Istituto Nazionale di Fisica Nucleare

ROOT Lecture

HASCO Summer School 2019

Antonio Sidoti
Antonio.sidoti@bo.infn.it

Istituto Nazionale Fisica Nucleare – Sezione di Bologna

Question

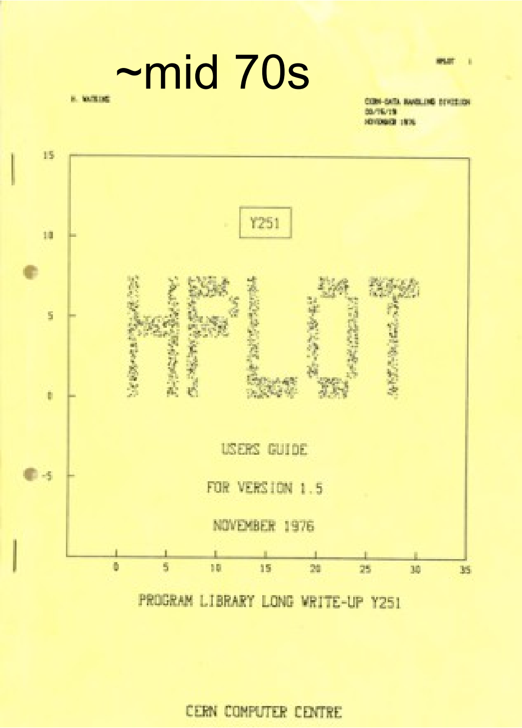
Go to **www.menti.com** and use the code **35 69 0**

Introduction

- Analysis and visualization are fundamental for particle physics (experimental and not only)
- ROOT is the primary tool for data analysis in high energy physics (not only collider physics)
- May be not the reference tool if you are doing detector R&D, theory/phenomenology, machine learning → Useful to have an idea on how to use it since at some point you will use it
- In different contexts from HEP (e.g. astrophysics) not the tool used preferentially → many physicists with particle physics background are going to astroparticle → ROOT is being uses in that context too!

