pytb

Apr 29, 2020

Contents:

This is a collection of useful snippets I find myself to use regularly during prototyping.

Most of the functions are especially useful when working on remote machines via jupyter notebooks (e.g. a Jupyter-Hub) with long-running processes (e.g. Deeplearning).

Checkout the *Quickstart* section for common usecases and example code.

View the complete documentation

View the code

CHAPTER 1

Quickstart

- Monitor long running tasks and get notified by email if something goes wrong or the job has finished
- Schedule task execution
- Debug Remotely over a TCP connection
- Load Jupyter Notebooks as Python Modules
- Reload modules when importing again (do not cache the result)
- Mirroring all output of a script to a file
- Flexibly test a number possible configurations of a function
- Automatically configure the framework
- Configure defaults

CHAPTER 2

Installation

via pip:

> pip install py-toolbox

or via distutils:

```
git clone https://github.com/dangrie158/py-toolbox.git pytb
cd pytb
python setup.py install
```

CHAPTER 3

Development

Clone the repo and install the development requirements. After this you can install the package in development mode to just link the sources into your python path.

```
git clone https://github.com/dangrie158/py-toolbox.git pytb
cd pytb
direnv allow
# if you're not using direnv, you really should
# otherwise create a new virtualenv for the package

pip install -r dev-requirements.txt
python3 setup.py develop

make test
```

3.1 Command Line Interface

The toolkit can be run as an executable (e.g. using the `-m` switch of the python command or by using the automatically created `pytb` command)

3.1.1 Task Scheduler `pytb schedule`

Command line interface for the *schedule module*.

Currently only the `--at` mode is supported using a cron-like syntax.

The cron-like pattern has the following order: min hour day month weekday. The following pattern rules are supported:

- `i` sequence contains only the element i
- `*` indicates that all values possible for this part are included
- `i,j,k` specifies a list of possible values

- `i-j` specifies a range of values *including* `j`
- `i-j/s` additionally specifies the step-size

For weekday, the values `0` and `7` both represent sunday.

```
usage: pytb schedule [-h] [--at * * * * *] script ...

positional arguments:
script          script path or module name to run
args            additional parameter passed to the script

optional arguments:
-h, --help      show this help message and exit
--at * * * * *  Execute the task each time the cron-like pattern matches
```

3.1.2 Notifications for long running scripts

Command line interface for the *notification module*.

You can choose a build-in notifier via the `via-email` and `via-stream` switches. Notification rules can be configured via the `--when-done`, `--when-stalled` and `--every` options.

```
usage: pytb notify [-h] [--every X] [--when-stalled X] [--when-done]
                {via-email,via-stream} ...

positional arguments:
{via-email,via-stream}
                        notifier

optional arguments:
-h, --help            show this help message and exit
--every X             Send a notification every X seconds
--when-stalled X      Send a notification if the script seems to be stalled
                        for more than X seconds
--when-done           Send a notification whenever the script finishes
```

E-Mail Notifier

```
usage: pytb notify via-email [-h] [--recipients RECIPIENTS [RECIPIENTS ...]]
                            [--smtp-host SMTP_HOST] [--smtp-port SMTP_PORT]
                            [--sender SENDER] [--use-ssl] [-m]
                            script ...

positional arguments:
script                script path or module name to run
args                  additional parameter passed to the script

optional arguments:
-h, --help            show this help message and exit
--recipients RECIPIENTS [RECIPIENTS ...]
                        Recipient addresses for the notifications
--smtp-host SMTP_HOST
                        Address of the external SMTP Server used to send
                        notifications via E-Mail
```

(continues on next page)

(continued from previous page)

```
--smtp-port SMTP_PORT
                        Port the external SMTP Server listens for incoming
                        connections
--sender SENDER       Sender Address for notifications
--use-ssl             Use a SSL connection to communicate with the SMTP
                        server
-m                    Load an executable module or package instead of a file
```

Note: If you want to specify multiple recipients as the last option in your command line, use -- to seperate the argument list from the script option with multiple arguments.

Example:

```
pytb notify --when-done --when-stalled 5 via-email --recipients recipient1@mail.com␣
↪recipient2@mail.com -- myscript.py param1 --param2=val
```

Stream Notifier

```
usage: pytb notify via-stream [-h] [--stream STREAM] [-m] script ...

positional arguments:
script           script path or module name to run
args             additional parameter passed to the script

optional arguments:
-h, --help       show this help message and exit
--stream STREAM  The writable stream. This can be a filepath or the special
                values `<stdout>` or `<stderr>`
-m               Load an executable module or package instead of a file
```

Note: If you want to use the `stdout` or `stderr` stream as output, simply use the constants `<stdout>` or `<stderr>` for the stream parameter. If your shell tries to replace those values (e.g. `zsh`), quote the strings.

