EveScript

Release 0.5.0

Charlee Li

Mar 30, 2021

CONTENTS

CHAPTER

ONE

OVERVIEW

EveScript is a simple script language for event-based automation tasks.

```
from evescript.compiler import EveScriptCompiler
from evescript.executor import EveScriptExecutor

script = '''
if ($lightSensor > 20) {
  say("It's daytime now!")
}
'''

def lightSensor():
    return read_light_sensor_port()

compiler = EveScriptCompiler()
compiled_script = compiler.compile(script)

executor = EveScriptExecutor({
    'actions': { 'say': lambda x: print(x) },
    'variables': { '$lightSensor': lightSensor },
})

executor.run_script(compiled_script)
# Out: It's daytime now!
```

EveScript allows you to write simple event-based scripts that evaluate various conditions and execute actions. The conditions and actions are highly customizable to maximize the flexibility. EveScript can be used in embedded systems (such as Raspberry Pi) to implement a flexible event-based system.

1.1 Installation

```
pip install evescript
```

CHAPTER

TWO

INSTALLATION

At the command line:

```
pip install evescript
```

CHAPTER

THREE

USAGE

EveScript consists of a compiler `EveScriptCompiler` and an executor `EveScriptExecutor`.

3.1 Write a Script

An EveScript is written in the following form:

```
if (expression) {
  action1(...)
  action2(...)
}
```

where, the expression is a bool expression, and the **actions** are a list of "function" calls. The condition expression consists of **variables**, **operators** and constants (strings, numbers, and booleans).

Actions, **variables**, and **operators** are three types of entities that must be provided before the script can execute.

Here is a quick example (`quickstart.es`) for you to start script with EveScript.

```
if ($lightSensor < 20) {
  say("It's getting dark now!")
}
```

In this script, we used three entities that need to be provided when executing.

- `$lightSensor`: This is a variable. A variable is a custom function that is provided as a data source to the script.
- `say`: This is an action. An action is a custom function that will be called when the expression is true.

We will provide these entities in the *Run a script* section.

For more details, see *EveScript Language Reference*.

3.2 Compile a Script

An EveScript file (`*.es`) needs to be compiled before executed. This is done by calling the `compile()` method in the `EveScriptCompiler` class.

```
from evescript.compiler import EveScriptCompiler

with open('quickstart.es') as f:
  script = f.read()

compiler = EveScriptCompiler()
compiled_script = compiler.compile(f)
```

3.3 Run a Script

The compiled script can be executed with `EveScriptExecutor`. When instancing `EveScriptExecutor`, the entities (actions, operators, and variables) used in the scripts must be provided. Each entity is a function or lambda.

```
from evescript.executor import EveScriptExecutor

ACTIONS = {
  'say': lambda s: print(s),
}

VARIABLES = {
  '$lightSensor': lambda x: 10,
}

# Provide the actions and the variables used in the script
executor = EveScriptExecutor({
  'actions': ACTIONS,
  'variables': VARIABLES,
})

# run the first trigger
executor.run_script(compiled_script)
```

Since we mocked the varialbe `$lightSensor` to make it always returns 10, the action will be executed and will print `It's getting dark now!`.

3.4 Decompile a Script

Sometimes it is necessary to retrieve the script source for given compiled script. This can be done with `EveScriptDecompiler`. It provides a `decompile` method that takes a compiled script and returns the original script text.

NOTE: The decompiled script may not be identical to the original script - the whitespaces, new lines may differ, and the decompiled script will not contain any comments that may have appeared in the original script.

The following snippet shows how to decompile:

```
import os
import sys

from evescript.compiler import EveScriptCompiler
from evescript.decompiler import EveScriptDecompiler

script = '''
if ($lightSensor < 20) {
  say("It's getting dark now!")
}
'''

compiler = EveScriptCompiler()
compiled_script = compiler.compile(script)

decompiler = EveScriptDecompiler()
decompiled_script = decompiler.decompile(compiled_script)

print(decompiled_script)
```

CHAPTER

FOUR

REFERENCE

4.1 EveScript Language Reference

4.1.1 Basic Syntax

The basic form of an EveScript is as follows:

```
# if statement
if (expression) {
  action1(...)
  action2(...)
}

if (expression) {
  ...
}

# call action directly
action3()
```

