mirakuru Documentation

Release 0.9.0

The A Room @ Clearcode

Mar 23, 2018

Contents

Mirakuru is a process orchestration tool designed for functional and integration tests.

Maybe you want to be able to start a database before you start your program or maybe you just need to set additional services up for your tests. This is where you should consider using **mirakuru** to add superpowers to your program or tests.

CHAPTER 1

Package status

CHAPTER 2

About

In a project that relies on multiple processes there might be a need to guard code with tests that verify interprocess communication. So one needs to set up all of required databases, auxiliary and application services to verify their cooperation. Synchronising (or orchestrating) test procedure with tested processes might be a hell.

If so, then **mirakuru** is what you need.

`Mirakuru` starts your process and waits for the clear indication that it's running. Library provides six executors to fit different cases:

- SimpleExecutor - starts a process and does not wait for anything. It is useful to stop or kill a process and its subprocesses. Base class for all the rest of executors.
- Executor - base class for executors verifying if a process has started.
- OutputExecutor - waits for a specified output to be printed by a process.
- TCPExecutor - waits for the ability to connect through TCP with a process.
- HTTPExecutor - waits for a successful HEAD request (and TCP before).
- PidExecutor - waits for a specified .pid file to exist.

```
from mirakuru import HTTPExecutor
from httplib import HTTPConnection, OK


def test_it_works():
    # The ``./http_server`` here launches some HTTP server on the 6543 port,
    # but naturally it is not immediate and takes a non-deterministic time:
    executor = HTTPExecutor("./http_server", url="http://127.0.0.1:6543/")

    # Start the server and wait for it to run (blocking):
    executor.start()
    # Here the server should be running!
    conn = HTTPConnection("127.0.0.1", 6543)
    conn.request("GET", "/")
```

```
    assert conn.getresponse().status is OK
    executor.stop()
```

A command by which executor spawns a process can be defined by either string or list.

```
# command as string
TCPExecutor('python -m smtpd -n -c DebuggingServer localhost:1025', host='localhost',␣
↪port=1025)
# command as list
TCPExecutor(
    ['python', '-m', 'smtpd', '-n', '-c', 'DebuggingServer', 'localhost:1025'],
    host='localhost', port=1025
)
```

CHAPTER 3

Authors

The project was firstly developed by Mateusz Lenik as the summon_process. Later forked, renamed into **mirakuru** and tended to by The A Room @ Clearcode and the other authors.

CHAPTER 4

License

`mirakuru` is licensed under LGPL license, version 3.

CHAPTER 5

Contributing and reporting bugs

Source code is available at: ClearcodeHQ/mirakuru. Issue tracker is located at GitHub Issues. Projects PyPI page.

When contributing, don't forget to add your name to the AUTHORS.rst file.

CHAPTER 6

Contents

6.1 Basic executors

Mirakuru *Executor* is something that you will use when you need to make some code dependant from other process being run, and in certain state, and you wouldn't want this process to be running all the time.

Tests would be best example here or a script that sets up processes and databases for dev environment with one simple run.

6.1.1 SimpleExecutor

mirakuru.base.SimpleExecutor is the simplest executor implementation. It simply starts the process passed to constructor, and reports it as running.

```
from mirakuru import SimpleExecutor

process = SimpleExecutor('my_special_process')
process.start()

# Here you can do your stuff, e.g. communicate with the started process

process.stop()
```

6.1.2 OutputExecutor

mirakuru.output.OutputExecutor is the executor that starts the process, but does not report it as started, unless it receives specified marker/banner in process output.

```
from mirakuru import OutputExecutor

process = OutputExecutor('my_special_process', banner='processed!')
process.start()
```

```
# Here you can do your stuff, e.g. communicate with the started process

process.stop()
```

What happens during start here, is that the executor constantly checks output produced by started process, and looks for the banner part occurring within the output. Once the output is identified, as in example *processed!* is found in output. It is considered as started, and executor releases your script from wait to work.

