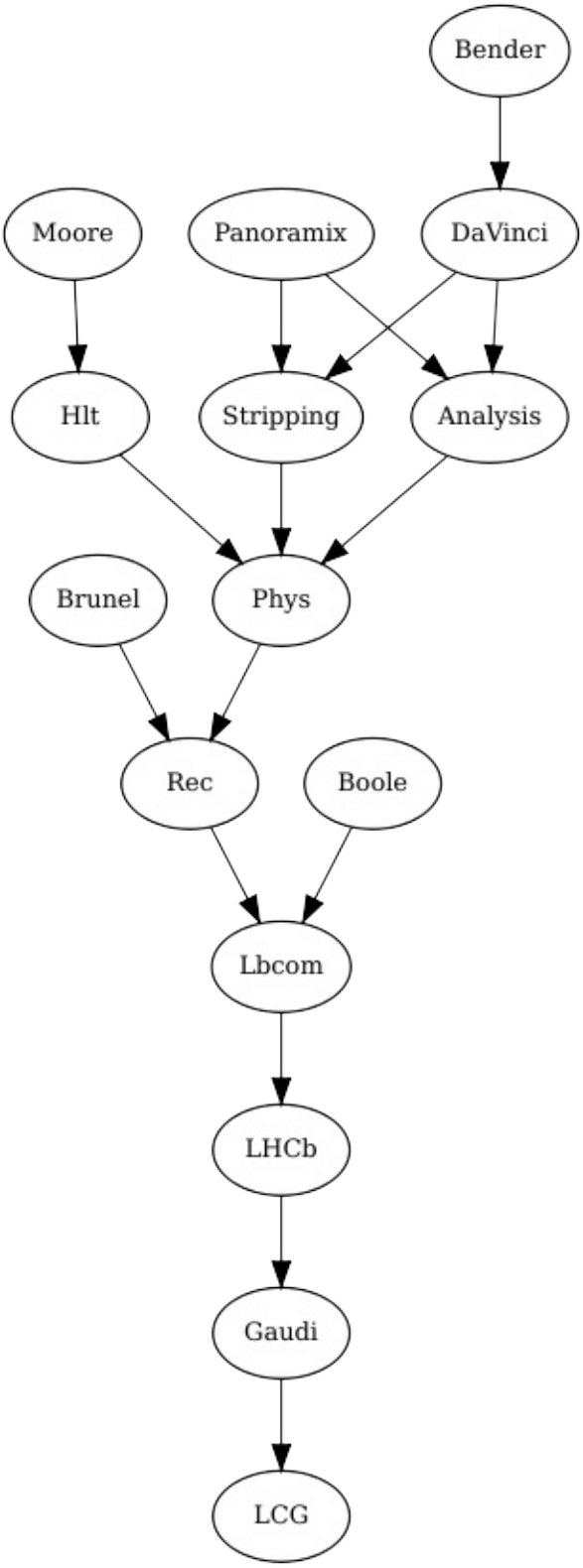
Table of Contents

The Bender tutorials 

These are tutorials for Bender application: "User-friendly 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:

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!

Getting started

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

The first two *almost useless*, but *very important* examples

Learning objectives

- Understand the overall structure of Bender *module* and the configuration of the application

Do-nothing

Learning objectives

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

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 complete module is here

For all subsequent lessons we'll gradually extend 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 the main Bender module `Bender.Main`, and the only one really important task for the user is to code the function `configure`.

DaVinci

Learning objectives

- Understand the internal structure of the *configure* function

For the *typical* case in practice, the function *configure* (as the name suggests) contains three parts

1. *static configuration*: the configuration of `DaVinci` configurable (almost unavoidable)

2. *input data and application manager*: define the input data and instantiate Gaudi's application manager (mandatory)
3. *dynamic configuration*: the configuration of `GaudiPython` components (optional)

Static configuration

For the first part, the instantiation of DaVinci configurable is almost unavoidable step:

```
from    Configurables import DaVinci
rootInTES = '/Event/PSIX'
dv = DaVinci ( DataType   = '2012'    ,
               InputType  = 'MDST'    ,
               RootInTES  = rootInTES )
```

