mirakuru Documentation

Release 0.5.0

The A Room @ Clearcode

May 07, 2015

Contents

Maybe you want to be able to start database only when you start your program, or maybe you need just to set up additional processes for your tests, this is where you should consider using **mirakuru**, to add superpowers to your program, or tests.

CHAPTER 1

Package status

About

As developers, we have to work on project that rely on multiple processes to run. We guard ourselves with tests. But sometimes it's not enough what one process sends, and the other receives. Sometimes there's need to actually exchange data between processes. Or write selenium tests. Or maybe write a program that takes care of starting databases or other required services itself.

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

`Mirakuru` starts your required process, and wait for clear indication, that it's running. There are three basic executors with predefined conditions:

- Executor - simply starts
- OutputExecutor - awaits for specified output to be given by process
- TCPExecutor - waits for ability to connect through tcp with process
- HTTPExecutor - waits for successful HEAD request (and tcp before)
- PidExecutor - waits for a specified file to exist

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


def test_it_works():
    executor = HTTPExecutor("./server", url="http://localhost:6543/")

    # start and wait for it to run
    executor.start()
    # should be running!
    conn = HTTPConnection("localhost", 6543)
    conn.request('GET', '/')
    assert conn.getresponse().status is OK
    executor.stop()
```

The `server` command in this case is just a bash script that sleeps for some time and then launches the builtin SimpleHTTPServer on port 6543.

Command by which executor spawns a process, can be 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

Author

The project was first developed by Mateusz Lenik as summon_process. Later forked, renamed to **mirakuru** and tended to by The A Room @ Clearcode.

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 AUTHORS.rst file.

CHAPTER 6

Contents

6.1 Basic executors

Mirakuru's *Executor* is something that You'll use, when you'll 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 Executor

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

```
from mirakuru import Executor

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

# Do your stuff

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()

# Do your stuff

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, like in example *processed!* is found in output. It's 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 process on given host:port to see if it started accepting connections. Once it does, it reports the process as started and code returns to normal execution.

```
from mirakuru import TCPExecutor

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

# Do your stuff

process.stop()
```

6.1.4 HTTPExecutor

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

```
from mirakuru import HTTPExecutor

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

# Do your stuff

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.

6.1.5 PidExecutor

mirakuru.pid.PidExecutor is an executor that starts the given process, 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='/bla/ble/my_special_process.pid')
process.start()

# Do your stuff

process.stop()
```

6.1.6 As a Context manager

Starting

Mirakuru's executors can also work as a context managers.

```
from mirakuru import HTTPExecutor

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

    # Do your stuff
    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')
process.start()

# do some stuff

with process.stopped():

    # Do something hidden

    assert process.running() is False
assert process.running() is True
```

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

6.2 Api

6.2.1 Basic executors

Base executor with the most basic functionality.

class `mirakuru.base.`**`Executor`**(*command*, *shell=False*, *timeout=None*, *sleep=0.1*, *sig_stop=15*, *sig_kill=9*)

Bases: *`mirakuru.base.SimpleExecutor`*

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

Initialize executor.

Parameters

- **`list) command`** ((*str,*) – command to run to start service
- **`shell`** (*bool*) – see *subprocess.Popen*
- **`timeout`** (*int*) – time to wait for process to start or stop. if None, wait indefinitely.
- **`sleep`** (*float*) – how often to check for start/stop condition
- **`sig_stop`** (*int*) – signal used to stop process run by executor. default is SIGTERM
- **`sig_kill`** (*int*) – signal used to kill process run by executor. default is SIGKILL

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

`after_start_check``()`
Method fired after the start of executor.

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

`pre_start_check``()`
Method fired before the start of executor.

Should be overridden in order to return boolean value if some process is already started. :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.

`mirakuru.base.`**`PS_XE_PID_MATCH`** **= <_sre.SRE_Pattern object>**
_sre.SRE_Pattern matching PIDs in result from *$ ps xe -ww* command.

class `mirakuru.base.`**`SimpleExecutor`**`(`*command*, *shell=False*, *timeout=None*, *sleep=0.1*, *sig_stop=15*, *sig_kill=9*`)`
Bases: `object`

Simple subprocess executor with start/stop/kill functionality.

Initialize executor.