6.1.3 TCPExecutor

mirakuru.tcp.TCPExecutor is the executor that should be used to start processes that are using TCP connection. This executor tries to connect with the process on given host:port to see if it started accepting connections. Once it does, it reports the process as started and a code returns to normal execution.

```
from mirakuru import TCPExecutor

process = TCPExecutor('my_special_process', host='localhost', port=1234)
process.start()

# Here you can do your stuff, e.g. communicate with the started process

process.stop()
```

6.1.4 HTTPExecutor

mirakuru.http.HTTPExecutor is executor that will be used to start web applications for example. To start it, you apart from command, you need to pass a URL. This URL will be used to make a HEAD request. Once successful, the executor will be considered started, and a code will return to normal execution.

```
from mirakuru import HTTPExecutor

process = HTTPExecutor('my_special_process', url='http://localhost:6543/status')
process.start()

# Here you can do your stuff, e.g. communicate with the started process

process.stop()
```

This executor, however, apart from HEAD request, also inherits TCPExecutor, so it'll try to connect to process over TCP first, to determine, if it can try to make a HEAD request already.

By default HTTPExecutor waits until its subprocess responds with 2XX HTTP status code. If you consider other codes as valid you need to specify them in 'status' argument.

```
from mirakuru import HTTPExecutor

process = HTTPExecutor('my_special_process', url='http://localhost:6543/status',␣
↪status='(200|404)')
process.start()
```

The "status" argument can be a single code integer like 200, 404, 500 or a regular expression string - '^(2|4)00$', '2dd', 'd{3}', etc.

6.1.5 PidExecutor

mirakuru.pid.PidExecutor is an executor that starts the given process, and then waits for a given file to be found before it gives back control. An example use for this class is writing integration tests for processes that notify their running by creating a .pid file.

```
from mirakuru import PidExecutor

process = PidExecutor('my_special_process', filename='/var/msp/my_special_process.pid
↪')
process.start()

# Here you can do your stuff, e.g. communicate with the started process

process.stop()
```

6.1.6 As a Context manager

Starting

Mirakuru executors can also work as a context managers.

```
from mirakuru import HTTPExecutor

with HTTPExecutor('my_special_process', url='http://localhost:6543/status') as␣
↪process:

    # Here you can do your stuff, e.g. communicate with the started process
    assert process.running() is True

assert process.running() is False
```

Defined process starts upon entering context, and exit upon exiting it.

Stopping

Mirakuru also allows to stop process for given context. To do this, simply use built-in stopped context manager.

```
from mirakuru import HTTPExecutor

process = HTTPExecutor('my_special_process', url='http://localhost:6543/status').
↪start()

# Here you can do your stuff, e.g. communicate with the started process

with process.stopped():

    # Here you will not be able to communicate with the process as it is killed here
    assert process.running() is False

assert process.running() is True
```

Defined process stops upon entering context, and starts upon exiting it.

6.1.7 Methods chaining

Mirakuru encourages methods chaining so you can inline some operations, e.g.:

```
from mirakuru import SimpleExecutor

command_stdout = SimpleExecutor('my_special_process').start().stop().output
```

6.2 Api

6.2.1 Basic executors

Executor with the core functionality.

mirakuru.base.**ENV_UUID = 'mirakuru_uuid'**
Name of the environment variable used by mirakuru to mark its subprocesses.

class mirakuru.base.**Executor**(*command*, *shell=False*, *timeout=None*, *sleep=0.1*, *sig_stop=<Signals.SIGTERM: 15>*, *sig_kill=<Signals.SIGKILL: 9>*)
Bases: *mirakuru.base.SimpleExecutor*

Base class for executors with a pre- and after-start checks.

Initialize executor.

Parameters

- **list) command** (*(str,)* – command to be run by the subprocess
- **shell** (*bool*) – same as the *subprocess.Popen* shell definition
- **timeout** (*int*) – number of seconds to wait for the process to start or stop. If None or False, wait indefinitely.
- **sleep** (*float*) – how often to check for start/stop condition
- **sig_stop** (*int*) – signal used to stop process run by the executor. default is *signal.SIGTERM*
- **sig_kill** (*int*) – signal used to kill process run by the executor. default is *signal.SIGKILL* (*signal.SIGTERM* on Windows)

Note: **timeout** set for an executor is valid for all the level of waits on the way up. That means that if some more advanced executor establishes the timeout to 10 seconds and it will take 5 seconds for the first check, second check will only have 5 seconds left.

Your executor will raise an exception if something goes wrong during this time. The default value of timeout is `None`, so it is a good practice to set this.