Here we are preparing application to read `PSIX.MDST` - uDST with few useful selections for B&Q Working Group. Note that in this part one can use all power of DaVinci/Gaudi `Congifurables` . In practice, for physics analyses, it is veyr convinient to use here `Selection` framework, that allows to configure `DaVinci` in a very compact, safe, robust and nicely readable way, e.g. let's get from Transient Store some `selection` and print its content

```
from PhysConf.Selections import AutomaticData,  PrintSelection
particles = AutomaticData  ( 'Phys/SelPsi2KForPsiX/Particles' )
particle  = PrintSelection ( particles )
```

As the last sub-step of (1), one needs to pass the selection object to `DaVinci`

```
dv.UserAlgorithms.append ( particles )
```

Where is `SelectionSequence` ?

The underlying `SelectionSequence` object will be created automatically. You should not worry about it.

Input data and application manager

This part is rather trivial and almost always standard:

```
from Bender.Main import setData, appMgr
## define input data
setData  ( inputdata , catalogs , castor )
## instantiate the application manager
gaudi = appMgr()  ## NOTE THIS LINE!
```

while `setData` can appear anywhere inside `configure` function, the line with `appMgr()` is very special. After this line, no *static configuration* can be used anymore. Therefore all the code dealing with `Configurables` and `Selections` must be placed above this line.

Dynamic configuration

For this particular example, it is not used, but will be discussed further in conjunction with other lessons.

The complete `configure` function is:

```
1  from Bender.Main import setData, appMgr, SUCCESS
2  ## The configuration of the job
3  def configure ( inputdata        ,    ## the list of input files
4                  catalogs = []    ,    ## xml-catalogs (filled by GRID)
```

```
                castor   = False ,    ## use the direct access to castor/EOS ?
                params   = {}    ) :

    ## import DaVinci & configure it!
    from Configurables import DaVinci
    ## delegate the actual configuration to DaVinci
    rootInTES = '/Event/PSIX'
    dv = DaVinci ( DataType   = '2012'    ,
                   InputType  = 'MDST'    ,
                   RootInTES  = rootInTES )

    from PhysConf.Selections import AutomaticData, PrintSelection
    particles  = AutomaticData( 'Phys/SelPsi2KForPsiX/Particles' )
    particles  = PrintSelection ( particles )
    dv.UserAlgorithms.append ( particles  )

    ## define the input data
    setData  ( inputdata , catalogs , castor )

    ## get/create application manager
    gaudi = appMgr()
    return SUCCESS
```

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

The prepared and *ready-to-use* function `run` is imported `Bender.Main`:

```
from Bender.Main import run
```

Now our Bender module (well, it is actually pure `DaVinci`, no real Bender here!) is ready to be used with Ganga/Grid. For local interactive tests we can use the trick with `__main__` clause: The `__main__` clause in our case 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)
```

The complete moodule can be accessed here

How to run it?

Again, the answer is trivial (and universal):

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

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

Challenge

Try to convert any of your existing `DaVinci` simple *script* into Bender *module* and run it interactively. You can use the result of this excersize for subsequent lessons.

What is `castor` ? Why `LFN` is used as input file name?

Bender is smart enough, and for many cases it can efficiently convert input `LFN` into the real file name.

1. First, if you have Grid proxy enabled (`lhcb-proxy-init`) is uses internally `LHCbDirac` to locate and access the file. This way is not very fast, but for all practial cases this look-up is almost always successful, however for some cases certain hints could be very useful. In particular, you can specify the list of Grid sites to look for data files:

   ```
   ## define input data
   setData  ( inputdata , catalogs , castor = castor ,  grid = ['RAL','CERN','GRIDKA'] )
   ```

2. Second, for CERN, one can use option `castor = True` , that activates the local look-up on input files at CERN-CASTOR and CERN-EOS storages (`root://castorlhcb.cern.ch` and `root://eoslhcb.cern.ch`). This look-up is much faster than the first option, but here the success is not guaranteed, since not all files have their replicas at CERN.
3. Third, for access to special locations, e.g. some local files, Bender also makes a try to look into directories specified via the environment variable `BENDERDATAPATH` (column separated list of paths) and also try to contruct the file names using the content of environment variable `BENDERDATAPREFIX` (semicolumn separated list of prefixes used for constryction the final file name). The file name is constructed using all `(n+1)*(m+1)` variants, where `n` is number of items in `BENDERDATAPATH` and `m` is number of items in `BENDERDATAPREFIX` . Using the combination of `BENDERDATAPATH` and `BENDERDATAPREFIX` variables one can make very powerful matching of *short* file names (e.g. LFN) to the actual file. Using these variables one can easily perform a local and efficient access to Grid files from some *close* Tier-1/2 center.

Keypoints

Wth these two examples, you should aready be able to

- code sove *valid* (but useless) Bender modules
- run them interactively

The first Bender algorithms

Prequisites

- One needs to undertand the stucture of Bender *module* : `run` , `configure` functions and the `__main__` clause
- One needs to know the sctructure and the content of `configure` function

Learning objectives

- Understand Bender algorithms
  - How to code them?
  - How to embedd them into the application?

Hello, world!

Traditionally for tutorials, the first algorithm prints `Hello, world` . The Bender algorithm inherits from the class `Algo` , imported from `Bender.Main` module. This python base is indeed a `C++` -class, that inherits from `LoKi::Algo` class, that in turn inherits from `DaVinciTupleAlgorithm` . The simplest algorithm is rather trivial:

