dlpt

Release 2.0.0

Domen Jurkovic @ Damogran Labs

Aug 19, 2023

MODULES:

README: https://github.com/damogranlabs/dlpt

CHAPTER

ONE

DLPT.IMPORTER MODULE

Dynamically import python module in the runtime.

class dlpt.importer.**ModuleImporter**(*file_path: str*, *base_dir_path: str | None = None*)

Bases: object

__init__(*file_path: str*, *base_dir_path: str | None = None*)

Dynamically import module from a given file_path.

Parameters

- **file_path** – abs path to a python module (file) which will be dynamically imported.
- **base_dir_path** – path to a root directory from where module will be imported. If not specified, file_path root directory is used as base_dir_path. Example: file_path = C:/root/someDir/someSubdir/myModule.py base_dir_path = C:/root/someDir/ -> module will be imported as: *someSubdir.myModule*

Note: base_dir_path is added to *sys.path*. It is NOT removed once object is garbage-collected.

get_module() → module

Returns

Imported module instance (object).

has_object(*name: str*, *raise_exception: bool = True*) → bool

Check if imported module has object with name.

Parameters

- **name** – name of the object to check existence in imported module.
- **raise_exception** – if True, exception is raised if object is not found. Otherwise, bool is returned (True if object is found, False otherwise).

Returns

True if imported module has object with *name*, False otherwise.

CHAPTER

TWO

DLPT.JSON MODULE

Read and write JSON files, JSON files with comments and (un)picklable JSON data.

dlpt.json.**remove_comments**(*data_str: str*) → str

Return given string with removed C/C++ style comments.

Parameters
data_str – string to remove comments from.

Returns
Input string without C/C++ style comments.

dlpt.json.**read**(*file_path: str*) → Dict[str, Any]

Open the given JSON file, strip comments and return dictionary data.

Parameters
file_path – path to a file that needs to be parsed.

Returns
Data (dictionary) of a given JSON file.

dlpt.json.**write**(*data: Dict[str, Any]*, *file_path: str*, *indent: int = 2*, *sort_keys: bool = True*, **args*, ***kwargs*)

Write given data to a file in a JSON format.

Parameters

- **data** – serializable object to store to a file in JSON format.
- **file_path** – destination file path.
- **indent** – number of spaces to use while building file line indentation.
- **sort_keys** – if True, data keys are sorted alphabetically, else left unchanged.
- ***args** – *json.dump()* additional arguments.
- ****kwargs** – *json.dump()* additional keyword arguments.

dlpt.json.**read_jsonpickle**(*file_path: str*, *classes: object | List[object] | None = None*, **args*) → Any

Read given file and return unpicklable data - python objects with jsonpickle module.

Parameters

- **file_path** – path to a file that needs to be read.
- **classes** – see jsonpickle *decode()* docstring. TLDR: if un-picklable objects are from modules which are not globally available, use `classes` arg to specify them.
- ***args** – *jsonpickle.decode()* additional arguments.

Returns
Python object(s) of unpickled JSON data.

dlpt.json.**write_jsonpickle**(*data: Any*, *file_path: str*, *indent: int = 2*, **args*)

Write given data to a file in a JSON format with jsonpickle module, which adds data type info for unpickling with *read_jsonpickle()*.

Parameters

- **data** – serializable object to store to a file in JSON format.
- **file_path** – destination file path.
- **indent** – number of spaces for line indentation.
- ***args** – *jsonpickle.encode()* additional arguments.

CHAPTER

THREE

DLPT.LOG MODULE

Common wrappers and helper functions to simplify most common use cases of builtin 'logging' module.

1. **Create logger and (optionally) set it as a default *dlpt* logger.**

   Note:

   'Default' logger means that (once initialized), any *dlpt* log functions, such as *info()* and *warning()* will log to this *default* logger. Once initialized as default logger, any other file can simply use *dlpt* log functions without setting up logger.

   Example:

   ```
   >>> #file1.py
   >>> logger = dlpt.log.create_logger("my_logger")
   >>> dlpt.log.add_file_hdlr(logger, "dlpt_example.log")
   >>> dlpt.log.info("Log from file1.")
   >>>
   >>> #file2.py
   >>> dlpt.log.info("Log from file2.")
   ```

2. **Use *dlpt.log.add_*()* functions to add common handlers to any** `logging.Logger` instance: console (terminal) handler, file handler, rotating file handler, server socket handler (for logs when multiple processes are used).

Logging server To unify logs from multiple processes, user can create logging server via function *create_log_server_proc()*. This process will create a custom logger with file handler and open a socket connection on a designated port. Any logger (from any process) that has configured logging server handler (via *add_logging_server_hdlr()*) will push logs to this logging server and therefore to a shared log file. Note that log statements order might not be exactly the same as this is OS-dependant.

`dlpt.log.create_logger`(*name: str | None = None*, *set_as_default: bool = True*, *level: int | None = 10*) → Logger

Create new logger instance with the given 'name' and optionally set it as a default logger whenever *dlpt.log.** log functions are invoked.

Parameters

- **name** – Optional name of the new logger instance or root by default.
- **set_as_default** – If True, created logger instance will be set as a default logger whenever *dlpt.log.** log functions are invoked.
- **level** – log level for this specific logger. If None, everything is logged (`logging.DEBUG` level).

dlpt.log.**add_console_hdlr**(*logger: Logger | str*, *fmt: Formatter | None = None*, *level: int = 10*) → StreamHandler

Add console handler to logger instance.