---

after_start_check()
Check process after the start of executor.

Should be overridden in order to return boolean value if executor can be treated as started. :rtype: bool

check_subprocess()
Make sure the process didn't exit with an error and run the checks.

Return type bool

Returns the actual check status

Raises ***ProcessExitedWithError*** – when the main process exits with an error

pre_start_check()
Check process before the start of executor.

Should be overridden in order to return True when some other executor (or process) has already started with the same configuration. :rtype: bool

start()
Start executor with additional checks.

Checks if previous executor isn't running then start process (executor) and wait until it's started. :returns: itself :rtype: Executor

class mirakuru.base.**SimpleExecutor**(*command*, *shell=False*, *timeout=None*, *sleep=0.1*, *sig_stop=<Signals.SIGTERM: 15>*, *sig_kill=<Signals.SIGKILL: 9>*)

Bases: object

Simple subprocess executor with start/stop/kill functionality.

Initialize executor.

Parameters

- **list) command** (*(str,*) – command to be run by the subprocess
- **shell** (*bool*) – same as the *subprocess.Popen* shell definition
- **timeout** (*int*) – number of seconds to wait for the process to start or stop. If None or False, wait indefinitely.
- **sleep** (*float*) – how often to check for start/stop condition
- **sig_stop** (*int*) – signal used to stop process run by the executor. default is *signal.SIGTERM*
- **sig_kill** (*int*) – signal used to kill process run by the executor. default is *signal.SIGKILL* (*signal.SIGTERM* on Windows)

Note: **timeout** set for an executor is valid for all the level of waits on the way up. That means that if some more advanced executor establishes the timeout to 10 seconds and it will take 5 seconds for the first check, second check will only have 5 seconds left.

Your executor will raise an exception if something goes wrong during this time. The default value of timeout is None, so it is a good practice to set this.

_clear_process()
Close stdin/stdout of subprocess.

It is required because of ResourceWarning in Python 3.

_kill_all_kids(*sig*)
Kill all subprocesses (and its subprocesses) that executor started.

This function tries to kill all leftovers in process tree that current executor may have left. It uses environment variable to recognise if process have origin in this Executor so it does not give 100 % and some daemons fired by subprocess may still be running.

Parameters **sig** (*int*) – signal used to stop process run by executor.

Returns process ids (pids) of killed processes

:rtype list

_set_timeout(*timeout=None*)
Set timeout for possible wait.

Parameters **timeout** (*int*) – [optional] specific timeout to set. If not set, Executor._timeout will be used instead.

check_timeout()
Check if timeout has expired.

Returns True if there is no timeout set or the timeout has not expired. Kills the process and raises TimeoutExpired exception otherwise.

This method should be used in while loops waiting for some data.

Returns True if timeout expired, False if not

Return type bool

command = None
Command that the executor runs.

kill(*wait=True*, *sig=None*)
Kill the process if running.

Parameters

- **wait** (*bool*) – set to *True* to wait for the process to end, or False, to simply proceed after sending signal.
- **sig** (*int*) – signal used to kill process run by the executor. None by default.

Returns itself

Return type *SimpleExecutor*

output()
Return subprocess output.

process = None
A `subprocess.Popen` instance once process is started.

running()
Check if executor is running.

Returns True if process is running, False otherwise

Return type bool

start()
Start defined process.

After process gets started, timeout countdown begins as well.

Returns itself

Return type *SimpleExecutor*

Note: We want to open `stdin`, `stdout` and `stderr` as text streams in universal newlines mode, so we have to set `universal_newlines` to `True`.

---

stop(*sig=None*)
Stop process running.

Wait 10 seconds for the process to end, then just kill it.

Parameters **sig** (*int*) – signal used to stop process run by executor. None for default.

Returns itself

Return type *SimpleExecutor*

Note: When gathering coverage for the subprocess in tests, you have to allow subprocesses to end gracefully.

---

stopped()

Stop process for given context and starts it afterwards.

Allows for easier writing resistance integration tests whenever one of the service fails. :yields: itself :rtype: SimpleExecutor

wait_for(*wait_for*)

Wait for callback to return True.

Simply returns if wait_for condition has been met, raises TimeoutExpired otherwise and kills the process.