Parameters

- **`list) command`** (*(str,*) – command to run to start service
- **`shell`** (*bool*) – see *subprocess.Popen*
- **`timeout`** (*int*) – time to wait for process to start or stop. if None, wait indefinitely.
- **`sleep`** (*float*) – how often to check for start/stop condition
- **`sig_stop`** (*int*) – signal used to stop process run by executor. default is SIGTERM
- **`sig_kill`** (*int*) – signal used to kill process run by executor. default is SIGKILL

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

`_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 Exeuctor so it does not give 100 % and some deamons 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 timout 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 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 executor. None for default.

`output`()
Return process 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.

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.

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

stopped(**args*, ***kwds*)

Stopping process for given context and starts it afterwards.

Allows for easier writing resistance integration tests whenever one of the service fails.

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

mirakuru.base.**processes_with_env**(*env_name*, *env_value*)

Find PIDs of processes having env variable matching given one.

Function uses *$ ps e -ww* command so it works only on systems having such command available (linux, macos). If not available function will just log error.

Parameters

- **env_name** (*str*) – name of environment variable to be found
- **env_value** (*str*) – environment variable value

Returns process ids (PIDs) of processes that have certain environment variable with certain value

Return type set

This executor 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 run to start service
- **banner** (*str*) – string that has to appear in process output - should compile to regular expression.
- **shell** (*bool*) – see *subprocess.Popen*
- **timeout** (*int*) – time to wait for process to start or stop. if None, wait indefinitely.
- **sleep** (*float*) – how often to check for start/stop condition
- **sig_stop** (*int*) – signal used to stop process run by executor. default is SIGTERM
- **sig_kill** (*int*) – signal used to kill process run by executor. default is SIGKILL

_wait_for_output()

Check if output matches banner.

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

start()

Start process.

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 run to start service
- **`host`** (*str*) – host under which process is accessible
- **`port`** (*int*) – port under which process is accessible
- **`shell`** (*bool*) – see *subprocess.Popen*
- **`timeout`** (*int*) – time to wait for process to start or stop. if None, wait indefinitely.
- **`sleep`** (*float*) – how often to check for start/stop condition
- **`sig_stop`** (*int*) – signal used to stop process run by executor. default is SIGTERM
- **`sig_kill`** (*int*) – signal used to kill process run by executor. default is SIGKILL

`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*, ***kwargs*)

Bases: *`mirakuru.tcp.TCPExecutor`*

Http enabled process executor.

Initialize HTTPExecutor executor.

Parameters

- **`list) command`** ((*str,*) – command to run to start service
- **`url`** (*str*) – url where executor can check if process has already started.
- **`shell`** (*bool*) – see *subprocess.Popen*
- **`timeout`** (*int*) – time to wait for process to start or stop. if None, wait indefinitely.

- **`sleep`** (*float*) – how often to check for start/stop condition
- **`sig_stop`** (*int*) – signal used to stop process run by executor. default is SIGTERM
- **`sig_kill`** (*int*) – signal used to kill process run by executor. default is SIGKILL

`after_start_check``()`
Check if defined url returns successful head.

`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 run to start service
- **`filename`** (*str*) – the file which is to exist
- **`shell`** (*bool*) – see *subprocess.Popen*
- **`timeout`** (*int*) – time to wait for the process to start or stop. if None, wait indefinitely.
- **`sleep`** (*float*) – how often to check for start/stop conditions
- **`sig_stop`** (*int*) – signal used to stop process run by executor. default is SIGTERM
- **`sig_kill`** (*int*) – signal used to kill process run by executor. default is SIGKILL

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's exceptions.

exception `mirakuru.exceptions.`**`AlreadyRunning`**(*executor*)

Bases: `exceptions.Exception`

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 occured.

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

Bases: `exceptions.Exception`

Is raised when the timeout expires while starting an executor.

Exception initialization.

Parameters

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

6.3 CHANGELOG

6.3.1 0.5.0

- Corrected code to conform with W503, D210 and E402 linters errors as reported by pylama 6.3.1
- [feature] introduces 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.

6.3.2 0.4.0

- [feature] ability to set up custom signal for stopping and killing proceses 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.

6.3.3 0.3.0

- [feature] PidExecutor that waits for specified file to be created.
- pypy compatibility
- [fix] closing all resources explicitly

6.3.4 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
- moved python version conditional imports into compat.py module

6.3.5 0.1.4

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

6.3.6 0.1.3

- fix issue where OutputExecutor would hang, if started process stopped producing output

6.3.7 0.1.2

- [fix] removed leftover sleep from TCPExecutor._wait_for_connection

6.3.8 0.1.1

- fixed MANIFEST.in
- updated packaging options

6.3.9 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