Note: Create custom formatter with: `logging.Formatter(<formatter>, datefmt=<time formatter>)`

Parameters

- **logger** – logger instance or logger name.
- **fmt** – Optional custom formatter for created handler. By default, DEFAULT_FORMATTER and DEFAULT_FORMATTER_TIME is used.
- **level** – set log level for this specific handler. By default, everything is logged (DEBUG level).

Returns
Created console (stream) handler object.

dlpt.log.**add_file_hdlr**(*logger: Logger | str*, *file_name: str | None = None*, *dir_path: str | None = None*, *fmt: Formatter | None = None*, *level: int = 10*, *mode: str = 'w'*) → Tuple[FileHandler, str]

Add file handler to logger instance.

Parameters

- **logger** – logger instance or logger name.
- **file_name** – name of a log file. If there is no file extension, default DEFAULT_LOG_FILE_EXT is appended. If None, logger name is used as a file name.
- **dir_path** – path to a directory where logs will be stored. If None, path is fetched with *get_default_log_dir()*.If log directory does not exist, it is created.
- **fmt** – Optional custom formatter for created handler. By default, DEFAULT_FORMATTER and DEFAULT_FORMATTER_TIME is used.
- **level** – Log level for this specific handler. By default, everything is logged (DEBUG level).
- **mode** – file open mode (*"w"*, "a", ... See logging docs.).

Returns
(created file handler, file path).

Return type
A tuple

dlpt.log.**add_rotating_file_hdlr**(*logger: Logger | str*, *file_name: str | None = None*, *dir_path: str | None = None*, *fmt: Formatter | None = None*, *level: int = 10*, *max_size_kb: int = 100*, *backup_count: int = 1*) → Tuple[RotatingFileHandler, str]

Add rotating file handler to logger instance.

Parameters

- **logger** – logger instance or logger name.
- **file_name** – name of a log file. If there is no file extension, default DEFAULT_LOG_FILE_EXT is appended. If None, logger name is used as a file name.
- **dir_path** – path to a directory where logs will be stored. If None, path is fetched with *get_default_log_dir()*. If log directory does not exist, it is created.
- **max_size_kb** – number of KB at which rollover is performed on a current log file.

- **backup_count** – number of files to store (if file with given name already exists).
- **fmt** – Optional custom formatter for created handler. By default, DEFAULT_FORMATTER and DEFAULT_FORMATTER_TIME is used.
- **level** – Log level for this specific handler. By default, everything is logged (DEBUG level).

Returns
(created rotating file handler, file path).

Return type
A tuple

dlpt.log.**add_logging_server_hdlr**(*logger: Logger | str*, *port: int = 9020*, *fmt: Formatter | None = None*, *level: int = 10*) → _SocketHandler

Add log socket handler to this logger instance. This function assume that log socket server is already initialized.

Parameters

- **logger** – logger instance or logger name.
- **port** – socket port where logger writes data to.
- **fmt** – Optional custom formatter for created handler. By default, DEFAULT_FORMATTER and DEFAULT_FORMATTER_TIME is used.
- **level** – Log level for this specific handler. By default, everything is logged (DEBUG level).

dlpt.log.**get_log_file_paths**(*logger: Logger | str*) → List[str]

Return log file paths of *logging.FileHandler(s)* of a given logger instance.

Parameters
logger – logger instance or logger name.

Returns
List of loggers file handlers file paths.

dlpt.log.**get_rotating_log_file_paths**(*logger: Logger | str*) → List[str]

Return log file paths of *logging.RotatingFileHandler(s)* of a given logger instance.

Parameters
logger – logger instance or logger name.

Returns
List of loggers rotating file handlers file paths.

dlpt.log.**get_default_logger**() → Logger | None

Get default logger instance object (if set).

Returns
Current logger instance when *dlpt.log.** log functions are invoked.

dlpt.log.**set_default_logger**(*logger: Logger*)

Set default logger instance.

Parameters
logger – logger instance or logger name.

Returns
Current logger instance object.

dlpt.log.**get_file_name**(*logger: Logger | str*, *file_name: str | None = None*) → str

Determine log file name, based on a current logger name or given *file_name*.

Parameters

- **logger** – logger instance or logger name.
- **file_name** – if given, this name is checked (for extension) or default file name is created from logger name.

Returns

File name with extension.

dlpt.log.**get_default_log_dir**() → str

Get default log directory path: <cwd>/DEFAULT_LOG_DIR_NAME.

Returns

Path to a directory where logs are usually created.

class dlpt.log.**ReleaseFileLock**(*logger: Logger | str*, *file_path: str*)

Bases: `object`

__init__(*logger: Logger | str*, *file_path: str*)

Temporary release file handler of logging file streams to be able to copy file (for example, log file is locked by logger on Windows.) Use as a context manager.

Parameters

- **logger** – logger instance or logger name.
- **file_path** – logging file path

Example

```
>>> with dlpt.log.ReleaseFileLock(logger, file_path):
>>>     shutil.move(file_path, tmp_path)
```

class dlpt.log.**LoggingServer**(*port: int = 9020*)

Bases: `ThreadingTCPServer`

Simple TCP socket-based logging server (receiver). All messages are pickled at client TX side and un-pickled here. Any received message is than further handled by `handleLogRecord()`.