Parameters **wait_for** (*callback*) – callback to call

Raises mirakuru.exceptions.TimeoutExpired

Returns itself

Return type *SimpleExecutor*

mirakuru.base.**cleanup_subprocesses**()

On python exit: find possibly running subprocesses and kill them.

Executor that awaits for appearance of a predefined banner in output.

class mirakuru.output.**OutputExecutor**(*command*, *banner*, ***kwargs*)

Bases: *mirakuru.base.SimpleExecutor*

Executor that awaits for string output being present in output.

Initialize OutputExecutor executor.

Parameters

- **list) command** ((*str*,) – command to be run by the subprocess
- **banner** (*str*) – string that has to appear in process output - should compile to regular expression.
- **shell** (*bool*) – same as the *subprocess.Popen* shell definition
- **timeout** (*int*) – number of seconds to wait for the process to start or stop. If None or False, wait indefinitely.
- **sleep** (*float*) – how often to check for start/stop condition
- **sig_stop** (*int*) – signal used to stop process run by the executor. default is *signal.SIGTERM*
- **sig_kill** (*int*) – signal used to kill process run by the executor. default is *signal.SIGKILL* (*signal.SIGTERM* on Windows)

_wait_for_output()

Check if output matches banner.

> **Warning:** Waiting for I/O completion. It does not work on Windows. Sorry.

start()
Start process.

Returns itself

Return type *OutputExecutor*

> **Note:** Process will be considered started, when defined banner will appear in process output.

TCP executor definition.

class mirakuru.tcp.**TCPExecutor**(*command*, *host*, *port*, ***kwargs*)
Bases: *mirakuru.base.Executor*

TCP-listening process executor.

Used to start (and wait to actually be running) processes that can accept TCP connections.

Initialize TCPExecutor executor.

Parameters

- **list) command** (*(str,)* – command to be run by the subprocess
- **host** (*str*) – host under which process is accessible
- **port** (*int*) – port under which process is accessible
- **shell** (*bool*) – same as the *subprocess.Popen* shell definition
- **timeout** (*int*) – number of seconds to wait for the process to start or stop. If None or False, wait indefinitely.
- **sleep** (*float*) – how often to check for start/stop condition
- **sig_stop** (*int*) – signal used to stop process run by the executor. default is *signal.SIGTERM*
- **sig_kill** (*int*) – signal used to kill process run by the executor. default is *signal.SIGKILL* (*signal.SIGTERM* on Windows)

after_start_check()
Check if process accepts connections.

> **Note:** Process will be considered started, when it'll be able to accept TCP connections as defined in initializer.

host = None
Host name, process is listening on.

port = None
Port number, process is listening on.

pre_start_check()
Check if process accepts connections.

Note: Process will be considered started, when it'll be able to accept TCP connections as defined in initializer.

HTTP enabled process executor.

class mirakuru.http.**HTTPExecutor**(*command*, *url*, *status='^2\d\d$'*, ***kwargs*)
Bases: *mirakuru.tcp.TCPExecutor*

Http enabled process executor.

Initialize HTTPExecutor executor.

Parameters

- **list) command** (*(str,*) – command to be run by the subprocess
- **url** (*str*) – URL that executor checks to verify if process has already started.
- **shell** (*bool*) – same as the *subprocess.Popen* shell definition
- **status** (*str|int*) – HTTP status code(s) that an endpoint must return for the executor being considered as running. This argument is interpreted as a single status code - e.g. '200' or '404' but also it can be a regular expression - e.g. '4..' or '(200|404)'. Default: any 2XX HTTP status code.
- **timeout** (*int*) – number of seconds to wait for the process to start or stop. If None or False, wait indefinitely.
- **sleep** (*float*) – how often to check for start/stop condition
- **sig_stop** (*int*) – signal used to stop process run by the executor. default is *signal.SIGTERM*
- **sig_kill** (*int*) – signal used to kill process run by the executor. default is *signal.SIGKILL*

DEFAULT_PORT = 80
Default TCP port for the HTTP protocol.

after_start_check()
Check if defined URL returns expected status to a HEAD request.

url = None
An `urlparse.urlparse()` representation of an url.

It'll be used to check process status on.

Pid executor definition.

class mirakuru.pid.**PidExecutor**(*command*, *filename*, ***kwargs*)
Bases: *mirakuru.base.Executor*

File existence checking process executor.

