osrf_pycommon Documentation

Release 0.0

William Woodall

February 29, 2016

Contents

`osrf_pycommon` is a python package which contains commonly used Python boilerplate code and patterns. Things like ansi terminal coloring, capturing colored output from programs using subprocess, or even a simple logging system which provides some nice functionality over the built-in Python logging system.

The functionality provided here should be generic enough to be reused in arbitrary scenarios and should avoid bringing in dependencies which are not part of the standard Python library. Where possible Windows and Linux/OS X should be supported, and where it cannot it should be gracefully degrading. Code should be pure Python as well as Python 2 and Python 3 bilingual.

Contents:

CHAPTER 1

The `cli_utils` Module

This module provides functions and patterns for creating Command Line Interface (CLI) tools.

1.1 Common CLI Functions

Commonly used, CLI related functions.

`osrf_pycommon.cli_utils.common.`**`extract_argument_group`**(*args*, *delimiting_option*)

Extract a group of arguments from a list of arguments using a delimiter.

Here is an example:

```
>>> extract_argument_group(['foo', '--args', 'bar', '--baz'], '--args')
(['foo'], ['bar', '--baz'])
```

The group can always be ended using the double hyphen `--`. In order to pass a double hyphen as arguments, use three hyphens `---`. Any set of hyphens encountered after the delimiter, and up to `--`, which have three or more hyphens and are isolated, will be captured and reduced by one hyphen.

For example:

```
>> extract_argument_group(['foo',
                           '--args', 'bar', '--baz', '---', '--',
                           '--foo-option'], '--args')
(['foo', '--foo-option'], ['bar', '--baz', '--'])
```

In the result the `--` comes from the `---` in the input. The `--args` and the corresponding `--` are removed entirely.

The delimiter and `--` terminator combination can also happen multiple times, in which case the bodies of arguments are combined and returned in the order they appeared.

For example:

```
>> extract_argument_group(['foo',
                           '--args', 'ping', '--',
                           'bar',
                           '--args', 'pong', '--',
                           'baz',
                           '--args', '--'], '--args')
(['foo', 'bar', 'baz'], ['ping', 'pong'])
```

Note: `--` cannot be used as the `delimiting_option`.

Parameters

- **args** (*list*) – list of strings which are ordered arguments.
- **delimiting_option** (*str*) – option which denotes where to split the args.

Returns tuple of arguments before and after the delimiter.

Return type tuple

Raises ValueError if the delimiting_option is `--`.

osrf_pycommon.cli_utils.common.**extract_jobs_flags**(*arguments*)

Extracts make job flags from a list of other make flags, i.e. -j8 -l8

The input arguments are given as a string separated by whitespace. Make job flags are matched and removed from the arguments, and the Make job flags and what is left over from the input arguments are returned.

If no job flags are encountered, then an empty string is returned as the first element of the returned tuple.

Examples:

```
>> extract_jobs_flags('-j8 -l8')
('-j8 -l8', '')
>> extract_jobs_flags('-j8 ')
('-j8', ' ')
>> extract_jobs_flags('target -j8 -l8 --some-option')
('-j8 -l8', 'target --some-option')
>> extract_jobs_flags('target --some-option')
('', 'target --some-option')
```

Parameters **arguments** (*str*) – string of space separated arguments which may or may not contain make job flags

Returns tuple of make jobs flags as a space separated string and leftover arguments as a space separated string

Return type tuple

1.2 The Verb Pattern

The verb pattern is a pattern where a single command aggregates multiple related commands by taking a required positional argument which is the "verb" for the action you want to perform. For example, `catkin build` is an example of a `command` and `verb` pair, where `catkin` is the command and `build` is the verb. In this example, the `catkin` command groups "actions" which are related to catkin together using verbs like `build` which will build a workspace of catkin packages.

1.2.1 Command Boilerplate

This is an example boilerplate of a command which will use verbs:

```
from __future__ import print_function

import argparse
import sys

from osrf_pycommon.cli_utils.verb_pattern import create_subparsers
from osrf_pycommon.cli_utils.verb_pattern import list_verbs
```

```
from osrf_pycommon.cli_utils.verb_pattern import split_arguments_by_verb

COMMAND_NAME = '<INSERT COMMAND NAME HERE>'

VERBS_ENTRY_POINT = '{0}.verbs'.format(COMMAND_NAME)


def main(sysargs=None):
    # Assign sysargs if not set
    sysargs = sys.argv[1:] if sysargs is None else sysargs

    # Create a top level parser
    parser = argparse.ArgumentParser(
        description="{0} command".format(COMMAND_NAME)
    )

    # Generate a list of verbs available
    verbs = list_verbs(VERBS_ENTRY_POINT)

    # Create the subparsers for each verb and collect the arg preprocessors
    argument_preprocessors, verb_subparsers = create_subparsers(
        parser,
        COMMAND_NAME,
        verbs,
        VERBS_ENTRY_POINT,
        sysargs,
    )

    # Determine the verb, splitting arguments into pre and post verb
    verb, pre_verb_args, post_verb_args = split_arguments_by_verb(sysargs)

    # Short circuit -h and --help
    if '-h' in pre_verb_args or '--help' in pre_verb_args:
        parser.print_help()
        sys.exit(0)

    # Error on no verb provided
    if verb is None:
        print(parser.format_usage())
        sys.exit("Error: No verb provided.")
    # Error on unknown verb provided
    if verb not in verbs:
        print(parser.format_usage())
        sys.exit("Error: Unknown verb '{0}' provided.".format(verb))

    # Short circuit -h and --help for verbs
    if '-h' in post_verb_args or '--help' in post_verb_args:
        verb_subparsers[verb].print_help()
        sys.exit(0)

    # First allow the verb's argument preprocessor to strip any args
    # and return any "extra" information it wants as a dict
    processed_post_verb_args, extras = \
        argument_preprocessors[verb](post_verb_args)
    # Then allow argparse to process the left over post-verb arguments along
    # with the pre-verb arguments and the verb itself
    args = parser.parse_args(pre_verb_args + [verb] + processed_post_verb_args)
    # Extend the argparse result with the extras from the preprocessor
```

```
    for key, value in extras.items():
        setattr(args, key, value)

    # Finally call the subparser's main function with the processed args
    # and the extras which the preprocessor may have returned
    sys.exit(args.main(args) or 0)
```