Sets Allow address reuse.

allow_reuse_address = True

daemon_threads = True

__init__(*port: int = 9020*)

Init log socket server. By default, this server logs all received messages to a root logger as configured.

Note: Use *add_logging_server_hdlr()* function to add socket handler to any logger instance and logs will be automatically pushed to the created *LoggingServer* process. Note that ports must be properly configured for logging to work.

Parameters

port – port where socket server reads data from.

handle_error(*request*, *client_address*)

Handle an error gracefully. May be overridden.

The default is to print a traceback and continue.

dlpt.log.**create_log_server_proc**(*file_path: str*, *port: int = 9020*) → int

Create socket server logger subprocess that logs all received messages on a given `port` socket to a log file handler.

Parameters

- **file_path** – absolute path to a log file, including extension.
- **port** – port where socket server will listen.

Note: *port* number must be unique - this means that the default *create_log_server_proc()* can be called only once. Further socket servers (of any process) and logger handlers must set ports manually.

Returns

PID of created socket server subprocess.

dlpt.log.**log_server_shutdown_request**(*logger: Logger | str*, *pid: int*, *timeout_sec: int = 3*) → bool

Send 'unique' log message that indicates to server to perform shutdown - close all connections and finish process.

Parameters

- **logger** – logger instance or logger name. Note that logging server handler must be available in this logger.
- **pid** – logging server PID.
- **timeout_sec** – time to wait until logging server PID is checked for status. In case of 0, function return value is not relevant.

Returns

True if server was successfully stopped (PID not alive anymore), False otherwise.

dlpt.log.**debug**(*msg: str*, *logger: Logger | str | None = None*, **args*, ***kwargs*)

Log to a given or default logger (if available) with 'DEBUG' level.

Parameters

- **msg** – message to log.
- **logger** – logger instance or logger name. If not set, default logger is used (as set by *create_logger()*).

dlpt.log.**info**(*msg: str*, *logger: Logger | str | None = None*, **args*, ***kwargs*)

Log to a given or default logger (if available) with 'INFO' level.

Parameters

- **msg** – message to log.
- **logger** – logger instance or logger name. If not set, default logger is used (as set by *create_logger()*).

dlpt.log.**warning**(*msg: str*, *logger: Logger | str | None = None*, **args*, ***kwargs*)

Log to a given or default logger (if available) with 'WARNING' level.

Parameters

- **msg** – message to log.

- **logger** – logger instance or logger name. If not set, default logger is used (as set by *create_logger()*).

dlpt.log.**warning_with_traceback**(*msg: str*, *logger: Logger | str | None = None*, **args*, ***kwargs*)

Log to a given or default logger (if available) with 'WARNING' level and append exception info traceback at the end (if available).

Parameters

- **msg** – message to log.
- **logger** – logger instance or logger name. If not set, default logger is used (as set by *create_logger()*).

dlpt.log.**error**(*msg: str*, *logger: Logger | str | None = None*, **args*, ***kwargs*)

Log to a given or default logger (if available) with 'ERROR' level.

Parameters

- **msg** – message to log.
- **logger** – logger instance or logger name. If not set, default logger is used (as set by *create_logger()*).

dlpt.log.**error_with_traceback**(*msg: str*, *logger: Logger | str | None = None*, **args*, ***kwargs*)

Log to a given or default logger (if available) with 'ERROR' level and append exception info traceback at the end (if available).

Parameters

- **msg** – message to log.
- **logger** – logger instance or logger name. If not set, default logger is used (as set by *create_logger()*).

dlpt.log.**critical**(*msg: str*, *logger: Logger | str | None = None*, **args*, ***kwargs*)

Log to a given or default logger (if available) with 'CRITICAL' level.

Parameters

- **msg** – message to log.
- **logger** – logger instance or logger name. If not set, default logger is used (as set by *create_logger()*).

dlpt.log.**critical_with_traceback**(*msg: str*, *logger: Logger | str | None = None*, **args*, ***kwargs*)

Log to a given or default logger (if available) with 'CRITICAL' level and append exception info traceback at the end (if available).

Parameters

- **msg** – message to log.
- **logger** – logger instance or logger name. If not set, default logger is used (as set by *create_logger()*).

CHAPTER

FOUR

DLPT.PRINT_FILE_CALLABLES MODULE

Print a list of all callable objects (functions/classes) from a given python file. See usage with "–help" argument.

dlpt.print_file_callables.**get_callable_objects_str**(*file_path: str*, *include_private: bool = False*) → List[str]

Get a printable list of strings of all callable objects (functions,
classes, methods) from a given python file.

Note: file must be syntactically correct - importable, since this function
performs a dynamic import.

Note: this function is meant for test/code development, not for actual
production usage.

Parameters

- **file_path** – path to a file to check for object definitions.
- **include_private** – if True, object that starts with '_' are also added. Note: '__' objects are always ignored.

Returns
List of string representation of callable objects from a given file.

dlpt.print_file_callables.**print_callable_objects**(*file_path: str*, *include_private: bool*)

Print a list of all callable objects (functions/classes) from a
given python file.

Note: file must be syntactically correct - importable, since this function
performs a dynamic import.

Note: this function is meant for test/code development, not for actual
production usage.

