value

delete(**args*, ***kwargs*)
Safe version, never, raises an error

insert(**args*, ***kwargs*)
Inserts at the index, and if the index is not provided appends to the end of the list

class tea.ds.config.**MultiConfig**(*filename=None*, *data=None*, *fmt=None*, *encoding=u'utf-8'*, *auto_save=True*)
Base class for configuration management

keys(**args*, ***kwargs*)
Returns a merged set of top level keys from all the configuration files

get(**args*, ***kwargs*)
Safe version always returns a default value

delete(**args*, ***kwargs*)
Safe version, never raises an error

tea.logger Module

Logging module

This logging module is designed as a wrapper around the python logging module.

When the module is loaded it configure the logging object with some default parameters (log to stderr with DEBUG level). After that, user can call the L{configure_logger} function and configure logging to file and stderr.

Usage

```
>>> import logging
>>> import tempfile
>>> from tea.logger import configure_logging
>>> configure_logging(filename=tempfile.mktemp())
>>> logger = logging.getLogger('test')
>>> logger.debug('Debug level log entry')
>>> logger.info('Info level log entry')
>>> logger.warning('Warn level log entry')
WARNING     - Warn level log entry [test:1]
>>> logger.error('Error level log entry')
ERROR       - Error level log entry [test:1]
>>> logger.critical('Critical level log entry')
CRITICAL    - Critical level log entry [test:1]
>>> try:
...     raise Exception('Test exception')
... except:
...     logger.exception('Error level log entry with stack trace')
ERROR       - Error level log entry with stack trace [test:4]
Traceback (most recent call last):
  ...
Exception: Test exception
```

tea.logger.log.**configure_logging**(*filename=None*, *filemode='a'*, *datefmt='%Y.%m.%d %H:%M:%S'*, *fmt='%(asctime)s.%(msecs)03d %(levelname)11s: %(message)s [%(name)s:%(lineno)d]'*, *stdout_fmt='%(levelname)-11s - %(message)s [%(name)s:%(lineno)d]'*, *level=10*, *stdout_level=30*, *initial_file_message=''*, *max_size=1048576*, *rotations_number=5*, *remove_handlers=True*)

Configure logging module.

Parameters

- **filename** (*str*) – Specifies a filename to log to.
- **filemode** (*str*) – Specifies the mode to open the log file. Values: `'a'`, `'w'`. *Default:* `a`
- **datefmt** (*str*) – Use the specified date/time format.
- **fmt** (*str*) – Format string for the file handler.
- **stdout_fmt** (*str*) – Format string for the stdout handler.
- **level** (*int*) – Log level for the file handler. Log levels are the same as the log levels from the standard `logging` module. *Default:* `logging.DEBUG`
- **stdout_level** (*int*) – Log level for the stdout handler. Log levels are the same as the log levels from the standard `logging` module. *Default:* `logging.WARNING`
- **initial_file_message** (*str*) – First log entry written in file.
- **max_size** (*int*) – Maximal size of the logfile. If the size of the file exceed the maximal size it will be rotated.
- **rotations_number** (*int*) – Number of rotations to save
- **remove_handlers** (*bool*) – Remove all existing handlers

Return type None

tea.msg Module

Simple and complete library for sending emails

class tea.msg.mail.**SMTPConnection**(*host=None*, *port=None*, *username=None*, *password=None*, *use_tls=None*, *fail_silently=False*)

A wrapper that manages the SMTP network connection.

open()

Ensures we have a connection to the email server. Returns whether or not a new connection was required (True or False).

close()

Closes the connection to the email server.

send_messages(*messages*)

Sends one or more EmailMessage objects and returns the number of email messages sent.

class tea.msg.mail.**EmailMessage**(*subject=''*, *body=''*, *sender=None*, *to=None*, *cc=None*, *bcc=None*, *attachments=None*, *headers=None*, *connection=None*)

A container for email information.

Initialize a single email message (which can be sent to multiple recipients).

All strings used to create the message can be unicode strings (or UTF-8 bytestrings). The SafeMIMEText class will handle any necessary encoding conversions.

recipients()

Returns a list of all recipients of the email (includes direct addressees as well as Bcc entries).

send(*fail_silently=False*)

Sends the email message.

attach(*filename=None*, *content=None*, *mimetype=None*)

Attaches a file with the given filename and content. The filename can be omitted (useful for multipart/alternative messages) and the mimetype is guessed, if not provided.

