Table of Contents

The Bender tutorials 

These are tutorials for Bender application: "User-friently python analysis environment for LHCb".

It is the first attempt to convert existing TWiki-based tutorials to GitHub platform, inspired by the great success of LHCb StarterKit lessons.

Bender is LHCb Python-based Physics Analysis Environment. It combines the physics content of DaVinci-project with the interactive python abilities provided by GaudiPython. It also could be considered as "Interactive LoKi". The major functionality comes from ROOT/Reflex dictionaries for the basic C++ classes and the interfaces.
These dictionaries are used primary for POOL persistency and effectively reused for interactivity. The main purpose of top-level scripts is the coherent orchestration of the Reflex dictionaries and the proper decoration of the available interfaces.

Bender dependencies are sketched here:dependencies

Doxigen documentation for Bender is accessible here.

It is assumed that users are already has *some* knowledge of LHCb software, in particular DaVinci and are familiar with LHCb Starterkit.

You can also add relative links within the website like this one to the first section!

Contributing

bender-tutorials is an open source project, and we welcome contributions of all kinds:

- New lessons;
- Fixes to existing material;
- Bug reports; and
- Reviews of proposed changes.

By contributing, you are agreeing that we may redistribute your work under these licenses. You also agree to abide by our contributor code of conduct.

Getting Started

1. We use the fork and pull model to manage changes. More information about forking a repository and making a Pull Request.
2. To build the lessons please install the dependencies.
3. For our lessons, you should branch from and submit pull requests against the `master` branch.
4. When editing lesson pages, you need only commit changes to the Markdown source files.
5. If you're looking for things to work on, please see the list of issues for this repository. Comments on issues and reviews of pull requests are equally welcome.

Dependencies

To build the lessons locally, install the following:

1. Gitbook

Install the Gitbook plugins:

```
$ gitbook install
```

Then (from the bender-tutorials directory) build the pages and start a web server to host them:

```
$ gitbook serve
```

You can see your local version by using a web-browser to navigate to `http://localhost:4000` or wherever it says it's serving the book.

Getting started

Click on the "Examples of formatting" section on the left

The title

Learning Objectives

- The starterkit lessons all start with objectives about the lesson
- Objective 2 with some *formatted* **text** ~~like~~ `this`

Basic formatting

You can make **bold**, *italic* and ~~strikethrough~~ text. Add relative links like this one and absolute links in a couple of different ways.

Have bulleted lists:

- Point 1
- Point 2
  - Sub point
    - Sub point
  - Sub point
- Point 2

Use numbered lists:

1. First
2. Second
   i. Second first
      i. Second first first
   ii. Second second
3. Third

LaTeX

You can use inline LaTeX maths such as talking about the decay $$D^{* +} \rightarrow \left(D^0 \rightarrow K^{-} \pi^{+} \right)$$.

Code highlighting

And have small lines of code inline like saying `print("Hello world")` or have multiple lines with syntax highlighting for python:

```
import sys

def stderr_print(string):
    sys.stderr.write(string)

stderr_print("Hello world")
```

bash:

```
lb-run Bender/latest $SHELL
dst_dump -f -n 100 my_file.dst 2>&1 | tee log.log
```

and more!

Callouts

Prequisites

- Prequisite 1
- Prequisite 2

Objectives

- Objective 1
- Objective 2

Challenge

Set a challenge here, and the solution will remain hidden until it's clicked

- How to print?

Solution

The answer is:

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

Extra details that are hidden by default

Some extra details

Keypoints

- Summary point 1

- Summary point 2

Quotes

> This was said by someone

Tables

Simple tables are possible

First Header	Second Header
Content from cell 1	Content from cell 2
Content in the first column	Content in the second column

Images

Section types

This is a section

Subsections

And a subsection

Subsubsections

And a subsubsection

The first two almost *useless*, but highly *illustrative* examples

Learning objectives

- Understand the overall structure of Bender *module* using oversimplified examples

Do-nothing

Any *valid* Bender module must have two essential parts

- function `run` with the predefined signature
- function `configure` with the predefined dignature

For the most trivial (*"do-nothing"*) scenario function `run` is

```
def run ( nEvents ) :
    # some fictive event loop
    for i in range( 0 , min( nEvents , 10 ) ) :  print ' I run event %i ' % i
    return 0
```

In a similar way, the simplest *"do-nothing"*-version of `configure` -function is

```
def configure (  datafiles ,  catalogs = [] , castor = False , params = {} ) :
    print 'I am configuration step!'
    return 0
```

As one clearly sees, these lines do nothing useful, but they are perfectly enough to be classified as the first Bender code. Moreover, the python module with these two function can already be submitted to Ganga/Grid, and Ganga will classify it as valid Bender code. Therefore this code is already *"ready-for-Ganga/Grid"*!