Parameters

- **file_path** – path to a file to check for object definitions.
- **include_private** – if True, object that starts with '_' are also added. Note: '__' objects are always ignored.

CHAPTER

FIVE

DLPT.PROC MODULE

Functions for spawning, killing and getting process info.

exception dlpt.proc.**SubprocError**(*cmd: str*, *returncode: int*, *stdout: str | None = None*, *stderr: str | None = None*)

Bases: SubprocessError

Same as *subprocess.CalledProcessError*, but does not swallow stderr.

Note: *stdout* is silently swallowed (as with *subprocess.SubprocessError*), since processes stdout can be long and exception can be unreadable.

__init__(*cmd: str*, *returncode: int*, *stdout: str | None = None*, *stderr: str | None = None*)

exception dlpt.proc.**SubprocTimeoutError**(*cmd: str*, *timeout_sec: float*, *stdout: str | None = None*, *stderr: str | None = None*)

Bases: SubprocessError

Same as *subprocess.TimeoutExpired*, but does not swallow stderr.

Note: *stdout* is silently swallowed (as with *subprocess.SubprocessError*), since processes stdout can be long and exception can be unreadable.

__init__(*cmd: str*, *timeout_sec: float*, *stdout: str | None = None*, *stderr: str | None = None*)

dlpt.proc.**get_name**(*pid: str | int*) → str

Get process name.

Note: No PID existence check is performed.

Parameters
pid – PID number.

Returns
Process name as string.

dlpt.proc.**get_executable**(*pid: str | int*) → str

Get process executable path.

Note: No PID existence check is performed.

Parameters
pid – PID number.

Returns
Process executable as string.

dlpt.proc.**get_cmd_args**(*pid: str | int*) → List[str]

Return a list of process command line arguments as it was intially spawned.

Note: No PID existence check is performed.

Parameters
pid – PID number.

Returns
A list of process command line spawn arguments.

dlpt.proc.**is_alive**(*pid: str | int | None*) → bool

Return True if PID exists and process is running, False otherwise. Raise exception if given PID is None.

Parameters
pid – PID number, string or integer.

Returns
True if given PID exists and is running, False otherwise.

dlpt.proc.**get_childs**(*pid: str | int*) → List[int]

Return a list of child processes PIDs.

Note: No PID existence check is performed.

Parameters
pid – PID number of a parent process, string or integer.

Returns
A list of process child processes PIDs.

dlpt.proc.**kill**(*pid: str | int*, *raise_exception: bool = True*, *timeout_sec: int | None = 3*) → bool

Kill process with a given PID.

Parameters

- **pid** – PID number.
- **raise_exception** – if True, exception is raised if process wasn't successfully killed. Otherwise return False.
- **timeout_sec** – wait for specified number of seconds for a process to be killed. If None, return immediately.

Returns
True on successfully terminated process, False otherwise (or exception, based on `raise_exception` input argument).

`dlpt.proc.`**`kill_childs`**(*pid: str | int*, *raise_exception: bool = True*) → List[int]

Kill all child processes of a process with a given PID.

Parameters

- **`pid`** – PID number, string or integer. Raise Exception if None.
- **`raise_exception`** – if True, exception is raised if any of child processes can't be killed.

Returns
A list of killed processes.

`dlpt.proc.`**`kill_tree`**(*pid: str | int*, *raise_exception: bool = True*) → List[int]

Kill parent process and all child processes.

Note: First, child processes are killed, than parent process.

Parameters

- **`pid`** – PID number, string or integer. Raise Exception if None.
- **`raise_exception`** – if True, exception is raised if any of processes can't be killed. Otherwise, False is returned.

Returns
A list of killed processes.

`dlpt.proc.`**`kill_tree_multiple`**(*pids: Sequence[str | int]*, *raise_exception: bool = True*) → List[int]

Iterate over given `pids` and perform *kill_tree()*.

Parameters

- **`pids`** – a list of PIDs - string or integer. Raise Exception if None.
- **`raise_exception`** – if True, exception is raised if any of processes can't be killed.

Returns
A list of killed processes.

`dlpt.proc.`**`get_alive`**(*name_filter: str*) → List[int]

Return a list of currently alive process PIDs.

Parameters
`name_filter` – filter return list by finding `name_filter` in a process name (lower case string is compared).

Example

```
>>> dlpt.proc.get_alive("python.exe")
[26316, 33672, 73992] # pids of local running python.exe processes
```

Returns
A list of currently alive process PIDs.

dlpt.proc.**spawn_non_blocking_subproc**(*args: Sequence[str | int]*) → int

Spawn non-blockin subprocess with given command line arguments and return PID.

Note: If spawned subprocess throw: "OSError: [WinError 740] The requested operation requires elevation" user does not have permission for executing them. Try to re-run script with admin permissions.

Parameters
args – list of subprocess arguments.

Example

```
>>> args = ['python.exe', 'proc.py']
>>> dlpt.proc.spawn_non_blocking_subproc(args)
1234
```

Returns
Spawned process PID.

dlpt.proc.**spawn_subproc**(*args: Sequence[str | int]*, *check_return_code: bool = True*, *stdout: int | None = -1*, *stderr: int | None = -1*, *stdin: int | None = -1*, *encoding: str = 'utf-8'*, *timeout_sec: float | None = None*, ***run_args*) → CompletedProcess

Spawn subprocess and return *subprocess.CompletedProcess* or raise exception. By default, raise exception on timeout (if given) or if return code is not zero. With **run_args, allow setting all subprocess.run() arguments.