Example:

```
python -m pytb notify --every 5 via-stream --stream="<stdout>" -m http.server
```

3.1.3 Remote Debugger `pytb rdb`

A simple command line interface for the remote debugger rdb. The subcommand expects a function parameter which should be either `client` or `server`.

The `server` function exposes a similar interface to the original `pdb` command line. Additionally you can specify the interface and port to bind to and listening for incoming connections as well as the verbosity of the debug server.

```
usage: pytb rdb server [-h] [--host HOST] [--port PORT] [--patch-stdio]
                      [-c commands] [-m]
                      script ...

positional arguments:
script         script path or module name to run
args           additional parameter passed to the script

optional arguments:
```

(continues on next page)

(continued from previous page)

```
-h, --help      show this help message and exit
--host HOST     The interface to bind the socket to
--port PORT     The port to listen for incoming connections
--patch-stdio   Redirect stdio streams to the remote client during debugging
-c commands     commands executed before the script is run
-m              Load an executable module or package instead of a file
```

More information on the `-c` and `-m` parameters can be found in the pdb Module Documentation

The `client` function creates a new *`pytb.rdb.RdbClient`* instance that connects to the specified host and port.

```
usage: pytb rdb client [-h] [--host HOST] [--port PORT]

optional arguments:
-h, --help   show this help message and exit
--host HOST  Remote host where the debug sessino is running
--port PORT  Remote port to connect to
```

Both functions fall back to the values provided in the effective .pytb.conf file (see *`pytb.config.Config`*) for the `--host`, `--port` and `--patch-stdio` parameters

Example usage:

Start a debug server listening on the interface and port read from the .pytb.conf file. This command does not start script execution until a client is connected:

```
pytb rdb server -c continue myscript.py arg1 arg2 --flag
```

From another terminal (possibly on another machine) connect to the session. Since we passed the 'continue' command when starting the server, the script will be executed until the end or to the first unhandled exception as soon as the client connects. Without this, script execution would be stopped before the first line is executed and the client would be presented with a debug shell. Because we do not specify a `--port` argument, the default port pecified in the config file is used.

```
python -m pytb rdb client --host 192.168.1.15
```

3.2 pytb.config module

3.2.1 Configure the Toolkit

Some modules use a configuration based on a config-file hierarchy. This hierarchy starts in the current working directory moving to parent folders until the root of the filesystem is reached.

The hierarchy is then traversed in reverse order and in each folder, a file named `.pytb.conf` is loaded if available. The API docs for the module reference when a configuration is used. The function of the config parameters is documented in the *Default Config*

3.2.2 Default Config

The pytb package provides a config file with sane default values that should work in most cases. The file is loaded as first file overwriting the hard-coded defaults in the *`pytb.config.Config`* class but being overwritten by any more-specific config file in the lookup hierarchy.

```
# default configuration values for the toolkit

# configures the behaviour of pytb.init()
[init]
# disable the module chache after initialisation
disable_module_cache = no

# install the notebook loader hook into the import system
install_notebook_loader = yes

# install rdb as default debugger called when calling the built-in 'breakpoint()'
install_rdb_hook = yes


# remote debugger
[rdb]
# the default port the debugger listens on and the client connects to
port = 8268

# bind address for the debug server
bind_to = 0.0.0.0

# address the client tries to connect to
host = 127.0.0.1

# whether or not to redirect the servers stdio streams to the debugging client
patch_stdio = no

#
[module_cache]

# these packages are exempt from reloading
#
# it does not make much sense to reload built-ins. Additionally there
# are some modules in the stdlib that do not like to be reloaded and
# throw an error, so we exclude them here as they do not make sense
# to live-reload them anyway
non_reloadable_packages =
    re
    importlib
    pkg_resources
    jsonschema
    numpy
    IPython

# automatic task progress notification via E-Mail
[notify]
# smtp server setup used to send notifications to the user
smtp_host = 127.0.0.1
smtp_port = 25
smtp_ssl = False

# sender address to use. If empty, use the machines FQDN
sender =

# a list of email-addresses where notifications are sent
email_addresses =
```

3.2.3 API Documentation

This module handles the .pytb.conf files

class `pytb.config.`**`Config`**`(`*verbose: Optional[bool] = False*`)`

Bases: `configparser.ConfigParser`

Provides functionality to load a hierarchy of `.pytb.config` files.