If the first parameter is a MIMEBase subclass it is inserted directly into the resulting message attachments.

attach_file(*path*, *mimetype=None*)

Attaches a file from the filesystem.

class tea.msg.mail.**EmailMultiAlternatives**(*subject=''*, *body=''*, *sender=None*, *to=None*, *cc=None*, *bcc=None*, *attachments=None*, *headers=None*, *connection=None*)

A version of EmailMessage that makes it easy to send multipart/alternative messages. For example, including text and HTML versions of the text is made easier.

Initialize a single email message (which can be sent to multiple recipients).

All strings used to create the message can be unicode strings (or UTF-8 bytestrings). The SafeMIMEText class will handle any necessary encoding conversions.

attach_alternative(*content*, *mimetype=None*)

Attach an alternative content representation.

tea.msg.mail.**send_mail**(*subject*, *sender*, *to*, *message*, *html_message=None*, *cc=None*, *bcc=None*, *attachments=None*, *host=None*, *port=None*, *auth_user=None*, *auth_password=None*, *use_tls=False*, *fail_silently=False*)

Easy wrapper for sending a single message to a recipient list. All members of the recipient list will see the other recipients in the 'To' field.

Note: The API for this method is frozen. New code wanting to extend the functionality should use the EmailMessage class directly.

tea.msg.mail.**send_mass_mail**(*datatuple*, *fail_silently=False*, *auth_user=None*, *auth_password=None*)

Given a datatuple of (subject, message, sender, recipient_list), sends each message to each recipient list. Returns the number of e-mails sent.

If auth_user and auth_password are set, they're used to log in.

Note: The API for this method is frozen. New code wanting to extend the functionality should use the EmailMessage class directly.

tea.process Module

tea.process.**execute**(*command*, **args*, ***kwargs*)

Execute a command with arguments and wait for output. Arguments should not be quoted!

Keyword arguments:

Parameters

- **env** (*dict*) – Dictionary of additional environment variables.
- **wait** (*bool*) – Wait for the process to finish.

Example:

```
>>> code = 'import sys;sys.stdout.write('out');sys.exit(0)'
>>> status, out, err = execute('python', '-c', code)
>>> print('status: %s, output: %s, error: %s' % (status, out, err))
status: 0, output: out, error:
>>> code = 'import sys;sys.stderr.write('out');sys.exit(1)'
>>> status, out, err = execute('python', '-c', code)
>>> print('status: %s, output: %s, error: %s' % (status, out, err))
status: 1, output: , error: err
```

tea.process.**execute_and_report**(*command*, **args*, ***kwargs*)

Executes a command with arguments and wait for output. If execution was successful function will return True, if not, it will log the output using standard logging and return False.

tea.process.**get_processes**(*sort_by_name=True*, *cmdline=False*)

Retrieves a list of processes sorted by name.

Parameters

- **sort_by_name** (*bool*) – Sort the list by name or by process ID's
- **cmdline** (*bool*) – Add process command line to output

Return type list[(int, str)] or list[(int, str, str)]

Returns List of process id, process name and optional cmdline tuples

tea.process.**find**(*name*, *arg=None*)

Find process by name or by argument in command line if arg param is available.

Parameters

- **name** (*str*) – process name to search for
- **arg** (*str*) – command line argument for a process to search for

Return type (int, str)

Returns A tuple of process id, process name

tea.process.**kill**(*pid*)
Kills a process by it's process ID.

Parameters **pid** (*int*) – Process ID of the process to kill.

class tea.process.base.**Process**(*command*, *arguments=None*, *env=None*, *redirect_output=True*, *working_dir=None*)
Abstract base class for the Process class that is implemented for every platform in it's own module.

Simple example of Process class usage can be:

```
>>> from tea.process import Process
>>> p = Process('python', ['-c', 'import time;time.sleep(5);print 3'])
>>> p.start()
>>> p.is_running
True
>>> p.wait()
True
>>> p.read()
'3\n'
>>> p.eread()
''
```

Creates the Process object providing the command and it's command line arguments.

