-blocks Documentation

Release 0.1

The -blocks developers

Dec 05, 2017

Contents

λ-blocks is a framework that allows developers to write data processing programs without writing code. For that purpose, it uses a graph representation of a program (often in the form of an ETL) written in YAML, which links together blocks of Python code.

In this scenario, the code blocks are the vertices of the computation graph, while the links between them represent how computed data "flows" from one block to the other.

This provides many benefits, among them are **ease of writing** (in the simple YAML description format), **code deduplication** (by reusing code blocks in different computation graphs, or by embedding sub-graphs into bigger ones), and the ability to **reason about a graph**, to optimize or debug it for example (through the use of plugins). While λ-blocks comes with some batteries included (a collection of blocks and plugins), it is easy (and encouraged) to add your own, in order to fit the purposes of your organization of the range of problems you're solving.

We encourage you to read the tutorial, to learn how to write and execute your programs with λ-blocks, before diving in the available block collections and plugins.

CHAPTER 1

Tutorial

In this tutorial, we go through the steps of installing λ-blocks, writing a computation graph, and proceed to execute it.

1.1 Installation

1.1.1 Dependencies

If you're using Debian, Ubuntu, or a system of this family, the required dependencies should all be available in your package manager:

```
apt install python3 python3-yaml
```

Also not required for this tutorial, these dependencies are needed for some blocks in the included library:

```
apt install python3-matplotlib python3-requests-oauthlib
```

If you're not using Debian or a Debian-based system, be sure to install Python 3, and PyYAML through *pip*.

Finally, if you want to use the Spark blocks, you will need Spark and Pyspark to be installed on your system (but this is not required for this tutorial).

1.1.2 λ-blocks

While λ-blocks is still in its early days of development, it is not available through *pip*, nor in any distribution package manager. Therefore, you can install it this way:

```
git clone https://github.com/lambdablocks/lambdablocks.git
cd lambdablocks
python3 setup.py install
```

1.1.3 Verification

Try executing:

```
blocks.py --help
```

If you get the help page of this executable, all is set!

1.2 Writing a computation graph

Now that everything is installed, let's dive into writing a first λ-blocks program.

Such a program, also called a computation graph, is written in YAML, a simple data representation format. Create a new file and name it *wordcount.yml*: it will contain the description of a computation graph to perform a Wordcount. Add this content:

```
---
name: wordcount
description: Counts words
---
- block: cat
  name: cat
  args:
    filename: examples/wordlist
```

This YAML file contains two parts: the first one is a key/value list giving information on the computation graph (such as its name and description). The second part is more interesting: it contains the list of the code blocks that are the vertices of our graph. For now, there is only one vertice: it uses the block *cat* from the *lb.blocks.unixlike* blocks library. It has a unique name *cat* (since we use only once the block *cat* in this program, the vertice name can be the same as the block name), and one argument, a path to a file. As you may have guessed, this block acts like the Unix *cat* utility: it reads a file.

This program won't do much, except for reading a file. You can try to execute it this way:

```
blocks.py -f wordcount.yml
```

If nothing happens, it is normal: the file has been read by λ-blocks, but it isn't supposed to be displayed on the console. If you get an error, the path you provided may be incorrect: be sure to execute the command within in the *lambdablocks* folder, or to change the *filename* argument.

Let's add a few vertices in our graph, and link them together to compute a Wordcount implementation:

```
---
name: wordcount
description: counts words
---
- block: cat
  name: cat
  args:
    filename: examples/wordlist

- block: group_by_count
  name: group
  inputs:
    data: cat.result
```

```
- block: sort
  name: sort
  args:
    key: "lambda x: x[1]"
    reverse: true
  inputs:
    data: group.result

- block: show_console
  name: show
  inputs:
   data: sort.result
```

We now have 4 blocks (or vertices):

- *cat* reads a file and outputs a list of lines found in this file;
- *group_by_count* reads a list, and outputs a list of unique items, along with the number of times they appear in the list;
- *sort* reads a list, and outputs a sorted list, sorted by the second item of each element;
- *show_console* displays its inputs on the user console.

A block has named inputs and named outputs. To link two blocks together, we specify the inputs of a block in the *inputs* key. For example, the block *group_by_count* takes one input, *data*, that is the output *result* of the block *cat*.

Let's try to execute this graph:

```
blocks.py -f wordcount.yml
```

That's it! You should get a list of fruits, along with their number of occurences.

1.3 Using plugins

λ-blocks, while processing a computation graph, can execute plugins, which are pieces of Python code able to act on the graph. For example, let's try the included *debug* plugin:

```
blocks.py -f wordcount.yml -p debug
```

This plugin will display an excerpt of the results produced by each block, which allows you to effectively see what every block is doing. This is useful to follow the data as it is transformed from the entry of the graph to all the following vertices.

You can also try to execute the *instrumentation* plugin the same way, which will measure the time taken by every block to compute, useful to detect bottlenecks:

```
blocks.py -f wordcount.yml -p debug instrumentation
```

Unsurprisingly, the *cat* block should be the slowest, because it requires to read a file on disk.

CHAPTER 2

Information for developers

2.1 Dependencies

Install the development dependencies:

```
apt install python3-nose2 python3-nose2-cov python3-sphinx
```

2.2 Tests

To run the tests:

```
PYTHONPATH=. make test
```

2.3 Development mode

To install the package in development mode, you can use:

```
pip3 install -e .
```

CHAPTER 3

Examples

3.1 Simple unix-like pipes

The file *examples/unix.yml* parses the file */etc/passwd* on your system (if it exists), greps it for *root*, cuts some fields, keeps only the last line, and finally displays the result:

```
blocks.py -f examples/unix.yml
```

3.2 Over http

The graph located in *examples/http.yml* does fetches a file over http, filters it, and saves the result in a file:

```
blocks.py -f examples/http.yml
```

3.3 Twitter Wordcount with sub-graph

The graph in *examples/twitter_wordcount.yml* will count the most used hashtags on the AFP timeline. For that purpose, it uses a sub-graph, located in *examples/topology-wordcount.yml*, which does the counting words part.

To make it work, you first need to fill in your Twitter API credentials, and then:

```
blocks.py -f examples/twitter_wordcount.yml
```

3.4 Wordcount with Spark and a sub-graph

The graph in *examples/spark_wordcount.yml* does a Wordcount over a file, using a sub-graph for counting words. It will display the result both in the console and in a matplotlib plot.. You need to install Matplotlib, Spark and Pyspark,

and then:

```
PYSPARK_PYTHON=python3 blocks.py -f examples/spark_wordcount.yml
```

CHAPTER 4

Available blocks

CHAPTER 5

Available plugins

5.1 cache_disk

5.2 debug

5.3 instrumentation

CHAPTER 6

Internal API

CHAPTER 7

Indices and tables

- genindex
- modindex
- search