```
from Bender.Main iport Algo, SUCCESS
class HelloWorld(Algo):
    """The most trivial algorithm to print 'Hello,world!'"""
    def analyse( self ) :   ## IMPORTANT!
        """The main 'analysis' method"""
        print       'Hello, world! (using native Python)'
        self.Print( 'Hello, World! (using Gaudi)')
        return SUCCESS      ## IMPORTANT!!!
```

Important note:

- one *must* implement the method `analyse` that gets no argument and returns `StatusCode`

Optionally one can (re)implement other important methods, like `__init__` , `initialize` , `finalize` , etc... In particular `initialize` could be used to locate some *tools* and or pre-define some useful code fragments, e.g. some *expensive* or non-trivial LoKi-functors.

Where to put the algorithm code?

It is recommended to put the algorithm code directly in the main body of your module, outside of `configure` function. It allows to have visual separation of the algorithmic and configuration parts. Also it helps for independent reuse of both parts.

How to embedd the algorithm into the application ?

There are two approaches *brute-force*, that works nicely with such primitive code as `HelloWorld` algorithm above and the

intelligent/recommended approach, that smoothly insert the algorithm into the overall flow of algorithms, provided by `DaVinci`

Brute-force

One can instantiate the algorithm in **configure** method **after** the instantiation of application manager, and add the algorithm, into the list of top-level algorithms, known to Gaudi:

```
gaudi = appMgr()
alg   = HelloWorld('Hello')
gaudi.addAlgorithm( alg )
```

For this particular simple case one can also just replace the list of top-level Gaudi algorithms with a single `HelloWorld` algorithm:

```
gaudi = appMgr()
alg   = HelloWorld('Hello')
gaudi.setAlgorithms( [ alg ] )
```

More on an optional _dynamic configuration_

As it has been said earlier, the part of `configure` function, placed after `gaudi=appMgr()` line corresponds to *dynamic configuration*, and here one can continue the further configuration of the algorthm, e.g.

```
gaudi    = appMgr()
alg      = HelloWorld('Hello')
alg.QUQU = 'qu-qu!'  ## define and set some "parameter"
gaudi.setAlgorithms( [ alg ] )
```

Later, this new *parameter* can be accessed e.g. in `analyse` function:

```
class HelloWorld(Algo):
    """The most trivial algorithm to print 'Hello,world!'"""
    def analyse( self ) :   ## IMPORTANT!
        """The main 'analysis' method"""
        print       'Hello, world! (using native Python)', self.QUQU ## use "parameter"
        self.Print( 'Hello, World! (using Gaudi)')
        return SUCCESS      ## IMPORTANT!!!
```

Such trick is in general a bit fragile, but it is often useful if one has several instances of the algorithm that differ only by some configuration parameter.

