# Introduction to ROOT RDataFrame (Python)

Anubhav Gupta, 21 March 2024

# Introduction

- ROOT's high-level analysis interface available in **ROOT v6.14+**
- Analysis is defined as a **sequence of operations** to be performed on the **data frame** object
- Much faster than TTree::Draw(), TTREE::GetEntry() or TTree::READER()
  - **Multithreading**
  - Parallel actions per event loop
  - Optimised filtering and I/O
- Provides various methods to perform **most common operations** required by **ROOT analysis**


Documentation

# Simple Analysis: Step 1, Build Dataframe Object

❑ Build a **dataframe object**

```
import ROOT
treeName = 'Mytree'
file = 'analysis_ntuple.root'
# create the dataframe object
df = ROOT.RDataFrame(treeName, file)
# df.Display(). Print() # not lazy,trigger the event loop
```

| Row | particle_charge | particle_eta | particle_mass | particle_px | particle_py |
|---|---|---|---|---|---|
| 0 | 1 | 2.162787 | 0.000511 | 80.168912 | 45.325287 |
| 1 | 1 | -2.038307 | 0.105000 | 80.057569 | 34.165339 |
| 2 | 1 | 0.194084 | 0.000000 | -20.263972 | 72.008009 |
| 3 | -1 | -1.477739 | 0.105000 | 61.113170 | 21.052559 |
| 4 | 1 | 0.852338 | 0.000000 | 18.356744 | 22.275733 |

**Lazy operation**: does not trigger the event loop

# Simple Analysis: Step 2, Transformations

- Apply series of transformation
    - **Define** new columns

```
# Define a new column, lazy!
df = df.Define("particle_pT", "sqrt(particle_px*particle_px+particle_py*particle_py)")
```

`Define` takes the name of the new column and its expression

# Simple Analysis: Step 2, Transformations

- ❑ Apply series of transformation
  - ❑ **Define** new columns
  - ❑ **Filter** events: apply cuts/selections

```
# Filter the events with pT > 50 GeV and |eta| < 0.8, lazy!
df = df.Filter("particle_pT > 50.", "pT cut")
df = df.Filter("fabs(particle_eta) < 0.8", "eta cut")
```

`Filter` takes the expression and name of the cut

# Simple Analysis: Step 3, Actions

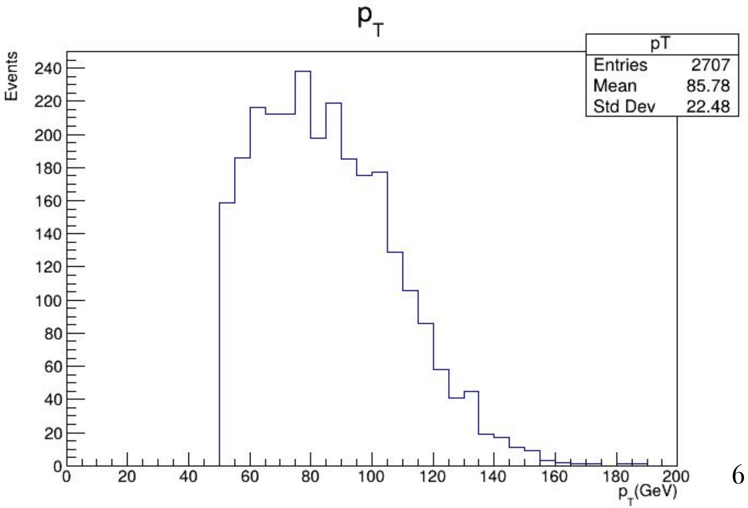
- Apply **actions** to the transformed data to produce results (histograms)

```
# Book histograms, lazy
h_pT = df.Histo1D(("pT",
"p_{T};p_{T}(GeV);Events",40,0,200),"particle_pT")

# Plot Histogram
canvas = ROOT.TCanvas()
h_pT.Draw()  # trigger the event loop
canvas.Draw()
canvas.SaveAs("pT.png")
```

specify a model histogram with

- a name, a title, xtitle,ytitle
- a predefined axis range



# Compactly

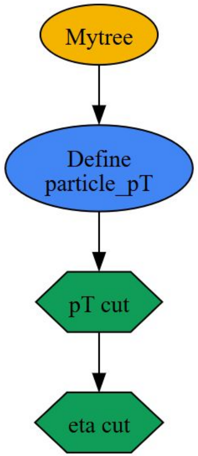
```
import ROOT
df = ROOT.RDataFrame('Mytree', 'analysis_ntuple.root')

h= df.Define("particle_pT", "sqrt(particle_px*particle_px+particle_py*particle_py)") \
   .Filter("particle_pT > 50.", "pT cut").Filter("fabs(particle_eta) < 0.8", "eta cut") \
   .Histo1D(("pT", "p_{T};p_{T}(GeV);Events",40,0,200),"particle_pT")

c = ROOT.TCanvas()
h.Draw()
c.Draw()
```

# Cutflow Reports

ROOT.RDF.SaveGraph(df,"DAG.dot")
from graphviz import Source
Source.from_file("DAG.dot")




```
report = df.Report()
report.Print()
```

```
pT cut     : pass=8331        all=10000       -- eff=83.31 % cumulative eff=83.31 %
eta cut    : pass=2707        all=8331        -- eff=32.49 % cumulative eff=27.07 %
```

Not a lazy action triggers the event loop!
Call it before using other non-lazy action like h.Draw()

# Saving Data To a File

```
# Save the data frame with new columns into root file
df.Snapshot("tree", "testoutput.root")
```

Non-lazy action: triggers the event loop!