This function is mostly boilerplate in that it will likely not change much between commands of different types, but it would also be less transparent to have this function created for you. If you are using this boilerplate to implement your command, then you should be careful to update `COMMAND_NAME` to reflect your command's name.

This line defines the `entry_point` group for your command's verbs:

```
VERBS_ENTRY_POINT = '{0}.verbs'.format(COMMAND_NAME)
```

In the case that your command is called `foo` then this would become `foo.verbs`. This name is important because it is how verbs for this command can be provided by your Python package or others. For example, each verb for your command `foo` will need entry in the `setup.py` of its containing package, like this:

```
setup(
    ...
    entry_points={
        ...
        'foo.verbs': [
            'bar = foo.verbs.bar:entry_point_data',
        ],
    }
)
```

You can see here that you are defining `bar` to be a an entry_point of type `foo.verbs` which in turn points to a module and reference `foo.verbs.bar` and `entry_point_data`. At run time this verb pattern will let your command lookup all things defined as `foo.verbs` and load up the reference to which they point.

1.2.2 Adding Verbs

In order to add a verb to your command, a few things must happen.

First you must have an entry in the `setup.py` as described above. This allows the command to find the `entry_point` for your verb at run time. The `entry_point` for these verbs should point to a dictionary which describes the verb being added.

This is an example of an `entry_point_data` dictionary for a verb:

```
entry_point_data = dict(
    verb='build',
    description='Builds a workspace of packages',
    # Called for execution, given parsed arguments object
    main=main,
    # Called first to setup argparse, given argparse parser
    prepare_arguments=prepare_arguments,
    # Called after prepare_arguments, but before argparse.parse_args
    argument_preprocessor=argument_preprocessor,
)
```

As you can see this dictionary describes the verb and gives references to functions which allow the command to describe the verb, hook into argparse parameter creation for the verb, and to execute the verb. The `verb`, `description`, `main`, and `prepare_arguments` keys of the dictionary are required, but the `argument_preprocessor` key is optional.

- `verb`: This is the name of the verb, and is how the command knows which verb implementation to match to a verb on the command line.
- `description`: This is used by the argument parsing to describe the verb in `--help`.
- `prepare_arguments`: This function gets called to allow the verb to setup it's own argparse options. This function should always take one parameter which is the `argparse.ArgumentParser` for this verb, to which arguments can be added. It can optionally take a second parameter which are the current command line arguments. This is not always needed, but can be useful in some cases. This function should always return the parser.
- `argument_preprocessor`: This function is optional, but allows your verb an opportunity to process the raw arguments before they are passed to argparse's `parse_args` function. This can be useful when argparse is not capable of processing the options correctly.
- `main`: This is the implementation of the verb, it gets called last and is passed the parsed arguments. The return type of this function is used for `sys.exit`, a return type of `None` is interpreted as `0`.

Here is an invented example of `main`, `prepare_arguments`, and `argument_preprocessor`:

```
def prepare_arguments(parser):
    parser.add_argument('--some-argument', action='store_true', default=False)
    return parser

def argument_preprocessor(args):
    extras = {}

    if '-strange-argument' in args:
        args.remove('-strange-argument')
        extras['strange_argument'] = True

    return args, extras

def main(options):
    print('--some-argument:', options.some_argument)
    print('-strange-argument:', options.strange_argument)
    if options.strange_argument:
        return 1
    return 0
```

The above example is simply to illustrate the signature of these functions and how they might be used.

1.2.3 Verb Pattern API

API for implementing commands and verbs which used the verb pattern.

osrf_pycommon.cli_utils.verb_pattern.**call_prepare_arguments**(*func*, *parser*, *sysargs=None*)

Call a prepare_arguments function with the correct number of parameters.

The `prepare_arguments` function of a verb can either take one parameter, `parser`, or two parameters `parser` and `args`, where `args` are the current arguments being processed.