Parameters **`verbose`** – output debugging information including the paths that will be checked for config files as well as all files that are actually parsed

`config_file_name = '.pytb.conf'`

filename of config files

`default_config_file = PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/py-t`

The loaction of the default config file (in the root of the package)

`static get_config_file_locations``()` → Sequence[pathlib.Path]

Get a list of possible configuration file paths by starting at the current working directory and add all parent paths until the root directory is reached

The default config file location is always the first path in the list. More specific configuration files should appear later in the list (from unspecific to more specific)

`getlist``(`**args*, ***kwargs*`)` → Sequence[str]

get a list of values that are seperated by a newline character

```
>>> config = Config()
>>> config.read_string("""
...     [test]
...     list=a
...         b
...         c
...     """)
>>> config.getlist('test', 'list')
['a', 'b', 'c']
```

`reload``()` → None

load the configuration by initialising the default values from *Config._defaults* and then traversing all possible configuration files overwriting all newly found values

`pytb.config.`**`current_config = <pytb.config.Config object>`**

An instance of config that is automatically initialized when importing the module

3.3 pytb.core module

3.3.1 Autoconfigure toolbox frameworks

The `pytb` package provides an `init()` method that allows to automatically configure certain frameworks from the toolbox.

The behavior of this method is configured based on the values in *`pytb.config.current_config`*.

The method has a parameter `verbose` which defaults to `True` which enables some output while initializing the subsystems. If you want to quietly initialize, set *verbose* explicitly to *False*.

To avoid problems with multiple initializations, the method raises a `RuntimeException` if init is called a second time.

```
>>> from pytb.core import init
>>> init()
'disable_module_cache' not set, skipping global context
installing NotebookLoader into 'sys.meta_path'
installing RDB as default debugger in 'sys.breakpointhook'
```

3.3.2 API Documentation

3.4 pytb.importlib module

3.4.1 Importing Jupyter-Notebooks as python modules

```
>>> from pytb.importlib import no_module_cache, NotebookLoader
>>> loader = NotebookLoader()
>>> loader.install()
>>> # will try to import the Notebook in './my/Notebook.ipynb'
>>> import pytb.test.fixtures.Notebook
>>> loader.uninstall()
```

NotebookLoaders can also be used as ContextManagers to only temporarly affect module loading and automatically remove the loader hook when exiting the context.

```
>>> from pytb.importlib import no_module_cache, NotebookLoader
>>> with NotebookLoader():
...     import pytb.test.fixtures.Notebook # will load the notebook
>>> # next line will fail if there is no package named 'my'
>>> import pytb.test.fixtures.Notebook
```

3.4.2 Automatically reload modules and packages when importing

This is especially useful in combination with a Notebook Loader. You can simply run an import cell again to reload the Notebook Code from disk.

Use a `NoModuleCacheContext` to force reloading of modules that are imported. An instance of the ContextManager is available as `pytb.importlib.no_module_cache`.

Some packages can not be reloaded as they define a global state that does not like to be created again. The default config defines a sane set of packages that are ignored by the reloader.

```
>>> from pytb.importlib import no_module_cache, NotebookLoader
>>> loader = NotebookLoader().install()
>>> # load the module if it was not previously loaded
>>> import pytb.test.fixtures.Notebook
>>> with no_module_cache:
...     # force reevaluation of the module (will execute all code again)
...     import pytb.test.fixtures.Notebook
```

3.4.3 API Documentation

3.5 pytb.io module

3.5.1 Redirecting output streams

The `io` Module offers function to temporarly redirect or mirror `stdout` and `stderr` streams to a file

Stream redirection:

```
>>> from pytb.io import redirected_stdout
>>> with redirected_stdout('stdout.txt'):
...     print('this will be written to stdout.txt and not to the console')
```

Stream mirroring

```
>>> from pytb.io import mirrored_stdstreams
>>> with mirrored_stdstreams('alloutput.txt'):
...     print('this will be written to alloutput.txt AND to the console')
```

3.5.2 API Documentation

This module contains a set of helpers for common Input/Output related tasks

class pytb.io.**Tee**(**args*)

Bases: `object`

A N-ended T-piece (manifold) for File objects that supports writing. This is useful if you want to write to multiple files or file-like objects (e.g. `sys.stdout`, `sys.stderr`) simultaneously.

```
>>> import sys, io
>>> file_like = io.StringIO()
>>> combined = Tee(file_like, sys.stdout)
>>> _ = combined.write('This is printed into a file and on stdout\n')
This is printed into a file and on stdout
>>> assert file_like.getvalue() == 'This is printed into a file and on stdout\n'
```

close() → None

Close all connected files

This does avoid closing `sys.__stdout__` and `sys.__stderr__`

```
>>> import sys, io
>>> file_like = io.StringIO()
>>> combined = Tee(file_like, sys.__stdout__)
>>> file_like.closed
False
>>> combined.close()
>>> file_like.closed
True
>>> sys.stdout.closed
False
```

flush() → None
Flush any buffers of all connected file output-streams

write(*text: str*) → int
Write to the manifold which, in turn, writes to all connected output streams

Parameters **text** – text to write to the manifold

Returns the number of bytes written to the last stream in the Manifold

pytb.io.**mirrored_stdout**(*file: Union[str, TextIO, pytb.io.Tee]*) → Generator[TextIO, None, None]
ContextManager that mirrors stdout to a given file-like object and restores the original state when leaving the context

