scriptor

Mikael Koli

Nov 12, 2022

CONTENTS:

- Documentation
- Source code (Github)
- Releases (PyPI)

Scriptor is a high-level abstraction for `subprocess` and `async.subprocess`. Scriptor makes it easy to execute command-line programs from Python.

Features:

- Run programs sync or async using the same syntax
- High-level program abstraction
- Easy program parametrization

A simple example:

```
>>> from scriptor import Program
>>> python = Program('python3')

>>> # Run a program (and wait for finish)
>>> python('myscript.py')
```

You can also conveniently parametrize programs:

```
>>> # Run: python3 myscript.py --report_date 2022-01-01
>>> python('myscript.py', report_date="2022-01-01")
```

You can create convenient interfaces to other programs with Scriptor. See:

- *Git interface*

CHAPTER

ONE

INTERESTED?

Install the package:

```
pip install scriptor
```

1.1 Tutorials

Welcome to Scriptor's tutorials.

1.1.1 Program

This section goes through how Scriptor's program class works.

Here is a minimal example of a program:

```
>>> from scriptor import Program
>>> program = Program("python3")
>>> program(c="print('Hello world')")
"Hello world"
```

Creating Program

There are two ways to create a program. The simplest one is to just create one from the `Program` class:

```
from scriptor import Program
git = Program('git')
```

Alternatively you can subclass `BaseProgram` if you want to make an interface for your program, ie.:

```
from scriptor import BaseProgram

class Git(BaseProgram):

    program = 'git'

    def fetch(self):
        "Run command: git fetch"
        return self('fetch')
```

(continues on next page)

(continued from previous page)

```
    def log(self, n):
        "Run command: git log -n ..."
        return self('log', n=n)
```

Then to create a program instance:

```
git = Git()
```

Changing Settings

Often it is desired to have specific settings for running a program. For example, you may want to run a program with a different current working directory (cwd) than where the current program is.

Here is an example:

```
>>> repo_1 = git.use(cwd="path/to/repo_1")
>>> repo_2 = git.use(cwd="path/to/repo_2")

>>> # Run git status with repo_1 as CWD
>>> repo_1("status")
...
```

Note: Method `program.use(...)` copies the instance thus you can easily create copies to run programs in different directories or settings.

Settings

cwd
: Current working directory used for running the program. By default same as current.

timeout
: Number of seconds to wait before terminating the program due to timeout. By default None.

arg_form: `'long'`, `'short'` or `None`
: Form of the argument, either `long` or `short`. If `None`, the argument form is interpret from the length of the key. By default None.

default_kwargs: Dict
: Default keyword arguments to pass all calls.

output_type: `'str'` or `'bytes'`
: Type of the program output (stdout), either `'str'` or `'bytes'`. By default `'str'`.

output_parser: callable
: Output parser. By default None.

Note: By default, Scriptor does not use shell to avoid command injection.

Running Program

We use the current interpreter as our example program:

```
import sys
from scriptor import Program
python = Program(sys.executable)
```

To run the program, just call it:

```
python('myscript.py')
```

Note that we passed one positional argument to the program (Python interpreter). Read more about *program parameters*.

You can also run the program async:

```
await python.call_async('myscript.py')
```

Outputs

Typically programs put their outputs to the standard output (stdout). Because of this, Scriptor returns the stdout as the output of the call.

For example, we have this script called `myscript.py`:

```
print("Hello")
print("world")
```

Then if we run this program:

```
>>> python('myscript.py')
'Hello\nworld'
```

Bytes as Output

Note that the output is string by default. You can also return raw bytes by setting the `output_type`:

```
>>> python = Program('python', output_type='bytes')
>>> python('myscript.py')
b'Hello\nworld\n'
```

Custom Output

Moreover, you can also add your own output parser if for example your program returns an object you wish to parse.

For example, let's say you have a program that returns a JSON in the stdout. Let's call it `myscript.py`:

```
print('{"name":"Miksu", "age":25, "life":null}')
```

Then we set a custom output parser:

```
>>> import json
>>> python = Program('python', output_parser=json.loads)
>>> python('myscript.py')
{"name": "Miksu", "age": 25, "life": None}
```

Errors

Standard error (stderr) is the typical output for the error messages if the program fails. Scriptor conveniently puts the stderr to the exception.

We have a Python script called `failing.py` which looks like:

```
raise RuntimeError("Oops")
```

If we run this program using Scriptor:

```
output = python('failing_script.py')
```

we get an error that looks like this:

```
Traceback (most recent call last):
File "...", line ..., in <module>
    python("failing_script.py")
...
File "...\scriptor\process.py", line ..., in _raise_for_error
    raise ProcessError(
scriptor.process.ProcessError: Traceback (most recent call last):
File "failing_script.py", line 1, in <module>
    raise RuntimeError("Oops!")
RuntimeError: Oops!
```

Note: The exception class is `scriptor.ProcessError` and not `RuntimeError`.