The only required parameter is the command to execute. It's important to note that the constructor only initializes the class, it doesn't executes the process. To actually execute the process you have to call **:met:'start'**.

Parameters

- **command** (*str*) – Path to the executable file.
- **arguments** (*list*) – list of command line arguments passed to the command
- **env** (*dict*) – Optional additional environment variables that will be added to the subprocess environment or that override currently set environment variables.
- **redirect_output** (*bool*) – True if you want to be able to get the standard output and the standard error of the subprocess, otherwise it will be redirected to /dev/null
- **working_dir** (*str*) – Set the working directory from which the process will be started.

start()
Starts the process.

kill()
Kills the process if it's running.

wait(*timeout=None*)
Waits for the process to finish.

It will wait for the process to finish running. If the timeout is provided, the function will wait only `timeout` amount of seconds and then return to it's caller.

Parameters **timeout** (*None or int*) – None if you want to wait to wait until the process actually finishes, otherwise it will wait just the `timeout` number of seconds.

Return type bool

Returns Return value only makes sense if you provided the timeout parameter. It will indicate if the process actually finished in the amount of time specified, i.e. if the we specify 3 seconds

and the process actually stopped after 3 seconds it will return `True` otherwise it will return `False`.

is_running
Property that indicates if the process is still running.

Return type bool

Returns True if the process is still running False otherwise

pid
Property that returns the PID of the process if it is running.

Return type int

Returns process id of the running process

exit_code
Property that returns the exit code if the process has finished running.

Return type int or None

Returns Exit code or None if the process is still running

write(*string*)
Write a string to the process standard input.

Parameters **string** (*str*) – String to write to the process standard input

read()
Read from the process standard output.

Return type str

Returns The data process has written to the standard output if it has written anything. If it hasn't or you already read all the data process wrote, it will return an empty string.

eread()
Read from the process standard error.

Return type str

Returns The data process has written to the standard error if it has written anything. If it hasn't or you already read all the data process wrote, it will return an empty string.

tea.shell Module

This module mimics some of the behaviors of the builtin `shutil` module, adding logging to all operations and abstracting some other useful shell commands (functions).

tea.shell.**split**(*s*, *posix=True*)
Split the string s using shell-like syntax

tea.shell.**search**(*path*, *matcher='*'*, *dirs=False*, *files=True*)
Recursive search function.

Parameters

- **path** – path to search recursively
- **matcher** – string pattern to search for or function that returns True/False for a file argument
- **dirs** – if True returns also directories that match the pattern

tea.shell.**chdir**(*directory*)
Change the current working directory

tea.shell.**goto**(**args*, ***kwds*)
Context object for changing directory.

Usage:

```
>>> with goto(directory) as ok:
...     if not ok:
...         print 'Error'
...     else:
...         print 'All OK'
```

tea.shell.**mkdir**(*path*[, *mode=0777*])
Create a leaf directory and all intermediate ones. Works like mkdir, except that any intermediate path segment (not just the rightmost) will be created if it does not exist. This is recursive.

Parameters

- **path** (*str*) – directory to create
- **mode** (*int*) – directory mode
- **delete** (*bool*) – delete directory/file if exists

Return type bool

Returns True if succeeded else False

tea.shell.**copy**(*source*, *destination*)
Copy file or directory

tea.shell.**gcopy**(*pattern*, *destination*)
Copy all file found by glob.glob(pattern) to destination directory

tea.shell.**move**(*source*, *destination*)
Recursively move a file or directory to another location.

If the destination is on our current file system, then simply use rename. Otherwise, copy source to the destination and then remove source.

Parameters

- **source** (*str*) – Source file or directory (file or directory to move).
- **destination** (*str*) – Destination file or directory (where to move).

Return type bool

Returns True if the operation is successful, False otherwise.

tea.shell.**gmove**(*pattern*, *destination*)
Move all file found by glob.glob(pattern) to destination directory

tea.shell.**remove**(*path*)
Delete a file or directory

Parameters **path** (*str*) – Path to the file or directory that needs to be deleted.

Return type bool

Returns True if the operation is successful, False otherwise.

tea.shell.**gremove**(*pattern*)
Remove all file found by glob.glob(pattern)

Parameters **pattern** (*str*) – Pattern of files to remove

tea.shell.**touch**(*path*, *content=''*, *encoding='utf-8'*)
Create a file at the given path if it does not already exists.