The details for the curious students: how Ganga/Grid treat Bender modules?

Actually Ganga executes at the remote node the following wrapper code

```
files    = ... ## this one comes from DIRAC
catalogs = ... ## ditto
params   = ... ## extra parameters (if needed): this comes from the user
nevents  = ... ## it comes from Ganga configuration

import USERMODULE   ## here it imports your module!
USERMODULE.configure ( files , catalogs ,  params = params )
USERMODULE.run       (  nevents )
```

Thats all! From this snippet you see:

- the code must have the structure of python *module*, namely no executable lines should appear in the main body of the file
  - (note the difference with respect to the *script*)
- it must have two functions `run` and `configure`
  - (everythnig else is not used)

The whole module is here:

```
## 1) some user code :
def run ( nEvents ) :
    for i in range( 0 , min( nEvents , 10 ) ) : print ' I run event %i ' % i
    return 0

## 2) configuration step
def configure (  datafiles ,  catalogs = [] , castor = False , params = {} ) :
    print 'I am configuration step!'
    return 0
```

DoNothing.py hosted with ❤ by **GitHub** **view raw**

In practice, before the submission the jobs to Ganga/Grid, the code needs to be tested using some test-data. This, formally unnesessary, but very important step can be easily embedded into your module using python's `__main__` clause:

```
if '__main__' == __name__ :
    print 'This runs only if module is used as the script! '
    configure ( [] , catalogs = [] , params = {} )
    run ( 10 )
```

Note that these lines effectively convert the *module* into *script*, and finally one gets:

```
## 1) some user code :
def run ( nEvents ) :
    for i in range( 0 , min( nEvents , 10 ) ) : print ' I run event %i ' % '
    return 0
## 2) configuration step
def configure (  datafiles ,  catalogs = [] , castor = False , params = {} ) :
    print 'I am configuration step!'
    return 0
## 3) steer the job
if '__main__' == __name__ :
    print 'This runs only if module is used as the script!'
    run ( 10 )
```

DoNothing.py hosted with ❤ by **GitHub** **view raw**

How to run it interactively?

The answer is trivial:

```
lb-run Bender/prod python DoNothing.py
```

That's all. Make a try and see what you get!

Unnesessary but very useful decorations:

It is highly desirable and *recommended* to put some *"decorations"* a top of this minimalistic lines:

- add magic `#!/usr/bin/env python` line as the top line of the module/script
- make the script executable: `chmod +x ./DoNothing.py`
- add a python documentation close to the begin of the script
  - fill some useful python attributes with the proper informaton
    - `__author__`
    - `__date__`
    - `__version__`
  - do not forget to add documenation in Doxygen-style and use in comments following tags
    - `@file`
    - `@author`
    - ...

With all these decorations the modules is here

For all subsequent lessons we'll gradually fill this script
with the additional functionality, step-by-step converting it to something much more useful.

In practice, ...

In practice, the prepared and *ready-to-use* function `run` is imported from some of central Bender modules, namely `Bender.Main` and the only one really important task for the user is to code the function `configure` . The `__main__` clause usually contains some input data for local tests:

```
if __name__ == '__main__' :
    inputdata = [
        '/lhcb/LHCb/Collision12/PSIX.MDST/00035290/0000/00035290_00000221_1.psix.mdst' ,
        '/lhcb/LHCb/Collision12/PSIX.MDST/00035290/0000/00035290_00000282_1.psix.mdst' ]
    configure( inputdata , castor = True )
    ## the event loop
    run(10000)
```

Hello, world!

Sections

Click on the subsection on the left to see the subsubsection

A subsection

Learning Objectives

- Objective 1
- Another objective

A subsubsection

Learning Objectives

- Objective 1
- Another objective