An exmaple script may look like this:

```
if ($currentTime matchCron "0 0 * * *" && $lightSensor > 20 || $lightSensor < 10) {
    say("Only run on midnight 00:00")
    play("music.mp3")
}

# another condition
if ($currentTime matchCron "* * * * *") {
    say("run every minute")
    say(true)
}
```

In the above script, `$currentTime`, `$lightSensor` are called **variables**. They are the data sources of this script. And the `matchCron`, `>`, `||`, `<` are **operators**, where `matchCron` is a custom operator, while others are built-in operators. And the `say`, `play` are **actions**.

The *if* statement can be nested too.

```
# nested if
if (expr1) {
  if (expr2) {
    action()
```

(continues on next page)

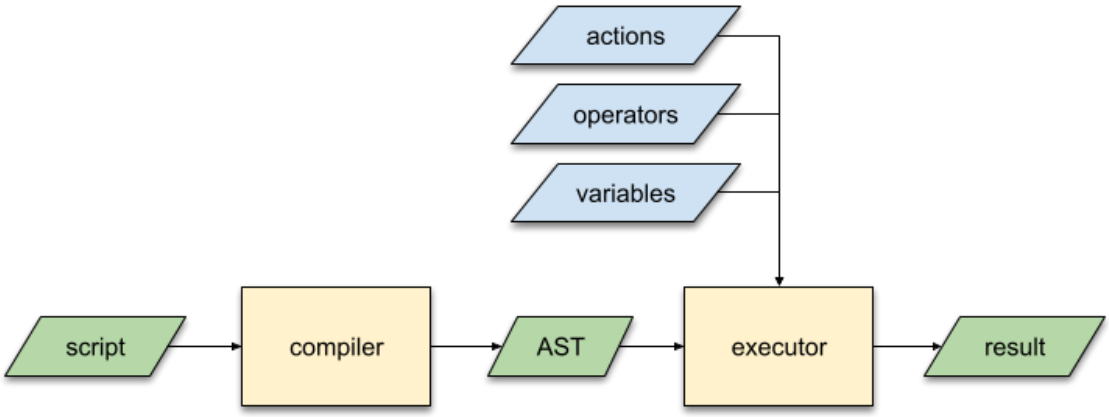
(continued from previous page)

```
  }
}
```

You can also use *if... else* statement.

```
# if else statement
if (expr1) {
  action1()
} else if (expr2) {
  action2()
} else {
  action3()
}
```

The following figure shows how a script is compiled and executed.



All variables, actions, and custom operators must be provided when executing this script. See *Actions*, *Operators*, and *Variables* for details.

4.1.2 Literals

EveScript supports the following types for literals:

- Numbers: Only integers (`3`, `10`, `-4`) and decimals (`0.6`, `-2.4`) are supported. Scientific notation such as `3e10` is **NOT** supported.
- String: A string must be quoted with double quotation marks: `"Hello, world"`. Single quotation mark is not supported.
- Boolean: `true` or `false`.

4.1.3 Statements

Statements are similar to the *if statements* in other languages. It uses the C-style:

- criteria must be placed in parenthesis
- use `{ }` around the code block (i.e. actions)

4.1.4 Variables

Variables are part of the expressions in the statement. When creating a `EveScriptExecutor`, variables must be provided so that they can be used in the script.

Contrast to the usual concept of "variable" in other languages, variables in EveScript are *NOT* memory units for storing values. They are more like small functions that act as data sources, such as reading data from hardware I/O ports, reading the system clock, or fetching data from Internet.

When creating an executor, provide a dict as the `variables` field of config, in which keys are the variable names and the values are the functions that provide data. The functions must take no parameters, and return a number, a string, or a boolean value.

The following code shows how to create a variable:

```
# Python code
def lightSensor():
  # read the sensor value from the light sensor
  return read_value_from_light_sensor()

# the key of the dict is the name of the variable
variables = {
    '$lightSensor': lightSensor,
}

executor = EveScriptExecutor({
    'variables': variables,
    'actions': {},
    'operators': {},
})
```

With this definition, the variable `$lightSensor` is useable in the script.

```
# EveScript code
if ($lightSensor > 10) {
    ...
}
```

4.1.5 Operators

EveScript provides some built-in operators in order to construct expressions.