Parameters

- **path** (*str*) – Path to the file.
- **content** (*str*) – Optional content that will be written in the file.
- **encoding** (*str*) – Encoding in which to write the content. Default: `utf-8`

tea.system Module

tea.utils Module

tea.utils.**get_object**(*path=''*, *obj=None*)
Returns an object from a dot path.

Path can either be a full path, in which case the *get_object* function will try to import the module and follow the path. Or it can be a path relative to the object passed in as the second argument.

Example for full paths:

```
>>> get_object('os.path.join')
<function join at 0x1002d9ed8>
>>> get_object('tea.process')
<module 'tea.process' from 'tea/process/__init__.pyc'>
```

Example for relative paths when an object is passed in:

```
>>> import os
>>> get_object('path.join', os)
<function join at 0x1002d9ed8>
```

tea.utils.**load_subclasses**(*klass*, *modules=None*)
Load recursively all submodules of the modules and return all the subclasses of the provided class

Parameters

- **klass** – Class whose subclasses we want to load.
- **modules** (*str or list[str]*) – List of additional modules or module names that should be recursively imported in order to find all the subclasses of the desired class. Default: None

tea.utils.compress.**unzip**(*archive*, *destination*, *filenames=None*)
Unzip the a complete zip archive into destination directory, or unzip a specific file(s) from the archive.

Usage:

```
>>> output = os.path.join(os.getcwd(), 'output')
>>> # Archive can be an instance of a ZipFile class
>>> archive = zipfile.ZipFile('test.zip', 'r')
>>> # Or just a filename
>>> archive = 'test.zip'
>>> # Extracts all files
>>> unzip(archive, output)
>>> # Extract only one file
>>> unzip(archive, output, 'my_file.txt')
```

```
>>> # Extract a list of files
>>> unzip(archive, output, ['my_file1.txt', 'my_file2.txt'])
>>> unzip_file('test.zip', 'my_file.txt', output)
```

Parameters

- **archive** (zipfile.ZipFile or str) – Zipfile object to extract from or path to the zip archive.
- **destination** (*str*) – Path to the output directory
- **filenames** (*str, list or None*) – Path(s) to the filename(s) inside the zip archive that you want to extract.

tea.utils.compress.**mkzip**(*archive*, *items*, *mode='w'*, *save_full_paths=False*)
Recursively zip a directory

Parameters

- **archive** (zipfile.ZipFile or str) – ZipFile object add to or path to the output zip archive.
- **items** (*str or list*) – Single item or list of items (files and directories) to be added to zipfile
- **mode** (*str*) – w for create new and write a for append to
- **save_full_paths** (*bool*) – preserve full paths

tea.utils.compress.**seven_zip**(*archive*, *items*, *self_extracting=False*)
Create a 7z archive

tea.utils.compress.**seven_unzip**(*archive*, *output*)
Extract a 7z archive

tea.utils.crypto.**encrypt**(*data*, *digest=True*)
Performs encryption of provided data.

tea.utils.crypto.**decrypt**(*data*, *digest=True*)
Decrypts provided data.

class tea.utils.daemon.**Daemon**(*pidfile*, *stdin='/dev/null'*, *stdout='/dev/null'*, *stderr='/dev/null'*)
A generic daemon class.

Usage: subclass the Daemon class and override the run() method

daemonize()
Do the UNIX double-fork magic

See Stevens' "Advanced Programming in the UNIX Environment" for details (ISBN 0201563177) http://www.erlenstar.demon.co.uk/unix/faq_2.html#SEC16

start(**args*)
Start the daemon

stop()
Stop the daemon

restart(**args*)
Restart the daemon

run(**args*)
You should override this method when you subclass Daemon.

It will be called after the process has been daemonized by start() or restart().

tea.utils.encoding.**smart_text**(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Returns a unicode object representing 's'. Treats bytes using the 'encoding' codec.

If strings_only is True, don't convert (some) non-string-like objects.

tea.utils.encoding.**smart_bytes**(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Returns a bytes version of 's', encoded as specified in 'encoding'.

If strings_only is True, don't convert (some) non-string-like objects.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

R

S

T

U

W