Note: If spawned subprocess throw: "OSError: [WinError 740] The requested operation requires elevation" user does not have permission for executing them. Try to re-run script with admin permissions.

Parameters

- **args** – command line arguments with which process will be spawned. Can be shell commands, like ping. Note: all commandline arguments (specifically paths) must be properly encoded. For example, path containing tilde will throw error.
- **check_return_code** – if True, return code is checked by run() function. In case it is not zero, *SubprocError* is raised. If False, *subprocess.CompletedProcess* is returned.
- **stdout** – STDOUT routing specifier.
- **stderr** – STDERR routing specifier.
- **stdin** – STDIN routing specifier. Note: By default, 'stdin' is set to subprocess.PIPE, which should raise exception if spawned subprocess require user input.

- **encoding** – STDOUT/ERR string encoding
- **timeout_sec** – timeout in seconds. If None, no timeout is implemented. Else, if timeout is reached, process is killed and TimeoutExpired exception re-raised.
- **run_args** – optional key-worded subprocess.run() arguments. Note: for the common basic *subprocess.run()* args, see *spawn_subproc()*.

Example

```
>>> args = ['python.exe', 'proc.py']
>>> env_vars = {**os.environ, 'TEST_KEY': 'testenvvar'} # optional kwarg
>>> proc = dlpt.proc.spawn_subproc(args, timeout_sec: 3, env=env_vars)
>>> proc.pid
1234
>>> proc.returncode
0
```

Returns

`CompleteProcess` object once process execution has finished or was terminated.

`dlpt.proc.`**spawn_shell_subproc**(*args: Sequence[str | int]*, *check_return_code: bool = True*, *encoding: str = 'utf-8'*, *timeout_sec: float | None = None*, ***run_args*) → CompletedProcess

Similar to *spawn_subproc()* but for shell commands. STDOUT/ERR is hidden from user and only set in returned *proc.stdout/err*.

Note: If spawned subprocess throw: "OSError: [WinError 740] The requested operation requires elevation" user does not have permission for executing them. Try to re-run script with admin permissions.

Parameters

- **args** – command line arguments with which process will be spawned. Can be shell commands, like ping. Note: all commandline arguments (specifically paths) must be properly encoded. For example, path containing tilde will throw error.
- **check_return_code** – if True, return code is checked by *run()* function. In case it is not zero, *SubprocError* is raised. If False, *subprocess.CompletedProcess* is returned.
- **encoding** – STDOUT/ERR string encoding
- **timeout_sec** – timeout in seconds. If None, no timeout is implemented. Else, if timeout is reached, process is killed and TimeoutExpired exception re-raised.
- **run_args** – optional key-worded *subprocess.run()* arguments, that are added to *run()* call. Note: for the common, basic `subprocess.run()` args, see *spawn_subproc()*

Example

```
>>> args = ['dir']
>>> dlpt.proc.spawn_shell_subproc(args)
proc.py, pth.py, # ...
```

Returns

`CompleteProcess` object once process execution has finished or was terminated.

CHAPTER

SIX

DLPT.PTH MODULE

Functions for common path, file and directory manipulation.

class dlpt.pth.**ChangeDir**(*path: str*)

Bases: object

__init__(*path: str*)

Temporary change working directory of a block of code and revert to an original on exit.

Parameters
path – path to an existing local directory/file that is temporary set as working directory. If file path is given, its directory is taken as new temporary working dir.

Example

```
>>> with dlpt.pth.ChangeDir("C:/somePath"):
        func("that does something in CWD")
```

dlpt.pth.**check**(*path: str | None*) → str

Check if given path exists and return normalized path.

Note: Use standard *os.path.exists()* if you don't want to raise exception.

Parameters
path – path to check

Returns
Normalized path (if valid) or raise exception.

dlpt.pth.**resolve**(*path: str*) → str

Resolve path with pathlib module. This will (for existing files) fix any case mismatch, for example, drive letter.

Parameters
path – abs path to resolve.

Returns
Resolved path according to the OS.

dlpt.pth.**copy_file**(*src_file_path: str*, *dst_dir_path: str*, *dst_file_name: str | None = None*) → str

Copy given file to a new location, while dstFile is removed prior copying. Any intermediate directories are created automatically.

Parameters

- **src_file_path** – path to a file to be copied.
- **dst_dir_path** – absolute destination directory path.
- **dst_file_name** – new destination file name. If None, original file name is used.

Returns
A path to a copied file.

dlpt.pth.**copy_dir**(*src_dir_path: str*, *dst_dir_path: str*) → str

Copy given directory to a new location while creating any intermediate directories. Prior copying, destination directory is removed.

Parameters

- **src_dir_path** – path to a file to be copied.
- **dst_dir_path** – new destination path.

Returns
A path to a copied directory.

dlpt.pth.**remove_file**(*file_path: str*, *force_write_permissions: bool = True*, *retry: int = 3*)

This function tries to remove file (FILE, not DIRECTORY) on a given path. Optionally, write permissions are set to a file.