Parameters

- **func** (*Callabe*) – Callable `prepare_arguments` function.
- **parser** (`argparse.ArgumentParser`) – parser which is always passed to the function
- **sysargs** (*list*) – arguments to optionally pass to the function, if needed

Returns return value of function or the parser if the function returns None.

Return type argparse.ArgumentParser

Raises ValueError if a function with the wrong number of parameters is given

osrf_pycommon.cli_utils.verb_pattern.**create_subparsers**(*parser*, *cmd_name*, *verbs*, *group*, *sysargs*, *title=None*)

Creates argparse subparsers for each verb which can be discovered.

Using the verbs parameter, the availble verbs are iterated through. For each verb a subparser is created for it using the parser parameter. The cmd_name is used to fill the title and description of the add_subparsers function call. The group parameter is used with each verb to load the verb's description, prepare_arguments function, and the verb's argument_preprocessors if available. Each verb's prepare_arguments function is called, allowing them to add arguments. Finally a list of argument_preprocessors functions and verb subparsers are returned, one for each verb.

Parameters

- **parser** (argparse.ArgumentParser) – parser for this command
- **cmd_name** (*str*) – name of the command to which the verbs are being added
- **verbs** (*list*) – list of verbs (by name as a string)
- **group** (*str*) – name of the entry_point group for the verbs
- **sysargs** (*list*) – list of system arguments
- **title** (*str*) – optional custom title for the command

Returns tuple of argument_preprocessors and verb subparsers

Return type tuple

osrf_pycommon.cli_utils.verb_pattern.**default_argument_preprocessor**(*args*)

Return unmodifed args and an empty dict for extras

osrf_pycommon.cli_utils.verb_pattern.**list_verbs**(*group*)

List verbs available for a given entry_point group.

Parameters **group** (*str*) – entry_point group name for the verbs to list

Returns list of verb names for the given entry_point group

Return type list of str

osrf_pycommon.cli_utils.verb_pattern.**load_verb_description**(*verb_name*, *group*)

Load description of a verb in a given group by name.

Parameters

- **verb_name** (*str*) – name of the verb to load, as a string
- **group** (*str*) – entry_point group name which the verb is in

Returns verb description

Return type dict

osrf_pycommon.cli_utils.verb_pattern.**split_arguments_by_verb**(*arguments*)

Split arguments by verb.

Given a list of arguments (list of strings), the verb, the pre verb arguments, and the post verb arugments are returned.

For example:

```
>>> args = ['--command-arg1', 'verb', '--verb-arg1', '--verb-arg2']
>>> split_arguments_by_verb(args)
('verb', ['--command-arg1'], ['--verb-arg1', '--verb-arg2'])
```

Parameters **arguments** (*list*) – list of system arguments

Returns the verb (str), pre verb args (list), and post verb args (list)

Return type tuple

CHAPTER 2

The `process_utils` Module

This module provides functions for doing process management.

These are the main sections of this module:

2.1 Asynchronous Process Utilities

There is a function and class which can be used together with your custom Tollius or asyncio run loop.

The *osrf_pycommon.process_utils.async_execute_process()* function is a coroutine which allows you to run a process and get the output back bit by bit in real-time, either with stdout and stderr separated or combined. This function also allows you to emulate the terminal using a pty simply by toggling a flag in the parameters.

Along side this coroutine is a Protocol class, *osrf_pycommon.process_utils.AsyncSubprocessProtocol*, from which you can inherit in order to customize how the yielded output is handled.

Because this coroutine is built on the `trollius`/`asyncio` framework's subprocess functions, it is portable and should behave the same on all major OS's. (including on Windows where an IOCP implementation is used)

`osrf_pycommon.process_utils.`**`async_execute_process`**(*protocol_class*, *cmd=None*, *cwd=None*, *env=None*, *shell=False*, *emulate_tty=False*, *stderr_to_stdout=True*)

Coroutine to execute a subprocess and yield the output back asynchronously.

This function is meant to be used with the Python `asyncio` module, which is available via pip with Python 3.3 and built-in to Python 3.4. On Python >= 2.6 you can use the `trollius` module to get the same functionality, but without using the new `yield from` syntax.

Here is an example of how to use this function:

```
import asyncio
from osrf_pycommon.process_utils import async_execute_process
from osrf_pycommon.process_utils import AsyncSubprocessProtocol
from osrf_pycommon.process_utils import get_loop


@asyncio.coroutine
def setup():
```

```
        transport, protocol = yield from async_execute_process(
            AsyncSubprocessProtocol, ['ls', '/usr'])
        returncode = yield from protocol.complete
        return returncode

    retcode = get_loop().run_until_complete(setup())
    get_loop().close()
```

That same example using `trollius` would look like this:

```
    import trollius as asyncio
    from osrf_pycommon.process_utils import async_execute_process
    from osrf_pycommon.process_utils import AsyncSubprocessProtocol
    from osrf_pycommon.process_utils import get_loop


    @asyncio.coroutine
    def setup():
        transport, protocol = yield asyncio.From(async_execute_process(
            AsyncSubprocessProtocol, ['ls', '/usr']))
        returncode = yield asyncio.From(protocol.complete)
        raise asyncio.Return(returncode)

    retcode = get_loop().run_until_complete(setup())
    get_loop().close()
```

This difference is required because in Python < 3.3 the `yield from` syntax is not valid.