Used to start processes that create pid files (or any other for that matter). Starts the given process and waits for the given file to be created.

Initialize the PidExecutor executor.

If the filename is empty, a ValueError is thrown.

Parameters

- **list) command** (*(str,*) – command to be run by the subprocess

- **filename** (*str*) – the file which is to exist
- **shell** (*bool*) – same as the *subprocess.Popen* shell definition
- **timeout** (*int*) – number of seconds to wait for the process to start or stop. If None or False, wait indefinitely.
- **sleep** (*float*) – how often to check for start/stop condition
- **sig_stop** (*int*) – signal used to stop process run by the executor. default is *signal.SIGTERM*
- **sig_kill** (*int*) – signal used to kill process run by the executor. default is *signal.SIGKILL* (*signal.SIGTERM* on Windows)

Raises ValueError

after_start_check()
Check if the process has created the specified file.

Note: The process will be considered started when it will have created the specified file as defined in the initializer.

---

filename = None
the name of the file which the process is to create.

pre_start_check()
Check if the specified file has been created.

Note: The process will be considered started when it will have created the specified file as defined in the initializer.

6.2.2 Exceptions

Mirakuru exceptions.

exception mirakuru.exceptions.**AlreadyRunning**(*executor*)
Bases: *mirakuru.exceptions.ExecutorError*

Is raised when the executor seems to be already running.

When some other process (not necessary executor) seems to be started with same configuration we can't bind to same port.

Exception initialization.

Parameters **executor** (mirakuru.base.Executor) – for which exception occurred

exception mirakuru.exceptions.**ExecutorError**(*executor*)
Bases: Exception

Base exception for executor failures.

Exception initialization.

Parameters **executor** (mirakuru.base.Executor) – for which exception occurred

exception mirakuru.exceptions.**ProcessExitedWithError**(*executor*, *exit_code*)
Bases: *mirakuru.exceptions.ExecutorError*

Raised when the process invoked by the executor returns a non-zero code.

We allow the process to exit with zero because we support daemonizing subprocesses. We assume that when double-forking, the parent process will exit with 0 in case of successful daemonization.

Exception initialization with an extra `exit_code` argument.

Parameters

- **executor** (mirakuru.base.Executor) – for which exception occurred
- **exit_code** (*int*) – code the subprocess exited with

exception mirakuru.exceptions.**TimeoutExpired**(*executor*, *timeout*)

Bases: *mirakuru.exceptions.ExecutorError*

Is raised when the timeout expires while starting an executor.

Exception initialization with an extra `timeout` argument.

Parameters

- **executor** (mirakuru.base.Executor) – for which exception occurred
- **timeout** (*int*) – timeout for which exception occurred

6.3 Contribute to mirakuru

Thank you for taking time to contribute to mirakuru!

The following is a set of guidelines for contributing to mirakuru. These are just guidelines, not rules, use your best judgment and feel free to propose changes to this document in a pull request.

6.3.1 Bug Reports

1. Use a clear and descriptive title for the issue - it'll be much easier to identify the problem.
2. Describe the steps to reproduce the problems in as many details as possible.
3. If possible, provide a code snippet to reproduce the issue.

6.3.2 Feature requests/proposals

1. Use a clear and descriptive title for the proposal
2. **Provide as detailed description as possible**
    - Use case is great to have
3. There'll be a bit of discussion for the feature. Don't worry, if it is to be accepted, we'd like to support it, so we need to understand it thoroughly.

6.3.3 Pull requests

1. Start with a bug report or feature request
2. Use a clear and descriptive title
3. Provide a description - which issue does it refers to, and what part of the issue is being solved

4. Be ready for code review :)

6.3.4 Commits

1. Make sure commits are atomic, and each atomic change is being followed by test.
2. If the commit solves part of the issue reported, include *refs #[Issue number]* in a commit message.
3. If the commit solves whole issue reported, please refer to Closing issues via commit messages for ways to close issues when commits will be merged.