Parameters

- **file_path** – path to a file.
- **force_write_permissions** – if True, write permissions are set to a file so it can be removed.
- **retry** – on failure, retry removal specified number of times.

dlpt.pth.**remove_dir_tree**(*dir_path: str*, *force_write_permissions: bool = True*, *retry: int = 3*)

Remove directory (DIRECTORY, not FILE) and all its content on a given path.

Parameters

- **dir_path** – path of a directory to remove.
- **force_write_permissions** – if True, *shutil.rmtree()* error callback function is used to change permissions and retry.
- **retry** – on failure, retry removal specified number of times. Must be > 0. Sometimes file are locked with other processes, or a race condition occurred.

dlpt.pth.**clean_dir**(*dir_path: str*, *force_write_permissions: bool = True*)

Delete all directory content (files, sub-directories) in a given directory, but not the root directory itself.

Parameters

- **dir_path** – path to a directory to clean all its content.
- **force_write_permissions** – if True, write permissions are set to be able to delete files.

dlpt.pth.**create_dir**(*dir_path: str*)

Create directory (or directory tree) on a given specified path.

Parameters
dir_path – absolute path of a directory to create.

dlpt.pth.**create_clean_dir**(*dir_path: str*)

Create new or clean existing directory on a given specified path. Path existence is checked with *check()* at the end.

Parameters
dir_path – absolute path of a directory to create/clean.

dlpt.pth.**remove_old_items**(*dir_path: str*, *days: int*) → List[str]

Remove items (files, directories) inside the given directory that were modified more than specified number of days ago.

Note: Modification time and current time can be the same when this function is called right after creation. Hence, decimal part (milliseconds) of current/modification timestamp is discarded.

Parameters

- **dir_path** – path to a directory with files/directories to remove.
- **days** – number of days file/directory must be old to be removed (last modification time).

Returns
A list of removed items.

dlpt.pth.**with_fw_slashes**(*path: str*) → str

Convert path to use forward slashes.

Note: This function does not do *os.path.normpath()* so it is also usable for UNCs.

Parameters
path – path to convert.

Returns
A path with converted back slashes to forward slashes.

dlpt.pth.**with_double_bw_slashes**(*path: str*) → str

Convert and return path to use double back slashes.

Parameters
path – path to convert

Returns
A converted path with double back slashes.

dlpt.pth.**get_name**(*file_path: str*, *with_ext: bool = True*) → str

Return a file name from file path or raise exception.

Note: No file existence check is performed.

Parameters

- **file_path** – file path where file name will be fetched from.
- **with_ext** – if False, extension is striped from file name.

Returns
A file name with/without extension.

dlpt.pth.**get_ext**(*file_path: str*) → str

Return file extension (with dot) from file path or raise exception.

Note: No file existence check is performed.

Parameters
file_path – file path where file name will be fetched from.

Returns
A file extension.

dlpt.pth.**get_files_in_dir**(*dir_path: str*, *include_ext: List[str] | None = None*, *exclude_ext: List[str] | None = None*) → List[str]

Get a list of files in a given `dir_path`.

Note: Only one of `include_ext` or `exclude_ext` must be set, or exception is raised. Lower case extension strings are compared.

Parameters

- **dir_path** – path to a directory to scan.
- **include_ext** – if set, only files with given extension(s) are returned.
- **exclude_ext** – if set, files with given extension(s) are excluded from return list.

Returns
List of matching files from *dir_path* `.

dlpt.pth.**get_files_in_dir_tree**(*dir_tree_path: str*, *include_ext: List[str] | None = None*, *exclude_ext: List[str] | None = None*) → List[str]

Same as *get_files_in_dir()*, but scan through all files in all directories.

Note: Only one of `include_ext` or `exclude_ext` must be set, or exception is raised. Lower case extension strings are compared.

Parameters

- **dir_tree_path** – path to a directory tree to scan.
- **include_ext** – if set, only files with given extension(s) are returned.
- **exclude_ext** – if set, files with given extension(s) are excluded from return list.

Returns
List of matching files from `dir_path` and all its sub-directories.

dlpt.pth.**get_dirs_in_dir**(*dir_path: str*, *name_filter: str | None = None*, *case_insensitive: bool = True*) → List[str]

Get a list of directories in a given `dir_path`.

Parameters

- **dir_path** – path to a directory to scan.
- **name_filter** – if set, directories that contain this string are returned, based on `case_insensitive` setting.
- **case_insensitive** – if True, lower-cased name_filter string (if set) is checked in lower case directory name.

Returns

List of matching directories from `dir_path`.

dlpt.pth.**open_in_web_browser**(*url: str*)

Open given address in a default web browser as a non-blocking subprocess.

Parameters

url – web address to open.

dlpt.pth.**open_with_default_app**(*file_path: str*)

Open given file with OS default application as a non-blocking subprocess.

Parameters

file_path – path to a file to open.

CHAPTER

SEVEN

DLPT.TIME MODULE

Utility functions to convert time in/to/from various formats and track execution time of a code.

dlpt.time.**timestamp_to_datetime**(*timestamp: float*) → datetime

Return a datetime object for a given `timestamp` (as returned by *time.time()*).

Parameters
timestamp – timestamp as a number since the epoch (*time.time()*).

Returns
Datetime object of a `timestamp`.

dlpt.time.**timestamp_to_str**(*timestamp: float*, *fmt: str = '%H:%M:%S'*, *msec_digits: int = 0*) → str

Return a string of converted timestamp (as returned by *time.time()*) by following the given format.