In both examples, the first argument is the default `AsyncSubprocessProtocol` protocol class, which simply prints output from stdout to stdout and output from stderr to stderr.

If you want to capture and do something with the output or write to the stdin, then you need to subclass from the `AsyncSubprocessProtocol` class, and override the `on_stdout_received`, `on_stderr_received`, and `on_process_exited` functions.

See the documentation for the `AsyncSubprocessProtocol` class for more details, but here is an example which uses asyncio from Python 3.4:

```
    import asyncio
    from osrf_pycommon.process_utils import async_execute_process
    from osrf_pycommon.process_utils import AsyncSubprocessProtocol
    from osrf_pycommon.process_utils import get_loop


    class MyProtocol(AsyncSubprocessProtocol):
        def __init__(self, file_name, **kwargs):
            self.fh = open(file_name, 'w')
            AsyncSubprocessProtocol.__init__(self, **kwargs)

        def on_stdout_received(self, data):
            # Data has line endings intact, but is bytes in Python 3
            self.fh.write(data.decode('utf-8'))

        def on_stderr_received(self, data):
            self.fh.write(data.decode('utf-8'))

        def on_process_exited(self, returncode):
            self.fh.write("Exited with return code: {0}".format(returncode))
            self.fh.close()
```

```
    @asyncio.coroutine
    def log_command_to_file(cmd, file_name):

        def create_protocol(**kwargs):
            return MyProtocol(file_name, **kwargs)

        transport, protocol = yield from async_execute_process(
            create_protocol, cmd)
        returncode = yield from protocol.complete
        return returncode

    get_loop().run_until_complete(
        log_command_to_file(['ls', '/'], '/tmp/out.txt'))
    get_loop().close()
```

See the `subprocess.Popen` class for more details on some of the parameters to this function like `cwd`, `env`, and `shell`.

See the *osrf_pycommon.process_utils.execute_process()* function for more details on the `emulate_tty` parameter.

Parameters

- **protocol_class** (`AsyncSubprocessProtocol` or a subclass) – Protocol class which handles subprocess callbacks
- **cmd** (*list*) – list of arguments where the executable is the first item
- **cwd** (*str*) – directory in which to run the command
- **env** (*dict*) – a dictionary of environment variable names to values
- **shell** (*bool*) – if True, the `cmd` variable is interpreted by a the shell
- **emulate_tty** (*bool*) – if True, pty's are passed to the subprocess for stdout and stderr, see *osrf_pycommon.process_utils.execute_process()*.
- **stderr_to_stdout** (*bool*) – if True, stderr is directed to stdout, so they are not captured separately.

class `osrf_pycommon.process_utils.`**AsyncSubprocessProtocol**(*stdin=None*, *stdout=None*, *stderr=None*)

Protocol to subclass to get events from `async_execute_process()`.

When subclassing this Protocol class, you should override these functions:

```
    def on_stdout_received(self, data):
        # ...

    def on_stderr_received(self, data):
        # ...

    def on_process_exited(self, returncode):
        # ...
```

By default these functions just print the data received from stdout and stderr and does nothing when the process exits.

Data received by the `on_stdout_received` and `on_stderr_received` functions is always in bytes (`str` in Python2 and `bytes` in Python3). Therefore, it may be necessary to call `.decode()` on the data before printing to the screen.

Additionally, the data received will not be stripped of new lines, so take that into consideration when printing the result.

You can also override these less commonly used functions:

```
def on_stdout_open(self):
    # ...

def on_stdout_close(self, exc):
    # ...

def on_stderr_open(self):
    # ...

def on_stderr_close(self, exc):
    # ...
```

These functions are called when stdout/stderr are opened and closed, and can be useful when using pty's for example. The `exc` parameter of the `*_close` functions is None unless there was an exception.

In addition to the overridable functions this class has a few useful public attributes. The `stdin` attribute is a reference to the PipeProto which follows the `asyncio.WriteTransport` interface. The `stdout` and `stderr` attributes also reference their PipeProto. The `complete` attribute is a `asyncio.Future` which is set to complete when the process exits and its result is the return code.

The `complete` attribute can be used like this:

```
import asyncio
from osrf_pycommon.process_utils import async_execute_process
from osrf_pycommon.process_utils import AsyncSubprocessProtocol
from osrf_pycommon.process_utils import get_loop


@asyncio.coroutine
def setup():
    transport, protocol = yield from async_execute_process(
        AsyncSubprocessProtocol, ['ls', '-G', '/usr'])
    retcode = yield from protocol.complete
    print("Exited with", retcode)

# This will block until the protocol.complete Future is done.
get_loop().run_until_complete(setup())
get_loop().close()
```

In addtion to these functions, there is a utility function for getting the correct `asyncio` event loop:

`osrf_pycommon.process_utils.`**`get_loop`**`()`

This function will return the proper event loop for the subprocess async calls.

On Unix this just returns `asyncio.get_event_loop()`, but on Windows it will set and return a `asyncio.ProactorEventLoop` instead.