This is essentially using a `Tee` piece manifold to `file` and `sys.stdout` as a parameter to `redirected_stdout`

Parameters **file** – string or file-like object to mirror stdout to. If passed a string, the file is opened for writing and closed after the contextmanager exits

```
>>> import io
>>> outfile = io.StringIO()
>>> with mirrored_stdout(outfile):
...     print('this is written to outfile and stdout')
this is written to outfile and stdout
>>> assert outfile.getvalue() == 'this is written to outfile and stdout\n'
```

pytb.io.**mirrored_stdstreams**(*file: Union[str, TextIO, pytb.io.Tee]*) → Generator[TextIO, None, None]
Version of *mirrored_stdout()* but mirrors `stderr` and `stdout` to file

see *mirrored_stdout()*

pytb.io.**redirected_stderr**(*file: Union[str, TextIO, pytb.io.Tee]*) → Generator[TextIO, None, None]
Same functionality as `redirect_stdout` but redirects the stderr stram instead

see *redirected_stdout()*

pytb.io.**redirected_stdout**(*file: Union[str, TextIO, pytb.io.Tee]*) → Generator[TextIO, None, None]
ContextManager that redirects stdout to a given file-like object and restores the original state when leaving the context

Parameters **file** – string or file-like object to redirect stdout to. If passed a string, the file is opened for writing and closed after the contextmanager exits

```
>>> import io
>>> outfile = io.StringIO()
>>> with redirected_stdout(outfile):
...     print('this is written to outfile')
>>> assert outfile.getvalue() == 'this is written to outfile\n'
```

pytb.io.**redirected_stdstreams**(*file: Union[str, TextIO, pytb.io.Tee]*) → Generator[TextIO, None, None]
redirects both output streams (`stderr` and `stdout`) to `file`

see *redirected_stdout()*

pytb.io.**render_text**(*text: str, maxwidth: int = -1*) → str
Attempt to render a text like an (potentiall infinitely wide) terminal would.

Thus carriage-returns move the cursor to the start of the line, so subsequent characters overwrite the previous.

```
>>> render_text('asd\rbcd\rcde\r\nqwe\rert\n123', maxwidth=2)
'cd\ne \ner\nt \n12\n3'
```

Parameters

- **text** – Input text to render
- **maxwidth** – if > 0, wrap the text to the specified maximum length using the textwrapper library

3.6 pytb.itertools module

3.6.1 Flexibly test a number possible configurations of a function

Assume you have a function that takes a number of parameters:

```
>>> def my_func(a, b, c=2, **kwargs):
...     print(' '.join((a, b, c)), kwargs)
```

And you want to call it with multiple parameter combinations

```
>>> my_params = {
...     'a': 'a1',
...     'b': ('b1','b2'),
...     'c': ('c1', 'c2'),
...     'additional_arg': 'val'
... }
```

You can use the named_tuple() function of this module to create any possible combination of the provided parameters

```
>>> for params in named_product(my_params):
...     my_func(**params)
a1 b1 c1 {'additional_arg': 'val'}
a1 b1 c2 {'additional_arg': 'val'}
a1 b2 c1 {'additional_arg': 'val'}
a1 b2 c2 {'additional_arg': 'val'}
```

Excluding some combinations

If some parameter combinations are not allowed, you can use the functions ability to work with nested dicts to overwrite values defined in an outer dict

```
>>> my_params = {
...     'a': 'a1',
...     'b': ('b1','b2'),
...     'c': {
...         'c1': {'b': 'b1'},
...         'c2': {},
...         'c3': {
...             'additional_arg': 'other val',
...             'another arg': 'yet another val'}
...     },
```

(continues on next page)

(continued from previous page)

```
...     'additional_arg': 'val'
... }
```

```
>>> for params in named_product(my_params):
...     my_func(**params)
a1 b1 c1 {'additional_arg': 'val'}
a1 b1 c2 {'additional_arg': 'val'}
a1 b2 c2 {'additional_arg': 'val'}
a1 b1 c3 {'additional_arg': 'other val', 'another arg': 'yet another val'}
a1 b2 c3 {'additional_arg': 'other val', 'another arg': 'yet another val'}
```

Note that for `c='c1'` only `b='b1'` was used. You can also define new variables inside each dict that only get used for combinations in this branch.

`safe_copy` of values

By default, all values are (deep) copied before they are yielded from the generator. This is really useful, as otherwise any change you make to an object in any combination would change the object for all other combination.

If you have large objects in your combinations however, copying may be expensive. In this case you can use the `safe_copy` parameter to control *if* and *which* objects should be copied before yielding.