```
alg1 = MyALG ( ... )
alg2 = MyALG ( ... )
alg3 = MyALG ( ... )
alg1.decay_mode = '[D0 -> K-  pi+]CC'
alg2.decay_mode = '[D0 -> K-  K+ ]CC'
alg3.decay_mode = '[D0 -> pi- pi+]CC'
```

This approach is very easy and rather intuitive, but is not so easy to insert the algorithm into existing non-trivial flow of algorithms without a danger to destroy the flow. In this way one destroys various standard actions, like (pre)filtering, luminosity calculation etc., therfore it could not be recommended for the real physics analyses, but it could be used for some simple special cases.

Intelligent approach

For *intelligent* approach one uses `Selection` wrapper for Bender algotithm, `BenderSelection`. This wrapper behaves as any other

selection-objects, and it lives in *static configuration* part of `configure` function:

```
from PhysConf.Selections import AutomaticData,  PrintSelection
particles = AutomaticData  ( 'Phys/SelPsi2KForPsiX/Particles' )
particle  = PrintSelection ( particles )

## configuration object for Bender algorithm:
#                             name   , input selections
hello = BenderSelection ( 'Hello' , inputs = [ particle ]  )
dv.UserAlgorithms.append ( hello )
```

As the next step in *dynamic configuration* part of `configure` function one instantiates the algoritm taking all the configuration from the selection-object:

```
gaudi = appMgr()
alg   = HelloWorld( hello )
```

To complete the module one (as usual) need to combine in the file

1. implementation of `HelloWorld` algorithm
2. `configure` function with proper *static* and *dynamic* configurations
3. `__main__` clause
4. (`run` function is imported from `Bender.Main` module)

The complete module can be accessed here

Get data, fill histos & n-tuples

Well, now your Bender algorithm knows how to print `Hello,world!` . Note that it also gets some data: in the previous example we fed it with `particles` -selection. Now try to get this data inside the algorithm and make first simpel manipulations with data.

`select` method

The method `select` is a heart of Bender algorithm. It allows to select/filter the particles that satisfies some criteria from the input particles. The basic usage is:

```
myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
```

The method returns collection filtered particles
The first argument is the tag, that will be associated with selected particles, the second argument is the selection criteria. The tag *must* be unique, and the selection criteria coudl be in a form of

- *LoKi predicate*: LoKi-functor that get the particle as argument and return the boolean value
- *decay descriptor*, e.g. `'Beauty --> J/psi(1S) K+ K-'` . Some components of the decay descriptor can be *marked*, and in this case, only the *marked* partcles will be selected:

  ```
  myB = self.select ( 'beauty' , 'Beauty --> J/psi(1S)  K+  K-')   ## get the heads of the decay
  myK = self.select ( 'kaons'  , 'Beauty --> J/psi(1S) ^K+ ^K-')   ## get only kaons
  ```

As soon as one gets some good, filtered particles there are many possible actions

- print it!

  ```
  myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
  print myB
  ```

- loop

```
myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
for b in myB :
 print 'My Particle:', p
 print 'some quantities: ', M(p) , PT(p) , P(p)
```

- fill histograms

```
myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
for b in myB :
 #          what        histo-ID    low-edge  high-edge  #bins
 self.plot( PT (p)/GeV , 'pt(B)'  , 0        , 20       ,  50  )
 self.plot( M  (p)/GeV , 'm(B)'   , 5.2      , 5.4      , 100  )
 self.plot( M1 (p)/GeV , 'm(psi)' , 3.0      , 3.2      , 100  )
 self.plot( M23(p)/GeV , 'm(KK)'  , 1.0      , 1.050    ,  50  )
```

- fill n-tuple:

```
myB = self.select ( 'myB' , ('B0' ==  ABSID ) | ('B0' ==  ABSID ) )
t = self.nTuple('TupleName')
for b in myB :
 t.column_float ( 'pt'    , PT (p)/GeV)
 t.column_float ( 'm'     , M  (p)/GeV)
 t.column_float ( 'm_psi' , M1 (p)/GeV)
 t.column_float ( 'm_kk'  , M23(p)/GeV)
 t.write()
```

For n-tuples...

Sicne n-tuples(ROOT's `TTree` objects) resides in ROOT-file, to use n-tuples, one also need to declare the output file for `TTree`s: The easiest way is to rely on `TupleFile` property of `DaVinci` :