Operator	Description
`||`	logical OR
`&&`	logical AND
`!`	logical NOT
`==`	equal to
`!=`	not equal to
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to

However, you can also define custom operators to implement your own logical operations. Similar to variables, custom operators are also small functions or lambdas that take **two parameters** (i.e. custom operators must be binary operators), and return a **boolean** value.

The following code snippet demonstrates how to create and use a custom operator:

```
# Python code
from datetime import datetime
from croniter import croniter

def matchCron(t, cron):
    """An operator that matches the provided time `t` with the `cron` string."""
    return croniter.match(cron, t)

def currentTime():
    """A variable that returns current system time."""
    return datetime.now()

executor = EveScriptExecutor({
    'variables': { '$currentTime': currentTime },
    'operators': { 'matchCron': matchCron },
    'actions': {},
})
```

With this definition, `$currentTime` and `matchCron` can be used in the code to implement a crontab-like trigger:

```
# EveScript code
if ($currentTime matchCron "0 0 * * *") {
    ...
}
```

The following table lists the precedence of operators.

Precedence	Operators
1	`!`
2	`<`, `<=`, `>`, `>=`, `==`, `!=`, all custom operators
3	`&&`
4	`||`

4.1.6 Actions

Actions are the functions listedn in the `{ }` block. They must be defined and provided when instantiating the `EveScriptExecutor`.

An action function can take zero or more parameters, and has no return value.

Note there is no semicolon `;` at the end of each action.

The following code snippet shows how to define an action:

```
# Python code
def lightSensor():
  # read the sensor value from the light sensor
  return read_value_from_light_sensor()

def say(text):
  """Define an action `say` that prints a message on the console."""
  print(text)

# the key of the dict is the name of the variable
variables = {
    '$lightSensor': lightSensor,
}

executor = EveScriptExecutor({
    'variables': { '$lightSensor': lightSensor },
    'actions': { 'say': say },
    'operators': {},
})
```

With this definition, `say(text)` can be called in the scripts:

```
# EveScript code
if ($lightSensor > 10) {
    say("It's daytime now!")
}
```

4.1.7 EBNF Definition

```
<script> ::= { <statement> }

<block>  ::= "{" { <statement> } "}"
           |  <statement>

<statement> ::= <if_statement>
              |  <action>

<if_statement> ::= "if" "(" <expr> ")" <block> [ "else" <block> ]

<expr>   ::= <term> "||" <expr>
           |  <term>

<term>   ::= <factor> "&&" <term>
           |  <factor>

<factor> ::= "(" <expr> ")"
```

(continues on next page)

(continued from previous page)

```
          | "!" <factor>
          | <predicate>

<predicate> ::= <operand> <operator> <operand>
             | <boolean>

<operator>  ::= ">"
              |  ">="
              |  "<"
              |  "<="
              |  "=="
              |  "!="
              |  keyword

<operand>   ::= variable
              |  <const>

<const>     ::= string
              |  number
              |  <boolean>

<boolean>   ::= 'true'
              |  'false'


<action>    ::= keyword "(" <params> ")"

<params>    ::= <param> { "," <param> }
              | empty

<param>     ::= <operand>
```

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

EveScript could always use more documentation, whether as part of the official EveScript docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/charlee/evescript/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *evescript* for local development:

1. Fork evescript (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:YOURGITHUBNAME/evescript.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER

SIX

AUTHORS

- Charlee Li - https://charlee.li

CHAPTER

SEVEN

CHANGELOG

7.1 0.5.0 (2021-03-29)

- Support if. . . else statement.
- Support executing actions outside if statement.
- Support nested if statement.
- Support actions with no params.

7.2 0.4.0 (2021-03-27)

- Added support for using true/false as conditions.

7.3 0.3.0 (2021-03-20)

- Fixed typo (EveScriptExector => EveScriptExecutor)

7.4 0.2.1 (2021-03-20)

- Added decompiler.
- Compiler will not rename built-in operators.

7.5 0.1.1 (2021-03-19)

- First release on PyPI.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search