Runium

Release 0.1.7

Jul 26, 2019

Overview:

1 Features **3**

2 Installation **5**

3 Quickstart **7**

3.1 Runium 8
3.2 Features 8
3.3 Installation 9
3.4 Quickstart 9
3.5 Initialize Runium 10
3.6 Creating tasks 10
3.7 Callbacks 11
3.8 Running Tasks 14
3.9 Contributing to Runium 15

Runium is a Python library that makes it easy to write non-blocking, asynchronous tasks.

You can add new tasks as you please, choose when and how to execute them as **Threads** or **Processes** and attach callbacks to be executed as soon as the task is finished running. Run those tasks once or periodically or schedule to run them at a specific time.

The purpose of Runium is to do these simple, easy and clean with minimum to no changes to your code. Just one line of code is all it takes.

Documentation https://runium.readthedocs.io/en/latest/main.html

Github: https://github.com/AGTGreg/runium

Pypi: https://pypi.org/project/runium/

CHAPTER 1

Features

- **Concurrency**: Run a task once or many times in its own Thread or Process.
- **Repetition**: Run tasks periodically on even time intervals. Optionally for a certain amount of times.
- **Scheduling**: Run tasks at a certain date and time.
- **Callbacks**: Runium tasks can accept callback functions which are executed when the task is finished running.
- **Simplicity and Readability**: Do all the above in a single line of code that is easy to read.

CHAPTER 2

Installation

Runium is distributed on PyPI. The best way to install it is with pip:

```
$ pip install runium
```

CHAPTER 3

Quickstart

A basic example:

```
from runium.core import Runium


def task():
    print("I am working...")


# Initialize Runium
rn = Runium()

# Create a task
async_task = rn.new_task(task)

# Run it. This will return a future object.
future = async_task.run()

# Then you can wait for the result.
future.result()

# Of course you can do all these in one line :)
rn.new_task(task).run().result()
```

Callbacks:

```
async_task.on_finished(callback).run()

async_task.on_success(s_callback).on_error(e_callback).run()

async_task.on_iter(callback).run()
```

Choose how, when and how many times a task should run:

```
# Run it multiple times...
future = async_task.run(times=3)

# ...every 1 hour
future = async_task.run(every='1 hour', times=3)

# Or tell Runium to start the task in a specific time
future = async_task.run(start_in='5 hours')

# All the methods above will return a future object. You can use result()
# to wait for it.
future.result()

# Here it is in one line :)
rn.new_task(task).run(every='1 second', times=3).result()
```

3.1 Runium

Runium is a Python library that makes it easy to write non-blocking, asynchronous tasks.

You can add new tasks as you please, choose when and how to execute them as **Threads** or **Processes** and attach callbacks to be executed as soon as the task is finished running. Run those tasks once or periodically or schedule to run them at a specific time.

The purpose of Runium is to do these simple, easy and clean with minimum to no changes to your code. Just one line of code is all it takes.

Documentation https://runium.readthedocs.io/en/latest/main.html

Github: https://github.com/AGTGreg/runium

Pypi: https://pypi.org/project/runium/

3.2 Features

- **Concurrency**: Run a task once or many times in its own Thread or Process.
- **Repetition**: Run tasks periodically on even time intervals. Optionally for a certain amount of times.
- **Scheduling**: Run tasks at a certain date and time.
- **Callbacks**: Runium tasks can accept callback functions which are executed when the task is finished running.
- **Simplicity and Readability**: Do all the above in a single line of code that is easy to read.

3.3 Installation

Runium is distributed on PyPI. The best way to install it is with pip:

```
$ pip install runium
```

3.4 Quickstart

A basic example:

```
from runium.core import Runium


def task():
    print("I am working...")


# Initialize Runium
rn = Runium()

# Create a task
async_task = rn.new_task(task)

# Run it. This will return a future object.
future = async_task.run()

# Then you can wait for the result.
future.result()

# Of course you can do all these in one line :)
rn.new_task(task).run().result()
```

Callbacks:

```
async_task.on_finished(callback).run()

async_task.on_success(s_callback).on_error(e_callback).run()

async_task.on_iter(callback).run()
```

Choose how, when and how many times a task should run:

```
# Run it multiple times...
future = async_task.run(times=3)

# ...every 1 hour
future = async_task.run(every='1 hour', times=3)

# Or tell Runium to start the task in a specific time
future = async_task.run(start_in='5 hours')

# All the methods above will return a future object. You can use result()
# to wait for it.
future.result()
```

(continues on next page)

(continued from previous page)

```
# Here it is in one line :)
rn.new_task(task).run(every='1 second', times=3).result()
```

3.5 Initialize Runium

It all starts with the Runium object which lives in `runium.core`. You will use Runium to create new tasks and choose how to run them (as Threads or Processes).

Runium also keeps a list of all the pending tasks.