```
dv = DaVinci ( DataType   = '2012'          ,
               InputType  = 'MDST'          ,
               TupleFile  = 'MyTuples.root' ,  ## SEE HERE !!!
               RootInTES  = rootInTES       )
```

Challenge

Add `select` statements, histos and tuples to your `HelloWorld` algorithm, created earlier, and run it.

Solution

The complete module is accesible here

Keypoints

Wth these two examples, you should aready be able to

- code Bender *algorithms* and insetr them into overall algorithm flow
- loop over data, fill historgrams and n-tuples

Create the compound particles in Bender

Prequisites

- One needs to understand the way how Bender accesses the data

Learning objectives

- Understand how Bender algorithm combines the particles and creates the compound particles

Make-B

The next example illustrates how one combines the particles and create the compound particles inside the Bender algorithm. Let's consider a simple case of creation of `B+ -> J/psi(1S) K+` decays.

The first step is rather obvious: before getting the combinations, we need to get the individual components. Here `select` function does the job:

```
## get J/psi mesons from the input
psis  = self.select ( 'psi' , 'J/psi(1S) -> mu+ mu-' )
## get energetic kaons from the input:
kaons = self.select ( 'k'  , ( 'K+' == ID ) & ( PT > 1 * GeV ) &  ( PROBNNk > 0.2 ) )
```

The loop over `psi k` combinations is ratehr trivial:

```
## make a loop over J/psi K combinations :
for b in self.loop( 'psi k' , 'B+' ) :
    ## fast evaluation of mass (no fit here!)
    m12 = b.mass ( 1 , 2 ) / GeV
    print 'J/psiK mass is %s[GeV]'  % m12
    p1  = b.momentum ( 1 ) / GeV
    p2  = b.momentum ( 2 ) / GeV
    p12 = b.momentum ( 1 , 2 ) / GeV
    print 'J/psiK  momentum is is %s[GeV]'  % p12
```

Looping object (`b` here), as the name, suggests, make a loop over all `psi k` combinations. The loop is done in CPU efficient way, and no expensive vertex fitting is performed. One can estimate various *raw* (no fit) kinematical quantities using functions `momentum` , `mass` , etc... (Note that indices starts from `1` . For all LoKi-based functions the index `0` is reserved for *self-reference*, the mother particle itself). These `raw` quantities can be used for quick reject of *bad* combinations before making CPU-expensive vertex fit.
If/when combination satisfies certain criteria, the vertex
fit and creatino of the compound particle is triggered automatically if any of particle/vertex information is retrieved (either directy via `particle/vertex` method, or indirectly, e.g. via call to any *particle/vertex LoKi-functor*. The good created *mother* particles can be saved for subsequent steps under some unique tag:

```
for b in self.loop( 'psi k' , 'B+' ) :
    ## fast evaluation of mass (no fit here!)
    m12 = b.mass ( 1 , 2 ) / GeV
    if not 5 < m12 <   6    : continue
    chi2vx = VCHI2 ( b )   ## indirect call for vertex fitr and creation of B+ meson
    if not 0<= chi2v <  20  : continue
    m = M ( p ) / GeV
    if not 5 <  m < 5.6     :  continue
    m.save('MyB')
```

Obviously the looping can be combnied with filling of historgams and n-tuples.

How to deal with charge conjugation?

One can make two loops:

```
psis    = self.select ( 'psi' , 'J/psi(1S) -> mu+ mu-' )
kplus   = self.select ( 'k+'  , ( 'K+' == ID ) & ( PT > 1 * GeV ) &  ( PROBNNk > 0.2 ) )
kminus  = self.select ( 'k-'  , ( 'K-' == ID ) & ( PT > 1 * GeV ) &  ( PROBNNk > 0.2 ) )

bplus   = self.loop( 'psi k+', 'B+' ) ## the first  loop object
bminus  = self.loop( 'psi k-', 'B-' ) ## the second loop object
for cc in ( bplus , bminus ) :
   for b in cc :
      m12 = b.mass(1,2) / GeV
      ...
      b.save('MyB')
```

The popular alternative is *charge-blind* loop, that is a bit simpler, but it requires some accuracy:

```
psis    = self.select ( 'psi' , 'J/psi(1S) -> mu+ mu-' )
## ATTENTION: select both K+ and K-, note ABSPID here
k       = self.select ( 'k'   , ( 'K+' == ABSID ) & ( PT > 1 * GeV ) &  ( PROBNNk > 0.2 ) )

for b in self.loop ( 'psi k' , 'B+' )
      m12 = b.mass(1,2) / GeV
      ...
      psi = b(1) ## get the first daughetr
      k   = b(2) ## get the second daughter