3.6.2 API Documentation

Methods to work with iterables conveniently. (methods that could be in the python stdlib itertools package)

`pytb.itertools.`**`named_product`**(*values: Optional[Mapping[Any, Any]] = None, repeat: int = 1, safe_copy: Union[Sequence[str], bool] = True, **kwargs*) → Generator[Any, None, None]

Return each possible combination of the input parameters (cartesian product), thus this provides the same basic functionality of :meth:`itertools.product`. However this method provides more flexibility as it:

1. returns dicts instead of tuples

```
>>> list(named_product(a=('X', 'Y'), b=(1, 2)))
[{'a': 'X', 'b': 1}, {'a': 'X', 'b': 2}, {'a': 'Y', 'b': 1}, {'a': 'Y', 'b': 2}]
```

2. accepts either a dict or kwargs

```
>>> list(named_product({ 'a':('X', 'Y') }, b=(1, 2)))
[{'a': 'X', 'b': 1}, {'a': 'X', 'b': 2}, {'a': 'Y', 'b': 1}, {'a': 'Y', 'b': 2}]
```

3. accepts nested dicts

```
>>> list(named_product(
...     a=(
...             {'X': {'b':(1,2)}},
...             {'Y': {
...                     'b': (3, 4),
...                     'c': (5, )
...                 }
...             }
```

(continues on next page)

(continued from previous page)

```
...     )
... ))
[{'a': {'X': {'b': (1, 2)}}}, {'a': {'Y': {'b': (3, 4), 'c': (5,)}}}]
```

4. accepts scalar values

```
>>> list(named_product(b='Xy', c=('a', 'b')))
[{'b': 'Xy', 'c': 'a'}, {'b': 'Xy', 'c': 'b'}]
```

Parameters

- **values** – a dict of iterables used to create the cartesian product
- **repeat** – repeat iteration of the product N-times
- **safe_copy** – copy all values before yielding any combination. If passed `True` all values are copied. If passed `False` no values are copied. If passed an iterable of strings, only the values whose key is in the iterable are copied.
- ****kwargs** – optional keyword arguments. The dict of keyword arguments is merged with the values dict, with `kwargs` overwriting values in `values`

3.7 pytb.notification module

3.7.1 Setup monitoring for your long running tasks

Automatic task progress and monitoring notification via E-Mail. Especially useful to supervise long-running code blocks.

Concrete Notifier implementations

The base `Notify` class is an abstract class that implements the general notification management. However, it does not define how notifications are delivered.

The Framework implements different derived classes with a concrete implementation of the abstract `Notify._send_notification()` method:

NotifyViaEmail Send notifications as emails using an external SMTP server.

NotifyViaStream Write notifications as string to a stream. The stream can be any writable object (e.g. a TCP or UNIX socket or a *io.StringIO* instance). When using pythons socket module, use the sockets `makefile()` method to get a writable stream.

Manually sending Notifications

You can send Notification manually using the `now()` method:

```
>>> stream = io.StringIO()
>>> notify = NotifyViaStream("testtask", stream)
>>> # set a custom template used to stringify the notifications
>>> notify.notification_template = "{task} {reason} {exinfo}"
>>> notify.now("test successful")
```

(continues on next page)

(continued from previous page)

```
>>> stream.getvalue()
'testtask test successful '
```

Notify when a code block exits (on success or failure)

The `when_done()` method can be used to be notified when a code block exits. This method will always send exactly one notification when the task exits (gracefully or when a unhandeled exception is thrown) except if the `only_if_error` parameter is `True`. In this case a graceful exit will not send any notification.

```
>>> _ = stream.truncate(0)
>>> _ = stream.seek(0)
>>> with notify.when_done():
...     # potentially long-running process
...     pass
>>> stream.getvalue()
'testtask done '
```

When an exception occurs, the `{exinfo}` placeholder is populated with the exception message. The exception is reraised after the notification is sent and the context exited.

```
>>> _ = stream.seek(0)
>>> with notify.when_done():
...     raise Exception("ungraceful exit occurred")
Traceback (most recent call last):
    ...
Exception: ungraceful exit occurred

>>> stream.getvalue()
'testtask failed ungraceful exit occurred'
```

Periodic Notifications on the Progress of long-running Tasks

The `every()` method can be used to send out periodic notifications about a tasks progress. If the `incremental_output` parameter is `True` only the newly generated output since the last notification is populated into the `{output}` placeholder.

```
>>> _ = stream.seek(0)
>>> notify.notification_template = "{task} {reason} {output}\n"

>>> with notify.every(0.1, incremental_output=True):
...     time.sleep(0.12)
...     print("produced output")
...     time.sleep(0.22)
>>> print(stream.getvalue().strip())
testtask progress update <No output produced>
testtask progress update produced output
testtask progress update <No output produced>
testtask done produced output
```

Notify about stalled code blocks

Often you want to be notified if your long-running task may have stalled. The `when_stalled()` method tries to detect a stall and sends out a notification.