# Working With collections

- RDataFrame reads collections as the special type [ROOT::RVec]

  E.g., a branch containing an array of floating point numbers can be read as a `ROOT::RVec<float>`

- C-arrays, `Std::vectors,` and many other collection types can be read this way
- `RVec` is a container similar to `std::vector` (and can be used just like a `std::vector`)
- `RVec` offers a rich interface to operate on the array elements in a vectorized fashion, similarly to Python's NumPy arrays.

# Example With Collection

```python
import ROOT
df = ROOT.RDataFrame('myDataset','collections_dataset.root')
df = df.Define("good_pt", "sqrt(px*px + py*py)[E>100]")
```

- px, py and E are the columns; the elements of those columns are RVecs
- Operations on RVecs, such as sum, product, sqrt, preserve the dimensionality of the array
- [E>100] selects the elements of the array that satisfy the condition
- E > 100: boolean expressions on RVecs such as E > 100 return a mask, that is, an array with information whose values pass the selection (e.g., [0, 1, 0, 0] if only the second element satisfies the condition)

| Row | E | nPart | px | py |
|---|---|---|---|---|
| 0 | 130000.009398 | 40 | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | ... | | ... | ... |
| 1 | 130000.009398 | 53 | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | ... | | ... | ... |
| 2 | 130000.009398 | 185 | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | 0.939571 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | 0.938280 | | 0.000000 | 0.000000 |
| | ... | | ... | ... |

# Using C++ function in python

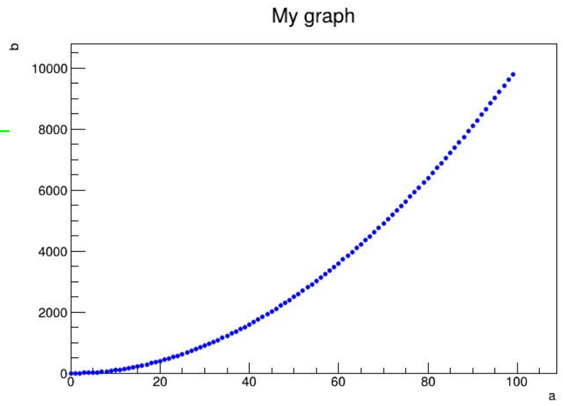
```
ROOT.gInterpreter.Declare(
"""
float square(float x)
{
return x*x;
}
"""
)
```

```
%%cpp
float asfloat(unsigned long int
entrynumber)
{
    return entrynumber;
}
```

Jupyter Notebook

```
df = ROOT.RDataFrame(100) # create dataframe with 100
entries
df= df.Define("a", "asfloat(rdfentry_)")
df = df.Define("b", "square(a)")

c = ROOT.TCanvas()
graph = df.Graph("a","b")
graph.SetMarkerStyle(20)
graph.SetMarkerSize(0.5)
graph.SetMarkerColor(ROOT.kBlue)
graph.SetTitle("My graph")
graph.Draw("AP")
c.Draw()
```



# Using Python functions in RDataFrame

```
@ROOT.Numba.Declare(['float'],'float')
def square(x):
    return x*x


@ROOT.Numba.Declare(['unsigned long'], 'float')
def asfloat(entry):
    return entry*1.0
```

```
df = ROOT.RDataFrame(100) # create dataframe with 100
entries
df= df.Define("a", "Numba::asfloat(rdfentry_)")
df = df.Define("b", "Numba::square(a)")
c = ROOT.TCanvas()
graph = df.Graph("a","b")
graph.SetMarkerStyle(20)
graph.SetMarkerSize(0.5)
graph.SetMarkerColor(ROOT.kBlue)
graph.SetTitle("My graph")
graph.Draw("AP")
c.Draw()
```

# Multithreading

```
import ROOT
ROOT.EnableImplicitMT(3)
```

Events run in parallel on multiple cpu cores

- Make sure the user defined functions are thread safe

# Summary

- Covered:
  - RDataFrame Basics
  - Working with Collections in RDataFrame
  - Using C++ functions in python and RDataFrame
  - Using python functions in RDataFrame Define using Numba
  - Enable multi-threading. Warning:  Make sure of thread safety!

# References And Links

- Web page: https://root.cern
- Documentation: https://root.cern/doc/master/
- Primer: https://root.cern/primer/
- Forum: https://root-forum.cern.ch
- Sources for this presentation and tutorials
  - https://github.com/root-project/student-course
  - https://github.com/root-project/summer-student-course/tree/main

# Code available

- ❑ /ceph/e4/users/agupta/public/programmingcours_Rdataframe
- ❑ https://cernbox.cern.ch/s/Ar4s8KEHSNKUr6s