6.3.5 Coding style

1. All python coding style are being enforced by Pylama and configured in pylama.ini file.
2. Additional, not always mandatory checks are being performed by QuantifiedCode

6.4 CHANGELOG

6.4.1 0.9.0

- [enhancement] Fallback to kill through SIGTERM on Windows, since SIGKILL is not available
- [enhancement] detect cases where during stop process already exited, and simply clean up afterwards

6.4.2 0.8.3

- [enhancement] when killing the process ignore OsError with errno *no such process* as the process have already died.
- [enhancement] small context manager code cleanup

6.4.3 0.8.2

- [bugfix] atexit cleanup_subprocesses() function now reimports needed functions

6.4.4 0.8.1

- [bugfix] Handle IOErrors from psutil (#112)
- [bugfix] Pass global vars to atexit cleanup_subprocesses function (#111)

6.4.5 0.8.0

- [feature] Kill all running mirakuru subprocesses on python exit.
- [enhancement] Prefer psutil library (>=4.0.0) over calling 'ps xe' command to find leaked subprocesses.

6.4.6 0.7.0

- [feature] HTTPExecutor enriched with the 'status' argument. It allows to define which HTTP status code(s) signify that a HTTP server is running.
- [feature] Changed executor methods to return itself to allow method chaining.
- [feature] Context Manager to return Executor instance, allows creating Executor instance on the fly.
- [style] Migrated % string formating to *format()*.
- [style] Explicitly numbered replacement fields in string.
- [docs] Added documentation for timeouts.

6.4.7 0.6.1

- [refactoring] Moved source to src directory.
- [fix, feature] Python 3.5 fixes.
- [fix] Docstring changes for updated pep257.

6.4.8 0.6.0

- [fix] Modify MANIFEST to prune tests folder.
- [feature] HTTPExecutor will now set the default 80 if not present in a URL.
- [feature] Detect subprocesses exiting erroneously while polling the checks and error early.
- [fix] Make test_forgotten_stop pass by preventing the shell from optimizing forking out.

6.4.9 0.5.0

- [style] Corrected code to conform with W503, D210 and E402 linters errors as reported by pylama *6.3.1*.
- [feature] Introduced a hack that kills all subprocesses of executor process. It requires 'ps xe -ww' command being available in OS otherwise logs error.
- [refactoring] Classes name convention change. Executor class got renamed into SimpleExecutor and StartCheckExecutor class got renamed into Executor.

6.4.10 0.4.0

- [feature] Ability to set up custom signal for stopping and killing processes managed by executors.
- [feature] Replaced explicit parameters with keywords for kwargs handled by basic Executor init method.
- [feature] Executor now accepts both list and string as a command.
- [fix] Even it's not recommended to import all but *from mirakuru import* * didn't worked. Now it's fixed.
- [tests] increased tests coverage. Even test cover 100% of code it doesn't mean they cover 100% of use cases!
- [code quality] Increased Pylint code evaluation.

6.4.11 0.3.0

- [feature] Introduced PidExecutor that waits for specified file to be created.
- [feature] Provided PyPy compatibility.
- [fix] Closing all resources explicitly.

6.4.12 0.2.0

- [fix] Kill all children processes of Executor started with shell=True.
- [feature] Executors are now context managers - to start executors for given context.
- [feature] Executor.stopped - context manager for stopping executors for given context.
- [feature] HTTPExecutor and TCPExecutor before .start() check whether port is already used by other processes and raise AlreadyRunning if detects it.
- [refactoring] Moved python version conditional imports into compat.py module.

6.4.13 0.1.4

- [fix] Fixed an issue where setting shell to True would execute only part of the command.

6.4.14 0.1.3

- [fix] Fixed an issue where OutputExecutor would hang, if started process stopped producing output.

6.4.15 0.1.2

- [fix] Removed leftover sleep from TCPExecutor._wait_for_connection.

6.4.16 0.1.1

- [fix] Fixed *MANIFEST.in*.
- Updated packaging options.

6.4.17 0.1.0

- Exposed process attribute on Executor.
- Exposed port and host on TCPExecutor.
- Exposed URL on HTTPExecutor.
- Simplified package structure.
- Simplified executors operating API.
- Updated documentation.
- Added docblocks for every function.
- Applied license headers.

- Stripped orchestrators.
- Forked off from *summon_process*.

CHAPTER 7

License

Python Module Index

m

Index

Symbols

A

C

D

E

F

H

K

M

O

P

R

S

T

U

W