Parameters

- **timestamp** – timestamp as a number since the epoch (*time.time()*).
- **fmt** – output string format.
- **msec_digits** – See the docs check `_format_msec()`

Returns
Timestamp as a string, based on a given format.

dlpt.time.**sec_to_str**(*seconds: float*, *fmt: str = '%H h %M min %S sec'*, *hide_zeroes: bool = True*) → str

Return a string of a converted time (in seconds) by following the given format.

Note: Only applicable for hours, minutes and seconds. Days and larger time
units are silently ignored (added to the hours).

Note: Seconds are always displayed as a 2 digit float, while hours and
numbers are integers. Example: 2 days and 4 hours -> 52 h 0 min 0.00 sec

Parameters

- **seconds** – time (duration) as a number of seconds.
- **fmt** – output string format. This function does not support setting float number of digits for seconds. Output format can be changed if `hide_zeroes` arg is True.
- **hide_zeroes** – if True, leading parts (hours, minutes) can be omitted (if zero), Otherwise, `fmt` is strictly respected. Note: this is applicable only in the most common use cases, where time is displayed in order <hours> <minutes> <seconds>. If `hide_zeroes` is True, leading zero-value parts are stripped to the first next time part: hours to minutes, minutes to seconds. 361.5 sec = 1 h 0 min 1.50 sec 360 sec = 1 h 0 min 0.00 sec 359 sec = 59 min 0.00 sec 59 sec = 59.00 sec Other special time formaters can be used by setting `hide_zeroes` to False.

Returns
Formatted string of a given seconds number.

dlpt.time.**time_to_seconds**(*d: int = 0*, *h: int = 0*, *m: int = 0*, *s: float = 0.0*) → float

Return 'seconds' representation of a given time as defined by days, hours, minutes and seconds.

Parameters

- **d** – number of days to add to returned seconds.
- **h** – number of hours to add to returned seconds.
- **m** – number of minutes to add to returned seconds.
- **s** – number of seconds to add to returned seconds.

Returns
'Seconds' representation of a given time duration.

dlpt.time.**datetime_to_str**(*dt: datetime*, *fmt: str = '%H:%M:%S'*) → str

Return a string representation of a given `dt` datetime.datetime` object.

Note: `dt` is *datetime.datetime* object, not *datetime.timedelta* -
check *timedelta_to_str()*.

Parameters

- **dt** – datetime object to convert to string.
- **fmt** – output string format.

Returns
String representation of *datetime.datetime* object.

dlpt.time.**timedelta_to_str**(*td: timedelta*, *fmt: str = '%M min %S sec'*) → str

Return a string representation of a `td` *datetime.timedelta* object.

Note: receives *datetime.timedelta* object, not *datetime.datetime* - check
datetime_to_str().

Parameters

- **td** – datetime.timedelta object to convert to string.
- **fmt** – output string format. Respect output format - does not hide zeroes.

Returns
String representation of *datetime.timedelta* object.

dlpt.time.**get_current_datetime_str**(*fmt: str = '%d-%b-%Y_%H:%M:%S'*, *msec_digits: int = 0*) → str

Return a string of a current timestamp by following the given format.

Parameters

- **fmt** – output string format.
- **msec_digits** – check `_format_msec()`.

Returns
Formatted current date and time string.

dlpt.time.**print_exec_time**(*func: Callable[[...], T_EXEC_TIME]*) → Callable[[...], T_EXEC_TIME]

Decorator to get and print (to console) approximate execution time. Additionally, user can get execution time with *get_last_measured_time_sec()*.

Parameters
func – function reference to get execution time.

Example

```
>>> @dlpt.time.print_exec_time
    def my_function(*args, **kwargs):
        time.sleep(42)
>>> my_function()
"'my_function' execution time: 42.63 sec"
>>> dlpt.time.get_last_measured_time_sec()
42.63
```

dlpt.time.**func_stopwatch**(*func: Callable[[...], T_EXEC_TIME]*) → Callable[[...], T_EXEC_TIME]

Call function and track its execution time. Similar to a *print_exec_time()* decorator, but can be used with function with arguments. Does not print time to console.

Parameters
func – function 'pointer' to track execution time.

Example

```
>>> def my_function(*args, **kwargs):
        time.sleep(42)
>>> my_function_timed = dlpt.time.func_stopwatch(my_function)
>>> my_function_timed(arg1, arg2)
>>> dlpt.time.get_last_measured_time_sec()
42.63
```

Returns
User function wrapped in *func_stopwatch()*.

dlpt.time.**get_last_measured_time_sec**() → float

Return execution time of the last function, that was timed by using *print_exec_time()* or *func_stopwatch()* function.

Note: only valid after function calls. Otherwise, return None or a
previous time.

Returns
Last timed function or None (if no function was timed before).

CHAPTER

EIGHT

DLPT.UTILS MODULE

Various utility functions that simplify everyday code. Example: - converting numbers from/to string - comparing lists, dictionaries - code inspection - strings operations, - ...

dlpt.utils.**float_to_str**(*number: float*, *show_num_of_digits: int = 2*) → str

Convert float number with any number of decimal digits and return string with `show_num_of_digits` places after ..

Parameters

- **number** – float number to convert to string.
- **show_num_of_digits** – number of decimal places (characters) that are added/stripped after ..

dlpt.utils.**get_int_from_str**(*number: str*) → int

Return integer representation of a given number string. HEX number strings must start with *0x*. Negative numbers are also supported.

