Expertmaker Accelerator

Quick Install using the "Project Skeleton" Repository + Example

Introduction and System Requirements

The `accelerator-project_skeleton` project provides a simple and convenient way to install the Accelerator locally. This document lists the necessary steps to set up the Accelerator using it. The last part of the document is dedicated to an example showing important Accelerator concepts.

The Accelerator will run on almost any hardware, from small laptops to large multi-CPU rack servers. It is assumed in this manual that the computer is running Ubuntu 16.04 LTS or Debian 9. The Accelerator team is actively testing on Ubuntu, Debian, and FreeBSD, but the Accelerator will most likely run on many other Linux distributions as well.

Installation

There are three steps in the installation: resolve dependencies, clone repository, and run the initiation script. These steps will be described next.

1. Dependencies

The first step is to make sure that all software package dependencies are met. This command will install all required packages

```
sudo apt-get install build-essential python-dev python3-dev zlib1g-dev git virtualenv
```

The installer requires only `git`, `virtualenv`, and some `dev` packages in order to compile C-code.

2. Clone Repository

Clone the `accelerator-project_skeleton` like this

```
git clone https://github.com/eBay/accelerator-project_skeleton.git
```

3. Setup

The Accelerator will now be installed *locally without any administrator privileges*. To continue, `cd` into the cloned directory

```
cd accelerator-project_skeleton
```

In this directory there is a file `init.py` that performs all the installation steps. It will work out-of-the-box, but for a customized install it is recommended to read and modify this file before continuing. The next step is to run the script

```
./init.py
```

This will do a complete setup, and the next section provides more information about the process. After the script is finished, the Accelerator installation is complete. It could be run by issuing

```
cd accelerator
./daemon.py
```

The first time the Accelerator is run, it will compile some functions written in the C programming language. On some systems, this process may generate a few warning messages, but that is okay. Setup is now complete.

Accessing Libraries in the Virtual Environment

Since the installer uses a virtual environment for installing packages, it must be initiated in all shells running for example `dsinfo`. This is done by issuing

```
source ../venv/py3/bin/activate
```

from the `accelerator` directory.

Overview of the Installation

The `accelerator-project_skeleton` script `init.py` will setup virtual environments for Python2 and Python3. In these virtual environments, it will download and install some depending packages, and `git clone` and install the `accelerator-gzutil` library. The Accelerator itself is `git cloned` into a git submodule in the `accelerator` directory.

The default configuration file is located in `conf/framework.conf`. This file is used to specify workdirs, method directories, and more. For more information, see the Accelerator User's Reference Manual.

References

`https://berkeman.github.io/pdf/acc_manual.pdf`

Example 1: Dataset Operations

This example shows how to create datasets and dataset chains, export datasets to CSV (Comma Separated Values) files, import datasets from CSV-files, append columns to datasets, and more.

Running the Example

The example is located in the `example1` directory. Here is a list of steps showing how to run it:

1. Clone and setup the `project_skeleton` as described earlier in this document

   ```
   git clone https://github.com/eBay/accelerator-project_skeleton.git
   cd accelerator-project_skeleton
   ./init.py
   ```

2. Now we can start the Accelerator. To start the server type

   ```
   cd accelerator
   ./daemon.py
   ```

   This terminal is now our *server* terminal, displaying the Accelerator's `stdout` and `stderr`.

3. Next, open a second terminal emulator for the *client* side. The Accelerator team prefers using GNU screen, but any terminal emulator like `xterm` is fine.

   In this second terminal (green), `cd` to the `accelerator` directory

   ```
   cd accelerator
   ```

   Since the installation is local (no administration privileges used), using virtual environments, we need to set up the virtual environment in this terminal like this

   ```
   source ../venv/py3/bin/activate
   ```

   Now we can run the the build script.