3.5.1 Runium()

`runium.core.Runium(mode='multithreading', max_workers=None, debug=True)`

In order to start using Runium, you must first import it from the core module and Intitialize it.

Runium will start a new Thread or Process pool and will create an empty tasks list.

Parameters

- **mode** – *(Optional)* A string indicating wether the tasks should be run as Threads or Processes. It can be either 'multithreading' (Default) or 'multiprocessing'.
- **max_workers** – *(Optional)* An integer that indicates the max number of Threads or Processes to use. In Multithreading mode if max_workers is None or not given, it will default to the number of processors on the machine, multiplied by 5. In Multiprocessing mode it will default to the number of processors on the machine.
- **debug** – *(Optional)* If True all the exceptions raised by the tasks will be printed in the console.

Example:

```
from runium.core import Runium

# This will run tasks as seperate Threads.
rn = Runium()
# This will run tasks as seperate Processes.
rn = Runium(mode='multiprocessing')
```

3.5.2 pending_tasks

`Runium.pending_tasks()`

Returns a dictionary with all the pending tasks that looks like this:

3.6 Creating tasks

After initializing Runium, the first thing you want to do is create new tasks. Creating a new task will not execute it but it will return a `Task` object instead were you can attach callbacks and call its `run()` method to start running the task.

3.6.1 new_task

`Runium.new_task(fn, kwargs={})`

Creates a new Task, and adds it to the tasks list. Returns a handy `runium.core.Task` object.

Parameters

- **fn** – The callable to be executed.
- **kwargs** – *(Optional)* A dictionary that contains the arguments of the callable (if any).

Example:

```
def simple_task():
    print("I'm working...")

runium.new_task(simple_task)
```

Here's what it looks like if your task has arguments:

```
def send_email(to, msg):
    print("Sending", msg, "to", to)
    return True


runium.new_task(
    send_email, kwargs={
        'to': 'mail.example.com',
        'msg': 'This is a test email.'
    })
```

Note: You can add the optional argument `runium` in your function to get access to some of the statistics of that task inside that function. For example:

```
def task(runium):
    print(runium['iterations_remaining'])

runium.new_task(task).run(times=3)
```

```
>> 3
>> 2
>> 1
```

3.7 Callbacks

A callback is a function that is attached to a `runium.core.Task` object and gets executed as soon as the task finishes.

Callbacks are executed in the same Thread/Process as the Task that calls them. As a result they are non-blocking.

3.7.1 on_finished

`Task.on_finished(fn, updates_result=False)`

Runs the callback after the task has been executed successfully or after an exception was raised.

Accepts a callable with the task's success and error results as its only arguments.

If the task was successfull (no exceptions were raised) then the success argument will contain the task's return and the error argument will be `None`.

If the task is unsuccessfull (an exception was raised) then the error argument will contain the exception object and success will be `None`.

Parameters

- **fn** – The callable to be executed with success and error as its only arguments.
- **updates_result** – *(Optional)* If this is True then the task's result will be replaced with whatever is returned by the callable.

Example:

```
def send_email():
    print("Sending email...")
    return "Email sent."


# The callback must have the success and error arguments.
def callback(success, error):
    if success:
        return True
    elif error:
        return "An error occurred."


# Attach the callback like this
async_task = runium.new_task(send_email).on_finished(callback).run()


# You may also choose to get the result of the callback instead of the task
# by setting the parameter updates_result to True.
async_task = runium.new_task(send_email).on_finished(
    callback, updates_result=True).run()

# This will return True
async_task.result()
```

3.7.2 on_success

`Task.on_success(fn, updates_result=False)`

Runs the callback after the task has been executed successfully and no exceptions were raised.

Accepts a callable with the task's result as its only argument.

Parameters

- **fn** – The callable to be executed with success as its only argument.
- **updates_result** – *(Optional)* If this is True then the task's result will be replaced with whatever is returned by the callable.

Example:

```
def send_email():
    print("Sending email...")
    return "Email sent."


# The callback must have the success argument.
def callback(success):
    return ("Success!")

# Attach the callback like this
async_task = runium.new_task(send_email).on_success(callback).run()

# This callback is often used together with on_error callback:
async_task = runium.new_task(
    send_email
).on_success(
    callback
).on_error(
    error_callback
).run()
```

3.7.3 on_error

`Task.on_error(fn, updates_result=False)`

Runs the callback after an exception was raised by the task.

Accepts a callable with the task's exception object as its only argument.

Parameters

- **fn** – The callable to be executed with error as its only argument.
- **updates_result** – *(Optional)* If this is True then the task's result will be replaced with whatever is returned by the callable.