2.2 Treatment of File Descriptors

Unlike `subprocess.Popen`, all of the `process_utils` functions behave the same way on Python versions 2.7 through 3.4, and they do not close *inheritable <https://docs.python.org/3.4/library/os.html#fd-inheritance>*. file descriptors before starting subprocesses. This is equivalent to passing `close_fds=False` to `subprocess.Popen` on all Python versions.

In Python 3.2, the `subprocess.Popen` default for the `close_fds` option changed from `False` to `True` so that file descriptors opened by the parent process were closed before spawning the child process. In Python 3.4, PEP 0446 additionally made it so even when `close_fds=False` file descriptors which are non-inheritable are still closed before spawning the subprocess.

If you want to be able to pass file descriptors to subprocesses in Python 3.4 or higher, you will need to make sure they are *inheritable <https://docs.python.org/3.4/library/os.html#fd-inheritance>*.

2.3 Synchronous Process Utilities

For synchronous execution and output capture of subprocess, there are two functions:

- *osrf_pycommon.process_utils.execute_process()*
- *osrf_pycommon.process_utils.execute_process_split()*

These functions are not yet using the `trollius`/`asyncio` framework as a back-end and therefore on Windows will not stream the data from the subprocess as it does on Unix machines. Instead data will not be yielded until the subprocess is finished and all output is buffered (the normal warnings about long running programs with lots of output apply).

The streaming of output does not work on Windows because on Windows the `select.select()` method only works on sockets and not file-like objects which are used with subprocess pipes. `asyncio` implements Windows subprocess support by implementing a Proactor event loop based on Window's IOCP API. One future option will be to implement this synchronous style method using IOCP in this module, but another option is to just make synchronous the asynchronous calls, but there are issues with that as well. In the mean time, if you need streaming of output in both Windows and Unix, use the asynchronous calls.

`osrf_pycommon.process_utils.`**`execute_process`**(*cmd*, *cwd=None*, *env=None*, *shell=False*, *emulate_tty=False*)

Executes a command with arguments and returns output line by line.

All arguments, except `emulate_tty`, are passed directly to `subprocess.Popen`.

`execute_process` returns a generator which yields the output, line by line, until the subprocess finishes at which point the return code is yielded.

This is an example of how this function should be used:

```
from __future__ import print_function
from osrf_pycommon.process_utils import execute_process

cmd = ['ls', '-G']
for line in execute_process(cmd, cwd='/usr'):
    if isinstance(line, int):
        # This is a return code, the command has exited
        print("'{0}' exited with: {1}".format(' '.join(cmd), line))
        continue  # break would also be appropriate here
    # In Python 3, it will be a bytes array which needs to be decoded
    if not isinstance(line, str):
        line = line.decode('utf-8')
    # Then print it to the screen
    print(line, end='')
```

`stdout` and `stderr` are always captured together and returned line by line through the returned generator. New line characters are preserved in the output, so if re-printing the data take care to use `end=''` or first `rstrip` the output lines.

When `emulate_tty` is used on Unix systems, commands will identify that they are on a tty and should output color to the screen as if you were running it on the terminal, and therefore there should not be any need to pass arguments like `-c color.ui=always` to commands like `git`. Additionally, programs might also behave differently in when `emulate_tty` is being used, for example, Python will default to unbuffered output when it detects a tty.

`emulate_tty` works by using psuedo-terminals on Unix machines, and so if you are running this command many times in parallel (like hundreds of times) then you may get one of a few different `OSError`'s. For example, "OSError: [Errno 24] Too many open files: '/dev/ttyp0'" or "OSError: out of pty devices". You should also be aware that you share pty devices with the rest of the system, so even if you are not using a lot, it is possible to get this error. You can catch this error before getting data from the generator, so when using `emulate_tty` you might want to do something like this:

```
from __future__ import print_function
from osrf_pycommon.process_utils import execute_process

cmd = ['ls', '-G', '/usr']
try:
    output = execute_process(cmd, emulate_tty=True)
except OSError:
    output = execute_process(cmd, emulate_tty=False)
for line in output:
    if isinstance(line, int):
        print("'{0}' exited with: {1}".format(' '.join(cmd), line))
        continue
    # In Python 3, it will be a bytes array which needs to be decoded
    if not isinstance(line, str):
        line = line.decode('utf-8')
    print(line, end='')
```

This way if a pty cannot be opened in order to emulate the tty then you can try again without emulation, and any other `OSError` should raise again with `emulate_tty` set to `False`. Obviously, you only want to do this if emulating the tty is non-critical to your processing, like when you are using it to capture color.

Any color information that the command outputs as ANSI escape sequences is captured by this command. That way you can print the output to the screen and preserve the color formatting.

If you do not want color to be in the output, then try setting `emulate_tty` to `False`, but that does not guarantee that there is no color in the output, instead it only will cause called processes to identify that they are not being run in a terminal. Most well behaved programs will not output color if they detect that they are not being executed in a terminal, but you shouldn't rely on that.

If you want to ensure there is no color in the output from an executed process, then use this function:

osrf_pycommon.terminal_color.remove_ansi_escape_senquences()

Exceptions can be raised by functions called by the implementation, for example, `subprocess.Popen` can raise an `OSError` when the given command is not found. If you want to check for the existence of an executable on the path, see: `which()`. However, this function itself does not raise any special exceptions.