   ```
   ./automatarunner example1
   ```

Looking at the Output

The build script for this session is `example1/automata_example1.py`. Please have a look at this code with one eye while looking at the output with the other. Here is the output:

```
example1.automata_example1
        -  example1_create_dataset                          DONE  TEST-0  0.4 seconds
        -  example1_create_dataset                          DONE  TEST-1  0.4 seconds
        -  example1_create_dataset                          DONE  TEST-2  0.5 seconds
        -  example1_create_dataset                          DONE  TEST-3  0.4 seconds
        -  example1_create_dataset                          DONE  TEST-4  0.4 seconds
        -  csvexport                                        DONE  TEST-5  0.2 seconds
Exported file stored in "/home/ab/accelerator/workdirs/TEST/TEST-5/random.tsv"
```

This shows that five `example1_create_dataset` jobs have been run, and their jobids are `TEST-0` to `TEST-4`. By looking at the code, we see that the `csvexport` job is exporting the *last* (`TEST-4`) dataset to a CSV file on disk. (The whole dataset chain could be exported to CSV too by simply changing an input parameter to `csvimport`.) The location of this file is printed to `stdout` as well. We move on to

```
        -  csvimport                                        DONE  TEST-6  0.2 seconds
        -  dataset_type                                     DONE  TEST-7  0.2 seconds
```

Here, we've *imported* the CSV-file we just created. Note that an import types the data to `bytes`, so we issue an `dataset_type` job that does proper typing of the data. Next,

```
        -  example1_calc_average                          DONE  TEST-8  0.0 seconds
Column rint:  sum=-220338.000000, length=100000, average=-2.203380
        -  example1_calc_average                          DONE  TEST-9  0.0 seconds
Column rflt:  sum=50109.285978, length=100000, average=0.501093
```

there is a loop iterating over the columns of the dataset in `TEST-7`. In each iteration, it will compute the average of the values of the column and print it to `stdout`. Finally,

```
        -  example1_add_column                            DONE  TEST-10  0.2 seconds
        -  csvexport                                      DONE  TEST-11  0.3 seconds
```

appends a new column to the `TEST-7` dataset. This dataset is then exported to a CSV file. Which labels to export is explicitly set in this case. The build script then prints out a pretty-printed version of what is in the current Urd list

```
JobList(
   [  0]      Created_number_0 : TEST-0
   [  1]      Created_number_1 : TEST-1
   [  2]      Created_number_2 : TEST-2
   [  3]      Created_number_3 : TEST-3
   [  4]      Created_number_4 : TEST-4
   [  5]             csvexport : TEST-5
   [  6]             csvimport : TEST-6
   [  7]          dataset_type : TEST-7
   [  8] example1_calc_average : TEST-8
   [  9] example1_calc_average : TEST-9
   [ 10]   example1_add_column : TEST-10
   [ 11]             csvexport : TEST-11
)
```

Here we can see that the first five jobs have been given explicit names that makes it possible to uniquely identify them.

A Look at the Datasets

The `dsinfo` command is a simple tool to list the most important aspects of a dataset. Let's examine `TEST-4`, which is the last dataset in a chain

```
./dsinfo.py TEST-4
```

this results in

```
Parent: None
Hashlabel: None
Columns:
    rflt  float64
    rint  int64
2 columns
100,000 lines
Chain length 5, from TEST-0 to TEST-4
500,000 total lines
```

which shows that we have 100.000 lines in `TEST-4`, and 500.000 lines in the chain starting at `TEST-0`. Furthermore, it has two columns, one floating point and one integer, and the dataset is not hashed.

If we look at `TEST-10`, which is the dataset with a column appended to `TEST-7`, it looks like this

```
Parent: TEST-7/default
Hashlabel: None
Columns:
    prod  number
    rflt  number
    rint  number
3 columns
100,000 lines
```

Indeed, this dataset has a parent, `TEST-7/default`, and there are three columns, all typed to `number`. Similarly, it is worth looking at the imported dataset `TEST-6` too.

Conclusion

This example shows how to do some important dataset operations, such as importing data from a CSV file, exporting a dataset to a CSV file, typing a dataset, creating a chain of datasets, iterating over a dataset and compute something, and appending a new column to an existing dataset. It also shows how to find the absolute path to a job result, how to access a dataset's column names, and more. The idea is that this example could provide a way to start playing with the Accelerator and maybe use it in future projects.

Acknowledgements

Thanks to Stefan Håkonsson for suggestions, testing, and proof reading.