Starting a Program

You can also start a program and handle the finish later. Scriptor provides further abstraction for `subprocess.Popen` and `asyncio.subprocess.Process` to make working with the processes more intuitive.

To start a process synchronously:

```
process = python.start('myscript.py')
```

To start a process with async:

```
process = await python.start_async('myscript.py')
```

Program Parameters

Typically programs take in parameters or data in the following forms:

- Positional arguments
- Keyword arguments (either short or long form)
- Standard input (stdin)

Positional Arguments

```
python("myscript.py", "positional_argument")
```

Keyword Arguments

Calling a program supports also keyword arguments. The most common way to pass keyword arguments to programs is to supply them either in short form (ie. `-o myfile.txt`) or long form (ie. `--output myfile.txt`).

Scriptor supports both forms and it guesses the form by the length of the argument name. If it is shorter than what is specified in the argument `long_form_threshold`, the argument will be passed in short form. If it is longer, it is passed as long form. This is `3` by default.

The below will run command: `python myscript.py --report_date 2022-11-11`

```
python("myscript.py", report_date="2022-11-11")
```

The below will run command: `python myscript.py -rd 2022-11-11`

```
python("myscript.py", rd="2022-11-11")
```

Standard Input (stdin)

```
from scriptor.program import Input

python("myscript.py", Input('some data'))
```

> **Warning:** You should pass only one `Input` per call.

Note: You can have positional arguments as well. You can pass those before or after the standard input.

These are identical:

```
python("myscript.py", Input('some data'), "myarg")
python("myscript.py", "myarg", Input('some data'))
```

1.2 Cookbook

1.2.1 Python

There is a built-in Python program class for convenience:

```
>>> from scriptor.builtin import Python
>>> python = Python('.../env/bin/python')
```

You can also use your current interpreter or the base interpreter:

```
>>> from scriptor.builtin import current_python, base_python
```

In addition to other methods `Program` has, Python instances also have some additional features:

```
>>> # Inspect the interpreter
>>> python.version
"Python 3.8.10"
>>> python.full_version
"Python 3.8.10 ..."

>>> # Run code
>>> python.run_script("path/to/myscript.py")
>>> python.run_module("path.to.myscript")
>>> python.run_code("print('Hello world')")
"Hello world"
```

1.2.2 Git

This is an example of how to use Git with Scriptor.

Functional

First we go through how to use Git functionally with Scriptor.

First we create a Git program:

```
from scriptor import Program
git = Program('git')
```

As git works on the current working directory (cwd), we need to change that to our repository:

```
myrepo = git.use(cwd="path/to/myrepo")
```

Note: Method `.use` copies the program. You can create programs for multiple repositories by:

```
repo_scriptor = git.use(cwd="/repos/scriptor")
repo_redmail = git.use(cwd="/repos/redmail")
```

Then we can use this:

```
>>> myrepo("status")
"""On branch main
nothing to commit, working tree clean"""

>>> myrepo("fetch")
>>> myrepo("log", n=2)
"""commit 7939dde2fef44369f72911de17188dd51bbfd2e5
Author: Mikael Koli <mikael.koli@example.com>
Date:   Sat Nov 12 12:33:09 2022 +0200

    Made the thing work.

commit 492f63918d750c794641f90fb4b5440c4195e17b
Author: Mikael Koli <mikael.koli@example.com>
Date:   Sat Nov 12 11:53:18 2022 +0200

    Broke the thing."""
```

Object Oriented

You can also create more abstraction by subclassing `BaseProgram`:

```
from scriptor import BaseProgram

class Git(BaseProgram):

    program = "git"

    def __init__(self, repo=None, *args, **kwargs):
        super().__init__(*args, **kwargs)
        if repo:
            self.cwd = repo

    def fetch(self):
        "Run: git fetch"
        return self("fetch")

    @property
    def status(self):
        return self("status")

    def log(self, n:int):
        return self("log", n=n)
    ...
```

Then to use this:

```
>>> git = Git("path/to/myrepo")
>>> git.status
"""On branch main
nothing to commit, working tree clean"""
```

(continues on next page)

(continued from previous page)

```
>>> git.fetch()
>>> git.log(n=2)
"""commit 7939dde2fef44369f72911de17188dd51bbfd2e5
Author: Mikael Koli <mikael.koli@example.com>
Date:   Sat Nov 12 12:33:09 2022 +0200

    Made the thing work.

commit 492f63918d750c794641f90fb4b5440c4195e17b
Author: Mikael Koli <mikael.koli@example.com>
Date:   Sat Nov 12 11:53:18 2022 +0200

    Broke the thing."""
```

1.3 Version history

- `0.1.0`
  - Reserved PyPI name

1.4 Indices and tables

- genindex

INDEX