Parameters

- **cmd** (*list*) – list of strings with the first item being a command and subsequent items being any arguments to that command; passed directly to `subprocess.Popen`.
- **cwd** (*str*) – path in which to run the command, defaults to None which means `os.getcwd()` is used; passed directly to `subprocess.Popen`.
- **env** (*dict*) – environment dictionary to use for executing the command, default is None which uses the `os.environ` environment; passed directly to `subprocess.Popen`.

- **shell** (*bool*) – If True the system shell is used to evaluate the command, default is False; passed directly to `subprocess.Popen`.
- **emulate_tty** (*bool*) – If True attempts to use a pty to convince subprocess's that they are being run in a terminal. Typically this is useful for capturing colorized output from commands. This does not work on Windows (no pty's), so it is considered False even when True. Defaults to False.

Returns a generator which yields output from the command line by line

Return type generator which yields strings

Availability: Unix (streaming), Windows (blocking)

`osrf_pycommon.process_utils.`**execute_process_split**(*cmd*, *cwd=None*, *env=None*, *shell=False*, *emulate_tty=False*)

`execute_process()`, except `stderr` is returned separately.

Instead of yielding output line by line until yielding a return code, this function always a triplet of `stdout`, `stderr`, and return code. Each time only one of the three will not be None. Once you receive a non-None return code (type will be int) there will be no more `stdout` or `stderr`. Therefore you can use the command like this:

```
from __future__ import print_function
import sys
from osrf_pycommon.process_utils import execute_process_split

cmd = ['time', 'ls', '-G']
for out, err, ret in execute_process_split(cmd, cwd='/usr'):
    # In Python 3, it will be a bytes array which needs to be decoded
    out = out.decode('utf-8') if out is not None else None
    err = err.decode('utf-8') if err is not None else None
    if ret is not None:
        # This is a return code, the command has exited
        print("'{0}' exited with: {1}".format(' '.join(cmd), ret))
        break
    if out is not None:
        print(out, end='')
    if err is not None:
        print(err, end='', file=sys.stderr)
```

When using this, it is possible that the `stdout` and `stderr` data can be returned in a different order than what would happen on the terminal. This is due to the fact that the subprocess is given different buffers for `stdout` and `stderr` and so there is a race condition on the subprocess writing to the different buffers and this command reading the buffers. This can be avoided in most scenarios by using `emulate_tty`, because of the use of `pty`'s, though the ordering can still not be guaranteed and the number of `pty`'s is finite as explained in the documentation for `execute_process()`. For situations where output ordering between `stdout` and `stderr` are critical, they should not be returned separately and instead should share one buffer, and so `execute_process()` should be used.

For all other parameters and documentation see: `execute_process()`

Availability: Unix (streaming), Windows (blocking)

2.4 Utility Functions

Currently there is only one utility function, a Python implementation of the `which` shell command.

osrf_pycommon.process_utils.**which**(*cmd*, *mode=1*, *path=None*, ***kwargs*)

Given a command, mode, and a PATH string, return the path which conforms to the given mode on the PATH, or None if there is no such file.

mode defaults to `os.F_OK | os.X_OK`. *path* defaults to the result of `os.environ.get("PATH")`, or can be overridden with a custom search path.

Backported from `shutil.which()` (https://docs.python.org/3.3/library/shutil.html#shutil.which), available in Python 3.3.

CHAPTER 3

The `terminal_color` Module

This module provides tools for colorizing terminal output.

This module defines the ansi escape sequences used for colorizing the output from terminal programs in Linux. You can access the ansi escape sequences using the *`ansi()`* function:

```
>>> from osrf_pycommon.terminal_color import ansi
>>> print(["This is ", ansi('red'), "red", ansi('reset'), "."])
['This is ', '\x1b[31m', 'red', '\x1b[0m', '.']
```

You can also use *`format_color()`* to do in-line substitution of keys wrapped in `@{}` markers for their ansi escape sequences:

```
>>> from osrf_pycommon.terminal_color import format_color
>>> print(format_color("This is @{bf}blue@{reset}.").split())
['This', 'is', '\x1b[34mblue\x1b[0m.']
```

This is a list of all of the available substitutions:

Long Form	Shorter	Value
@{blackf}	@{kf}	\033[30m
@{redf}	@{rf}	\033[31m
@{greenf}	@{gf}	\033[32m
@{yellowf}	@{yf}	\033[33m
@{bluef}	@{bf}	\033[34m
@{purplef}	@{pf}	\033[35m
@{cyanf}	@{cf}	\033[36m
@{whitef}	@{wf}	\033[37m
@{blackb}	@{kb}	\033[40m
@{redb}	@{rb}	\033[41m
@{greenb}	@{gb}	\033[42m
@{yellowb}	@{yb}	\033[43m
@{blueb}	@{bb}	\033[44m
@{purpleb}	@{pb}	\033[45m
@{cyanb}	@{cb}	\033[46m
@{whiteb}	@{wb}	\033[47m
@{escape}		\033
@{reset}	@|	\033[0m
@{boldon}	@!	\033[1m
@{italicson}	@/	\033[3m
@{ulon}	@_	\033[4m
@{invon}		\033[7m
@{boldoff}		\033[22m
@{italicsoff}		\033[23m
@{uloff}		\033[24m
@{invoff}		\033[27m

These substitution's values come from the ANSI color escape sequences, see: http://en.wikipedia.org/wiki/ANSI_escape_code

Also for any of the keys which have a trailing `f`, you can safely drop the trailing `f` and get the same thing.