A stall is detected by checking the output produced by the code block. If for a specified `timeout` no new output is produced, the code is considered to be stalled. If a stall was detected, any produced output will send another notification to inform about the continuation.

```
>>> _ = stream.truncate(0)
>>> _ = stream.seek(0)
>>> notify.notification_template = "{task} {reason}\n"

>>> with notify.when_stalled(timeout=0.1):
...     time.sleep(0.2)
...     print("produced output")
...     time.sleep(0.1)
>>> print(stream.getvalue().strip())
testtask probably stalled
testtask no longer stalled
```

Notify after any iteration over an Iterable

Simply wrap any `Iterable` in `Notify.on_iteration_of()` to get notified after each step of the iteration has finished.

```
>>> _ = stream.truncate(0)
>>> _ = stream.seek(0)
>>> notify.notification_template = "{reason}\n"

>>> for x in notify.on_iteration_of(range(5), after_every=2):
...     pass
>>> print(stream.getvalue().strip())
Iteration 2/5 done
Iteration 4/5 done
Iteration 5/5 done
```

Note: Because of how generators work in python , it is not possible to handle exceptions that are raised in the loop body. If you want to get notified about errors that occurred during the loop execution, you need to wrap the whole loop into a `when_done()` context with the `only_if_error` flag set to `True`.

```
>>> _ = stream.truncate(0)
>>> _ = stream.seek(0)
>>> notify.notification_template = "{reason}\n"

>>> for x in notify.on_iteration_of(range(5)):
...     if x == 1:
...         raise Exception("no notification for this :(")
Traceback (most recent call last):
    ...
Exception: no notification for this :(

>>> print(stream.getvalue().strip())
Iteration 1/5 done
```

3.7.2 API Documentation

Automatic task progress and monitoring notification via E-Mail. Especially useful to supervise long-running tasks

class `pytb.notification.`**`Notify`**(*task: str, render_outputs: bool = True*)
    Bases: `object`

A `Notify` object captures the basic configuration of how a notification should be handled.

The methods `when_done()`, `every()` and `when_stalled()` are reenterable context managers. Thus a single `Notify` object can be reused at several places and different context-managers can be reused in the same context.

Overwrite the method `_send_notification()` in a derived class to specify a custom handling of the notifications

Parameters

- **`task`** – A short description of the monitored block.
- **`render_outputs`** – If true, prerender the oputputs using `pytb.io.render_text()` This may be useful if the captured codeblock produces progressbars using carriage returns

`every`(*interval: Union[int, float, datetime.timedelta], incremental_output: bool = False, caller_frame: Optional[frame] = None*) → Generator[None, None, None]

Send out notifications with a fixed interval to receive progress updates. This contextmanager wraps a `when_done()`, so it is guaranteed to send to notify at least once upon task completion or error.

Parameters

- **`interval`** – `float`, `int` or `datetime.timedelta` object representing the number of seconds between notifications
- **`incremental_output`** – Only send incremental output summaries with each update. If `False` the complete captured output is sent each time
- **`caller_frame`** – the stackframe to use when determining the code block for the notification. If None, the stackframe of the line that called this function is used

`now`(*message: str*) → None

Send a manual notification now. This will use the provided `message` as the `reason` placeholder. No output can be capured using this function.

Parameters **`message`** – A string used to fill the `{reason}` placeholder of the notification

`on_iteration_of`(*iterable: Sequence[_IterType], capture_output: bool = True, after_every: int = 1, caller_frame: Optional[frame] = None*) → Generator[_IterType, None, None]

Send a message *after* each iteration of an iterable. The current iteration and total number of iterations (if the iterable implements `len()`) will be part of the `reason` placeholder in the notification.

```
for x in notify.on_iteration_of(range(5)):
    # execute some potentially long-running process on x
```

Parameters

- **`iterable`** – the iterable which items will be yielded by this generator
- **`capture_output`** – capture all output to the `stdout` and `stderr` stream and append it to the notification
- **`after_every`** – Only notify about each N-th iteration
- **`caller_frame`** – the stackframe to use when determining the code block for the notification. If None, the stackframe of the line that called this function is used

`when_done`(*only_if_error: bool = False*, *capture_output: bool = True*, *caller_frame: Optional[frame] = None*, *reason_prefix: str = ''*) → Generator[None, None, None]

Create a context that, when exited, will send notifications. If an unhandled exception is raised during execution, a notification on the failure of the execution is sent. If the context exits cleanly, a notification is only sent if `only_if_error` is set to `False`

By default, all output to the `stdio` and `stderr` streams is captured and sent in the notification. If you expect huge amounts of output during the execution of the monitored code, you can disable the capturing with the `capture_output` parameter.

To not spam you with notification when you stop the code execution yourself, `KeyboardInterrupt` exceptions will not trigger a notification.