Parameters

number – int/float string representation of a given number.

dlpt.utils.**get_float_from_str**(*number: str*) → float

Return float representation of a given number string. HEX number strings must start with *0x*.

Parameters

number – int/float/string representation of a given number.

dlpt.utils.**get_list_intersection**(*l1: List[Any]*, *l2: List[Any]*) → List[Any]

Return intersection of a given lists.

Note: Operation does not necessary maintain items order!

Note: Only applicable for a lists with primitive types - nested lists will fail - can't set().

Parameters

- **l1** – first list to compare.
- **l2** – second list to compare.

dlpt.utils.**get_list_str**(*data: List[Any]*, *separator: str = ', '*) → str

Return a human readable list string (for printing purposes).

Parameters

- **data** – list to transform to string.
- **separator** – separator to join list items with.

dlpt.utils.**get_list_difference**(*l1: List[Any]*, *l2: List[Any]*) → List[Any]

Return difference (items that are unique just to a one of given lists) of a given lists.

Note: Operation does not necessary maintain items order!

Parameters

- **l1** – first list to compare.
- **l2** – second list to compare.

dlpt.utils.**remove_list_duplicates**(*data: List[Any]*) → List[Any]

Return a list of items without any duplicates.

Note: Operation does not necessary maintain items order!

Parameters

data – list with possibly duplicated items.

dlpt.utils.**search_str_in_lines**(*str_to_search: str*, *lines: List[str]*, *exact_match: bool = False*) → int | None

Return index of a first line where `str_to_search` string can be found.

Otherwise, return None.

Parameters

- **str_to_search** – string to search in lines.
- **lines** – list of strings, where `str_to_search` is searched.
- **exact_match** – if True, only exact `str_to_search` string is compared in `lines`. Otherwise, only string presence is checked.

dlpt.utils.**are_list_values_equal**(*l1: List[Any]*, *l2: List[Any]*) → bool

Return True if lists have the same values, False otherwise.

Note: Items order is not respected. If order must also be respected, just use list comparison: `l1 == l2`

Note: List items must be hashable.

Parameters

- **l1** – first list to compare.

- **l2** – second list to compare.

dlpt.utils.**are_dict_keys_equal**(*d1: Dict[Any, Any]*, *d2: Dict[Any, Any]*) → bool

Return True if dictionaries have the same keys, False otherwise.

Parameters

- **d1** – first dict to compare.
- **d2** – second dict to compare.

dlpt.utils.**are_dict_values_equal**(*d1: Dict[Any, Any]*, *d2: Dict[Any, Any]*) → bool

Return True if dicts have the same values, False otherwise.

Note: When comparing dict items that are not simple types (*int*, *str*, ...), function might return False if comparing different instances, regardles if object type is the same.

Parameters

- **d1** – first dict to compare.
- **d2** – second dict to compare.

dlpt.utils.**map_dict_to_class**(*obj: object*, *data: Dict[str, Any]*) → object

Return an object `obj` updated by the values of data dictionary.

Note: Only data keys, that match `obj` variable names are updated. Others are silently ignored.

Parameters

obj – object instance (class) whose values must be updated.

dlpt.utils.**get_obj_public_vars**(*obj: object*) → Dict[str, Any]

Return a dictionary of class variables that does not start with '_' or '__'.

Each item represents: '<variable name>': <variable value>

Note: If given `obj` is a class reference, only 'public' static variables are returned. If given `obj` is a class instance, 'public' static and instance variables are returned.

Args

obj: object (class) to inspect.

dlpt.utils.**get_obj_public_methods**(*obj: object*) → Dict[str, Callable[[...], Any]]

Return a dictionary of object public methods that does not start with '_' or '__'. Each item represents: '<method name>': < method reference>

Note: Only class 'public' methods are returned, without *@staticmethod*. They are of type '<bound method...>'

Parameters

obj – object to inspect.

dlpt.utils.**get_module_callables**(*module_instance: module*) → Dict[str, Callable[[...], Any]]

Return a dictionary of public methods that does not start with '_' or '__'. Each item represents: '<callable name>': <callable reference>

Parameters
module_instance – module object to inspect.

dlpt.utils.**get_module_public_classes**(*module_instance: module*) → Dict[str, Callable[[...], Any]]

Return a dictionary of public classes that does not start with '_' or '__'. Each item represents: '<class name>': <class reference>

Parameters
module_instance – module object to inspect.

dlpt.utils.**get_module_public_functions**(*module_instance: module*) → Dict[str, Callable[[...], Any]]

Get a list of references to all callable objects from a given module instance.

Parameters
module_instance – module object to inspect.

dlpt.utils.**get_caller_location**(*depth: int = 2*) → str

Return a function/location/line number of a caller function. Return string format: "<function()> @ <absolute file path>:<line number>"

Note: Using inspect module can be slow -> `inspect.getouterframes()` call `inspect.stack()` which actually read files.

Warning: While steping through code in a debug session, stack can be full of a debugger (example: 'ptvsd') entries.

Parameters
depth – stack frame depth inspection. The first (index = 0) entry in the returned list represents current frame; the last entry represents the outermost call on frame's stack.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

C

D

E

F

G

H

I

K

L

M

O

P

R

S

T

W