Example:

```
def send_email():
    raise Exception("Email was not sent.)


# The callback must have the error argument.
def callback(error):
    resend_email()


# Attach the callback like this
async_task = runium.new_task(send_email).on_error(callback).run()

# This callback is often used together with on_success callback:
async_task = runium.new_task(
    send_email
).on_success(
    callback
).on_error(
    error_callback
).run()
```

3.7.4 on_iter

`Task.on_iter(fn, updates_result=False)`

Runs the callback every time the task is beeing executed successfully or after an exception was raised.

Accepts a callable with the task's success and error results as its only arguments.

If the task was successfull (no exceptions were raised) then the success argument will contain the task's return and the error argument will be `None`.

If the task is unsuccessfull (an exception was raised) then the error argument will contain the exception object and success will be `None`.

The difference between this type of callback and all the others is that the other callbacks will run only once after the task has been executed no matter how many times we've set it to run. But an on_iter callback will run on every iteration if the task is to be executed many times.

Parameters

- **fn** – The callable to be executed with success and error as its only arguments: fn(success, error)
- **updates_result** – *(Optional)* If this is True then the task's result will be replaced with whatever is returned by the callable.

Example:

```
# The callback must have the success and error arguments.
def callback(success, error):
    if success:
        print(success)
        return True
    elif error:
        print(error)
        return "An error occurred."

# The callback will be executed 3 times.
async_task = runium.new_task(send_email).on_iter(callback).run(times=3)
```

3.7.5 add_done_callback

This is not a Runium method but since `Task.run()` returns a Future object, you can also add callbacks using this method. But you have to call `run()` first before using this method. Read the documentation about it here: add_done_callback()

3.8 Running Tasks

After you create a Task you have to call its `run()` method in order to start running it. You can specify how often, how many times and when the task should start running.

Here's how it works:

3.8.1 run()

`Task.run(every=None, times=None, start_in=0)`

Starts running the task. Returns a Future object.

Parameters

- **every** – *(Optional)* Run the task every n seconds. This value can be an integer or a float that indicates the number of seconds or a string that starts with an integer and finishes with the time scale.
- **times** – *(Optional)* How many times the task should run. It accepts an integer.
- **start_in** – *(Optional)* The amount of seconds to wait before start runing the task. It accepts the same values as :every.

Note: In order to improve readability the `every` and `start_in` parameters can accept strings in this format:

`1 second 2 seconds 1 minute 2 minutes 1 hour 2 hours`

Example

```
# Run a task once.
async_task = runium.new_task(task).run()

# Run a task multiple times.
async_task = runium.new_task(task).run(times=3)

# Run the task at once then repeat indefinitely every 1 hour.
async_task = runium.new_task(task).run(every='1 hour')

# Start the task in 5 hours.
async_task = runium.new_task(task).run(start_in='5 hours')
```

Calling the `run()` method will give you a Future object so you may use its methods:

```
# You can call result() to wait for it.
async_task.result()
```

> **Warning:** Python's `future.excetion()` will allways return `None` because Runium catches all exceptions and passes them to `future.result()`.

3.9 Contributing to Runium

It is great you want to contribute! Let's make Runium great together! If you wish to add new features or fix a bug, you will have to follow some procedures and rules in order to get your changes accepted. This is to keep the code base nice and clean.

3.9.1 Contribution Process

- Fork the project on Github
- Clone the fork to your local machine
- Make the changes to the project
- Run the test suite with tox (if you changed any code)
- Commit if you haven't already

- Push the changes to your Github fork
- Make a pull request on Github

3.9.2 Getting started

First create a **Virtual Environment** with **Python 3.7** which is the default version used in Runium.

```
$ python3.7 venv -m runium-env
```

Then clone Runium inside the Virtual Environment and start coding!

3.9.3 Code style

We use PEP 8 rules and flake8. Basically if you use flake8 you're good. This applies to all text files (source code, tests, documentation). Just follow the surrounding code style as closely as possible.

3.9.4 Testing

Running the test suite is done using the **tox**. This will test the code base against all supported Python versions and performs some code quality checks using flake8 as well.

Any nontrivial code changes must be accompanied with the appropriate tests. The tests should not only maintain the coverage, but should test any new functionality or bug fixes reasonably well. If you're fixing a bug, first make sure you have a test which fails against the unpatched codebase and succeeds against the fixed version. Naturally, the test suite has to pass on every Python version. If setting up all the required Python interpreters seems like too much trouble, make sure that it at least passes on the lowest supported version of both Python. The full test suite is always run against each pull request, but it's a good idea to run the tests locally first.

3.9.5 Building the documentation

You will find all the documentation files inside `docs/source/`. We use **ReStructuredText** (.rst) for the docs files and **Sphinx** to compile them so if you want to contibute to the docs you'll have to install this as well: `pip install Sphinx`.

If you want to add a new file add it inside `docs/source/` then include it in the toctree inside `index.rst` and build the decumentation locally to make sure it works:

```
$ make clean
$ makde html
```

The documentation you just build will be inside `docs/build/`.