Parameters

- **`only_if_error`** – if the context manager exits cleanly, do not send any notifications
- **`capture_output`** – capture all output to the `stdout` and `stderr` stream and append it to the notification
- **`caller_frame`** – the stackframe to use when determining the code block for the notification. If None, the stackframe of the line that called this function is used
- **`reason_prefix`** – an additional string that is prepended to the `reason` placeholder when sending a notification. Used to implement *`on_iteration_of()`*.

`when_stalled`(*timeout: Union[int, float, datetime.timedelta], capture_output: bool = True, caller_frame: Optional[frame] = None*) → Generator[None, None, None]

Monitor the output of the code bock to determine a possible stall of the execution. The execution is considered to be stalled when no new output is produced within `timeout` seconds.

Only a single notification is sent each time a stall is detected. If a stall notification was sent previously, new output will cause a notification to be sent that the stall was resolved.

Contrary to the *`every()`* method, this does not wrap the context into a *`when_done()`* function, thus it may never send a notification. If you want, you can simply use the same *`Notify`* to create mutliple contexts:

```
with notify.when_stalled(timeout), notify.when_done():
    # execute some potentially long-running process
```

However, it will always send a notification if the code block exits with an exception.

Parameters

- **`timeout`** – maximum number of seconds where no new output is produced before the code block is considiered to be stalled
- **`capture_output`** – append all output to `stdout` and `stderr` to the notification
- **`caller_frame`** – the stackframe to use when determining the code block for the notification. If None, the stackframe of the line that called this function is used

class `pytb.notification.`**`NotifyViaEmail`**(*task: str, email_addresses: Optional[Sequence[str]] = None, sender: Optional[str] = None, smtp_host: Optional[str] = None, smtp_port: Optional[int] = None, smtp_ssl: Optional[bool] = None*)

Bases: *`pytb.notification.Notify`*

A *`NotifyViaEmail`* object uses an SMTP connection to send notification via emails. The SMTP server is configured either at runtime or via the effective `.pytb.config` files `notify` section.

Parameters

- **`email_addresses`** – a single email address or a list of addresses. Each entry is a seperate recipient of the notification send by this `Notify`
- **`task`** – A short description of the monitored block.
- **`sender`** – Sender name to use. If empty, use this machines FQDN

- **smtp_host** – The SMTP servers address used to send the notifications
- **smtp_port** – The TCP port of the SMTP server
- **smtp_ssl** – Whether or not to use SSL for the SMTP connection

All optional parameters are initialized from the effective `.pytb.config` if they are passed `None`

message_template = 'Hello {recipient},\n{task} {reason}. {exinfo}\n\n{code_block}\n\np

The message template used to create the message content.

You can customize it by overwriting the instance-variable or by deriving your custom *NotifyViaEmail*.

The following placeholders are available:

- `task`
- `sender`
- `recipient`
- `reason`
- `exinfo`
- `code_block`
- `output`

subject_template = '{task} on {sender} {reason}'

The template that is used as subject line in the mail.

You can customize it by the same techniques as the *message_template*. The same placeholders are available.

class pytb.notification.**NotifyViaStream**(*task: str, stream: IO[Any]*)

Bases: *pytb.notification.Notify*

NotifyViaStream will write string representations of notifications to the specified writable `stream`. This may be useful when the stream is a UNIX or TCP socket.

Also useful when when the stream is a `io.StringIO` object for testing.

The string representation of the notification can be configured via the *notification_template* attribute which can be overwritten on a per-instance basis.

Parameters

- **task** – A short description of the monitored block.
- **stream** – A writable stream where notification should be written to.

notification_template = '{task}\t{reason}\t{exinfo}\t{output}\n'

The string that is written to the stream after replacing all placeholders with the notifications properties.

The following placeholders are available

- `task`
- `reason`
- `exinfo`
- `code_block`
- `output`

class pytb.notification.**Timer**(*target: Any*, **args*, ***kwargs*)

Bases: `threading.Thread`

A gracefully stoppable Thread with means to run a target function repedatley every `X` seconds.

Parameters

- **target** – the target function that will be executed in the thread
- ***args** – additional positional parameters passed to the target function
- ****kwargs** – additional keyword parameters passed to the target function

call_every(*interval: Union[int, float, datetime.timedelta]*) → None

start the repeated execution of the target function every `interval` seconds. The target function is first invoked after waiting the interval. If the thread is stopped before the first interval passed, the target function may never be called

Parameters **interval** – `float`, `int` or `datetime.timedelta` object representing the number of seconds between invocations of the target function

run() → None

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop() → None

schedule the thread to stop. This is only meant to be used to stop a repeated scheduling of the target funtion started via `call_every()` but will not interrupt a long-running target function

Raises **RuntimeError** – if the thread was not started via `call_every()`

3.8 pytb.rdb module

3.8.1 Remote Debugging

Sometimes you do not have a nice way to control the debugger on the local machine. For example in a jupyter notebook the readline interface is horrible to use.