For example, `format_color("@{redf}")` and `format_color("@{red}")` are functionally equivalent.

Also, many of the substitutions have shorten forms for convenience, such that `@{redf}`, `@{rf}`, `@{red}`, and `@{r}` are all the same.

Note that a trailing `b` is always required when specifying a background.

Some of the most common non-color sequences have `{}`'less versions.

For example, `@{boldon}`'s shorter form is `@!`.

By default, the substitutions (and calls to *ansi()*) resolve to escape sequences, but if you call *disable_ansi_color_substitution_globally()* then they will resolve to empty strings.

This allows you to always use the substitution strings and disable them globally when desired.

On Windows the substitutions are always resolved to empty strings as the ansi escape sequences do not work on Windows. Instead strings annotated with `@{}` style substitutions or raw `\x1b[xxm` style ansi escape sequences must be passed to *print_color()* in order for colors to be displayed on windows. Also the *print_ansi_color_win32()* function can be used on strings which only contain ansi escape sequences.

Note: There are existing Python modules like colorama which provide ansi colorization on multiple platforms, so a valid question is: "why write this module?". The reason for writing this module is to provide the color annotation of strings and functions for removing or replacing ansi escape sequences which are not provided by modules like colorama. This module could have depended on colorama for colorization on Windows, but colorama works by

replacing the built-in `sys.stdout` and `sys.stderr`, which we did not want and it has extra functionality that we do not need. So, instead of depending on colorama, the Windows color printing code was used as the inspiration for the Windows color printing in the `windows.py` module in this `terminal_color` package. The colorama license was placed in the header of that file and the colorama license is compatible with this package's license.

---

osrf_pycommon.terminal_color.**ansi**(*key*)
Returns the escape sequence for a given ansi color key.

osrf_pycommon.terminal_color.**disable_ansi_color_substitution_globally**()
Causes *format_color()* to replace color annotations with empty strings.

It also affects *ansi()*.

This is not the case by default, so if you want to make all substitutions given to either function mentioned above return empty strings then call this function.

The default behavior can be restored by calling *enable_ansi_color_substitution_globally()*.

osrf_pycommon.terminal_color.**enable_ansi_color_substitution_globally**()
Causes *format_color()* to replace color annotations with ansi esacpe sequences.

It also affects *ansi()*.

This is the case by default, so there is no need to call this everytime.

If you have previously caused all substitutions to evaluate to an empty string by calling *disable_ansi_color_substitution_globally()*, then you can restore the escape sequences for substitutions by calling this function.

osrf_pycommon.terminal_color.**format_color**(*msg*)
Replaces color annotations with ansi escape sequences.

See this module's documentation for the list of available substitutions.

If *disable_ansi_color_substitution_globally()* has been called then all color annotations will be replaced by empty strings.

Also, on Windows all color annotations will be replaced with empty strings. If you want colorization on Windows, you must pass annotated strings to *print_color()*.

Parameters **msg** (*str*) – string message to be colorized

Returns colorized string

Return type str

osrf_pycommon.terminal_color.**get_ansi_dict**()
Returns a copy of the dictionary of keys and ansi escape sequences.

osrf_pycommon.terminal_color.**print_ansi_color_win32**(**args*, ***kwargs*)
Prints color string containing ansi escape sequences to console in Windows.

If called on a non-Windows system, a `NotImplementedError` occurs.

Does not respect *disable_ansi_color_substitution_globally()*.

Does not substitute color annotations like `@{r}` or `@!`, the string must already contain the `\033[1m` style ansi escape sequences.

Works by splitting each argument up by ansi escape sequence, printing the text between the sequences, and doing the corresponding win32 action for each ansi sequence encountered.

osrf_pycommon.terminal_color.**print_color**(**args*, ***kwargs*)
Colorizes and prints with an implicit ansi reset at the end

Calls *format_color()* on each positional argument and then sends all positional and keyword arguments to print.

If the end keyword argument is not present then the default end value ansi('reset') + '\n' is used and passed to print.

os.linesep is used to determine the actual value for \n.

Therefore, if you use the end keyword argument be sure to include an ansi reset escape sequence if necessary.

On Windows the substituted arguments and keyword arguments are passed to *print_ansi_color_win32()* instead of just print.

osrf_pycommon.terminal_color.**remove_ansi_escape_senquences**(*string*)
Removes any ansi escape sequences found in the given string and returns it.

osrf_pycommon.terminal_color.**sanitize**(*msg*)
Sanitizes the given string to prevent *format_color()* from substituting content.

For example, when the string 'Email: {user}@{org}' is passed to *format_color()* the @{org} will be incorrectly recognized as a colorization annotation and it will fail to substitute with a KeyError: org.