      ## ATTENTION: redefine PID on-flight
      if Q( k ) > 0 : b.setPID( 'B+' )
      else          : b.setPID( 'B-' )

      b.save('MyB')
```

The saved particles can be extracted back using the method `selected` :

```
myB = self.selected('MyB')
for b in myB :
   print M(b)/GeV
```

Configuration

It is clear that to build `B+ -> J/psi(1S) K+` decays, one needs to
feed the algorithm with J/psi(1S)-mesond and kaons. Using `Selection` machinery is the most efficient ansd transaprent way to do it.

```
from PhysConf.Selections import AutomaticData
jpsi = AutomaticData( '/Event/Dimuon/Phys/FullDSTDiMuonJpsi2MuMuDetachedLine/Particles' )

from StandardParticles   import StdLooseKaons as kaons
bsel = BenderSelection ( 'MakeB' , [ jpsi , kaons ] )
```

The complete example of creation of `B+ -> J/psi(1S) K+` decays starting from `DIMUON.DST` is accessible from here

Keypoints

Wth these example, you should be able to do

- code Bender *algorithm* that perofem loopint, combianig anc creation of compound particles

Advanced Bender

Here we discuss some *advanced* features of Bender, namely

- the powefull *fill* of n-tuples provied by `BenderTools.Fill` module
- the treatment of `TisTos` information in Bender, provided by the module `BenderTools.TisTos`

Advanced fill of n-tuples

The n-tuple filling functionality, described above is drasticlaly extended using the functions from `BenderTools.Fill` module. The import of this module add following functions to the base class `Algo` :

Method	Short description
`treatPions`	add information about pions
`treatKaons`	add information about kaons
`treatProtons`	add information about protons
`treatMuons`	add information about muons
`treatPhotons`	add information about photons
`treatDiGammas`	add information about di-photons (pi0, eta,...)
`treatTracks`	add information about the tracks
`treatKine`	add detailed kinematic information for the particle
`fillMasses`	masses of sub-combinations
`addRecSummary`	add rec-summary information
`addGecInfo`	add some GEC-info

These methods can be considered as a kind of very light *tuple-tools*. All of them are (well) documented and one can easily inspect them:

```
import BenderTools.Fill
from Bender.Main import Algo
help(Algo.treatPions)
```

Also all these methods print detailed how-to infomratino in log-file at the moment of the first invoke, and it vasn be very helpful to understand the branches in n-tuple/tree.
The typical usage of these methods is:

```
tup = self.nTuple('MyTuple')
for p in particles :

    psi = p(1) ## the first daughter: J/psi

    ## fill few kinematic variables for the particles:
    self.treatKine   ( tup , p   , '_b'   )  ## use the suffix to mark variables
    self.treatKine   ( tup , psi , '_psi' )  ## use the suffix to mark variables

    self.treatKaons  ( tup , p ) ## fill some basic information for all kaons
    self.treatMuons  ( tup , p ) ## fill some basic information for all muons
    self.treatTracks ( tup , p ) ## fill some basic information for all charged tracks

    tup.write()
```

Challenge

1. Add (some of) these functions into your previous Bender module with n-tuples.
2. Run it and observe the detailed printout in log-file
3. Observe new variables in your n-tuple/tree and find their description in the log-file or via `help(Algo.<THEMETHOD>)`

- Is the description for all new varibales clear enough?

Solution

The complete module is accessible [here] (https://gist.github.com/VanyaBelyaev/becc26fe5dea90aa96cb8f929faf6a53)

Handling of `TisTos` information in Bender (`BenderTools.TisTos` module)

Bender offers set of methods to handle `TisTos` -information in (relatively) easy way. This functionality comes from `BenderTools.TisTos` module. In short, it adds three relates method in the base class `Algo` :

Method	Short description
`decisions`	collect the trigger decisions for given particle
`trgDecs`	print the collected trigger statistics in readable way
`tisTos`	fill N-tuple with TisTos information

All of them are (realtively well) documented and one can easily inspect them:

```
import BenderTools.TisTos
from Bender.Main import Algo
help(Algo.decisions)
```

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.

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