The debugger is designed to act as a drop-in replacement for the standard `pdb` debugger. However, when starting a debug session (e.g. using `set_trace()`) the debugger opens a socket and listens on the specified interface and port for a client.

The client can either be a simple TCP socket tool (like `netcat`) or the provided `RdbClient`.

The debugger can be invoked by calling `set_trace()`

3.8.2 API Documentation

A remote debugging module for the python debugger pdb

class pytb.rdb.**RdbClient**(*host: Optional[str] = None*, *port: Optional[int] = None*)

Bases: `object`

A simple `netcat` like socket client that can be used as a convenience wrapper to connect to a remote debugger session.

If *host* or *port* are unspecified, they are laoded from the current `pytb.config.Config` s *[rdb]* section

pytb.rdb.**install_hook**() → None
Installs the remote debugger as standard debugging method and calls it when using the builtin *breakpoint()*

pytb.rdb.**set_trace**(**args*, *host: Optional[str] = None*, *port: Optional[int] = None*, *patch_stdio: Optional[bool] = None*, ***kwargs*) → None
Opens a remote PDB on the specified host and port if no session is running. If a session is already running (was started previously and a client is still connected) the session is reused instead.

Parameters **patch_stdio** – When true, redirects stdout, stderr and stdin to the remote socket.

pytb.rdb.**uninstall_hook**() → None
Restore the original state of sys.breakpointhook. If *install_hook()* was never called before, this is a noop

3.9 pytb.schedule module

3.9.1 Run commands on certain dates with a cron-like date expression

You can use the `at()` and `every()` as a decorator for to turn that function into a scheduled thread object. Use the `start_schedule()` start the scheduled execution.

You can use the `stop()` to stop any further scheduling. The decorated function is also still directly callable to execute the task in the calling thread.

3.9.2 API Documentation

A simple task scheduling system to run periodic tasks

class pytb.schedule.**Schedule**(*target: Callable[[...], Any]*, *interval: Generator[datetime.datetime, None, None]*)
Bases: `threading.Thread`

This represents a reoccuring task, exceuting `target` every time the schedule is due. The target is run in an extra thread by default. If you stop the schedule while the target function is running, the thread is canceled after finishing its current run.

Parameters

- **target** – The target function to execute each time the schedule is due
- **interval** – A generator yielding datetime objects that determine when the schedule is due. When this generator is exhausted, the schedule stops. Datetime objects in the past are simply ignored and the next value from the generator is used to schedule the job.

next_schedule() → datetime.datetime
Return the datetime object this schedule is due

run() → None
Start the schedule execution in an extra thread. The target function is called, passing all arguments supplied to this call.

start_schedule(**args*, ***kwargs*) → None
Start the scheduler and pass all supplied arguments to the target function each time the schedule is due

stop() → None
Stop the async execution of the schedule, cacnel all future tasks

`pytb.schedule.`**`at`**(*minute: str = '*', hour: str = '*', day: str = '*', month: str = '*', weekday: str = '*'*) → Callable[[...], pytb.schedule.Schedule]

run the task every time the current system-time matches the cron-like expression. Check the documentation for `parse_cron_spec()` for the supported syntax.

`pytb.schedule.`**`every`**(*interval: datetime.timedelta, start_at: Optional[datetime.datetime] = None*) → Callable[[...], pytb.schedule.Schedule]

Run a task repeadetly at the given interval

Parameters

- **`interval`** – run the command this often the most
- **`start_at`** – run the command for the first time only after this date has passed. If not specified, run the command immediatley

`pytb.schedule.`**`parse_cron_spec`**(*spec: str, max_value: int, min_value: int = 0*) → Sequence[int]

Parse a string of in a cron-like expression format to a sequence accepted numbers. The expression needs to have one of the following forms:

- `i` sequence contains only the element i
- `*` indicates that all values possible for this part are included
- `i,j,k` specifies a list of possible values
- `i-j` specifies a range of values *including* `j`
- `i-j/s` additionally specifies the step-size

Parameters

- **`spec`** – The cron-like expression to parse
- **`max_value`** – The maximum value allowed for this range. This is needed to specify the range using the '*' wildcard
- **`min_value`** – The minimum allowed value

Raises **`ValueError`** – if the spec tries to exceed the limits

Example:

```
>>> list(parse_cron_spec('5', max_value=7,))
[5]

>>> list(parse_cron_spec('*', max_value=7,))
[0, 1, 2, 3, 4, 5, 6, 7]

>>> list(parse_cron_spec('1-4', max_value=7,))
[1, 2, 3, 4]

>>> list(parse_cron_spec('1-4/2', max_value=7,))
[1, 3]
```

3.10 Indices and tables

- genindex
- modindex

- search

Python Module Index

Index