In order to prevent this, you can first "sanatize" the string, add color annotations, and then pass the whole string to *format_color()*.

If you give this function the string 'Email: {user}@{org}', then it will return 'Email: {{user}}@@{{org}}'. Then if you pass that to *format_color()* it will return 'Email: {user}@{org}'. In this way *format_color()* is the reverse of this function and so it is safe to call this function on any incoming data if it will eventually be passed to *format_color()*.

In addition to expanding { => {{, } => }}, and @ => @@, this function will also replace any instances of @!, @/, @_, and @| with @{atexclimation}, @{atfwdslash}, @{atunderscore}, and @{atbar} respectively. And then there are corresponding keys in the ansi dict to convert them back.

For example, if you pass the string '|@ Notice @|' to this function it will return '|@@ Notice @{atbar}'. And since ansi('atbar') always returns @|, even when *disable_ansi_color_substitution_globally()* has been called, the result of passing that string to *format_color()* will be '|@ Notice @|' again.

There are two main strategies for constructing strings which use both the Python str.format() function and the colorization annotations.

One way is to just build each piece and concatenate the result:

```
print_color("@{r}", "{error}".format(error=error_str))
# Or using print (remember to include an ansi reset)
print(format_color("@{r}" + "{error}".format(error=error_str) + "@|"))
```

Another way is to use this function on the format string, concatenate to the annotations, pass the whole string to *format_color()*, and then format the whole thing:

```
print(format_color("@{r}" + sanitize("{error}") + "@|")
      .format(error=error_str))
```

However, the most common use for this function is to sanitize incoming strings which may have unknown content:

```
def my_func(user_content):
    print_color("@{y}" + sanitize(user_content))
```

This function is not intended to be used on strings with color annotations.

Parameters **msg** (*str*) – string message to be sanitized

Returns sanitized string

Return type str

osrf_pycommon.terminal_color.**split_by_ansi_escape_sequence**(*string*, *include_delimiters=False*)

Splits a string into a list using any ansi escape sequence as a delimiter.

Parameters

- **string** (*str*) – string to be split
- **include_delimiters** (*bool*) – If True include matched escape sequences in the list (default: False)

Returns list of strings, split from original string by escape sequences

Return type list

osrf_pycommon.terminal_color.**test_colors**(*file=None*)

Prints a color testing block using *print_color()*

CHAPTER 4

The `terminal_utils` Module

This module has a miscellaneous set of functions for working with terminals.

You can use the *`get_terminal_dimensions()`* to get the width and height of the terminal as a tuple.

You can also use the *`is_tty()`* function to determine if a given object is a tty.

exception `osrf_pycommon.terminal_utils.`**`GetTerminalDimensionsError`**
Raised when the terminal dimensions cannot be determined.

`osrf_pycommon.terminal_utils.`**`get_terminal_dimensions`**`()`
Returns the width and height of the terminal.

Returns width and height in that order as a tuple

Return type tuple

Raises GetTerminalDimensionsError when the terminal dimensions cannot be determined

`osrf_pycommon.terminal_utils.`**`is_tty`**`(`*stream*`)`
Returns True if the given stream is a tty, else False

Parameters **`stream`** – object to be checked for being a tty

Returns True if the given object is a tty, otherwise False

Return type bool

CHAPTER 5

Installing from Source

Given that you have a copy of the source code, you can install `osrf_pycommon` like this:

```
$ python setup.py install
```

Note: If you are installing to a system Python you may need to use `sudo`.

If you do not want to install `osrf_pycommon` into your system Python, or you don't have access to `sudo`, then you can use a virtualenv.

CHAPTER 6

Hacking

Because osrf_pycommon uses setuptools you can (and should) use the develop feature:

```
$ python setup.py develop
```

Note: If you are developing against the system Python, you may need sudo.

This will "install" osrf_pycommon to your Python path, but rather than copying the source files, it will instead place a marker file in the PYTHONPATH redirecting Python to your source directory. This allows you to use it as if it were installed but where changes to the source code take immediate affect.

When you are done with develop mode you can (and should) undo it like this:

```
$ python setup.py develop -u
```

Note: If you are developing against the system Python, you may need sudo.

That will "uninstall" the hooks into the PYTHONPATH which point to your source directory, but you should be wary that sometimes console scripts do not get removed from the bin folder.

CHAPTER 7

Testing

In order to run the tests you will need to install nosetests, flake8, and Mock.

Once you have installed those, then run `nosetest` in the root of the `osrf_pycommon` source directory:

```
$ nosetests
```

CHAPTER 8

Building the Documentation

In order to build the docs you will need to first install Sphinx.

You can build the documentation by invoking the Sphinx provided make target in the `docs` folder:

```
$ # In the docs folder
$ make html
$ open _build/html/index.html
```

Sometimes Sphinx does not pickup on changes to modules in packages which utilize the `__all__` mechanism, so on repeat builds you may need to clean the docs first:

```
$ # In the docs folder
$ make clean
$ make html
$ open _build/html/index.html
```

O

A

C

D

E

F

G

I

L

O

P

R

S

T

W