History

~mid 70s



Y251

HPLOT

USERS GUIDE

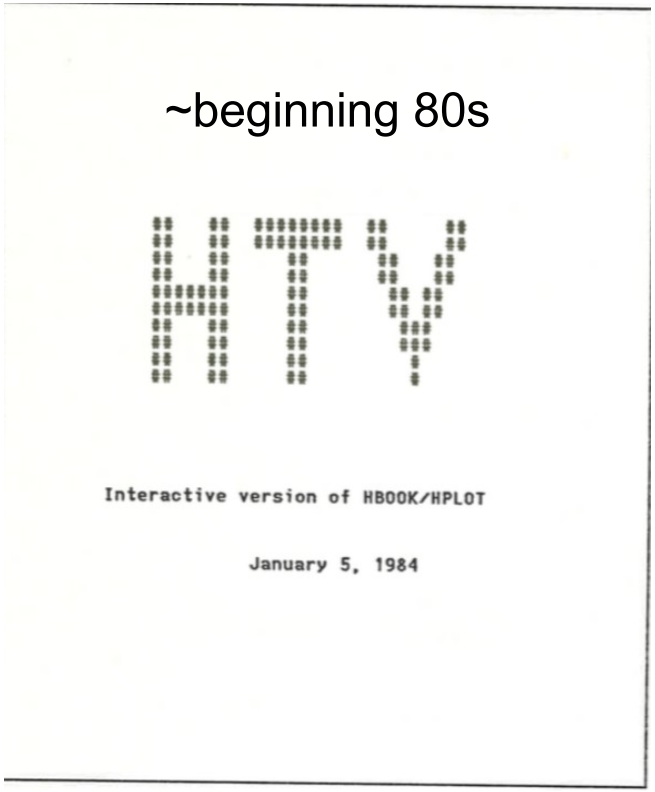
FOR VERSION 1.5

NOVEMBER 1976

PROGRAM LIBRARY LONG WRITE-UP Y251

CERN COMPUTER CENTRE

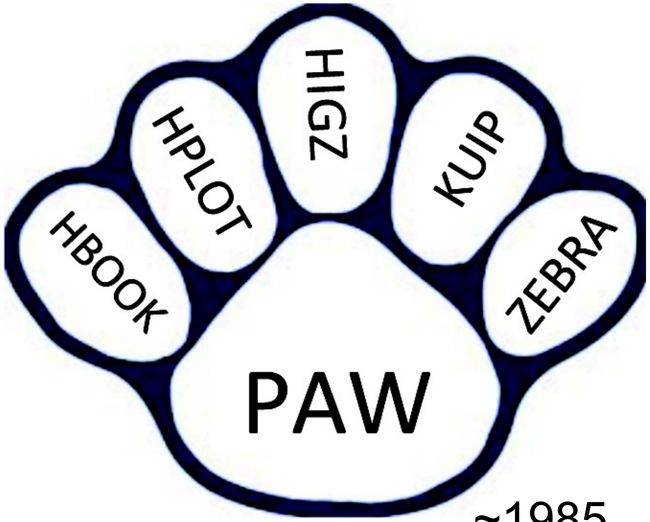
~beginning 80s



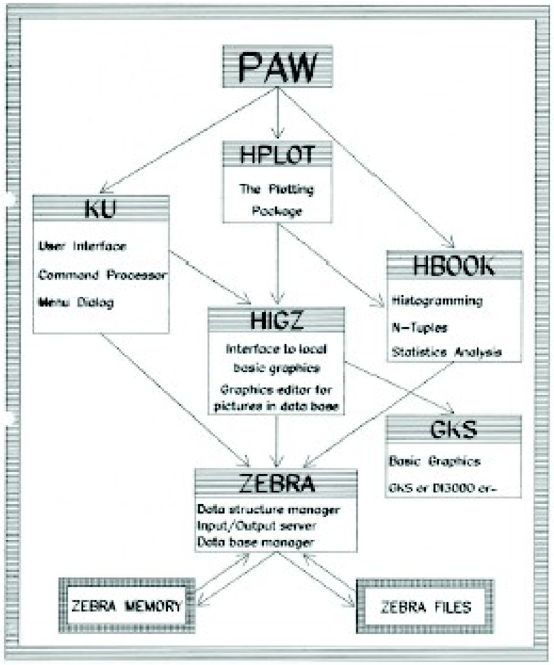
HTY

Interactive version of HBOOK/HPLOT

January 5, 1984



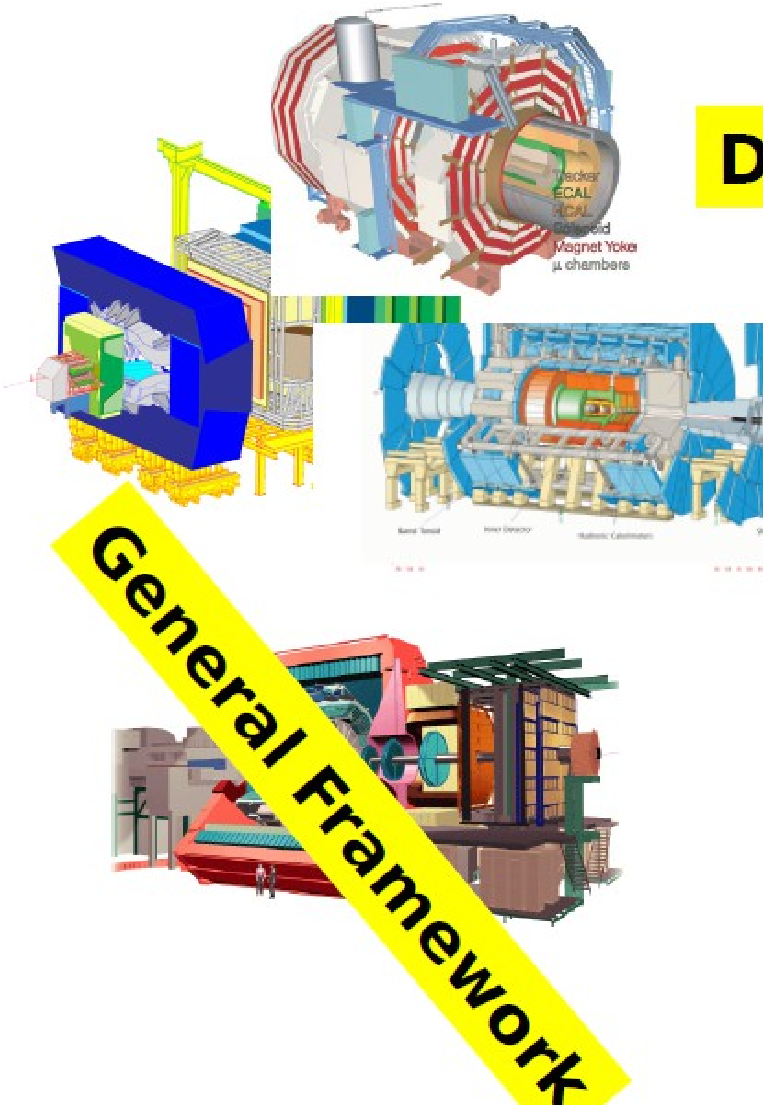
~1985



For a nice presentation on ROOT history and development look here https://indico.cern.ch/event/667648/ (also recording)

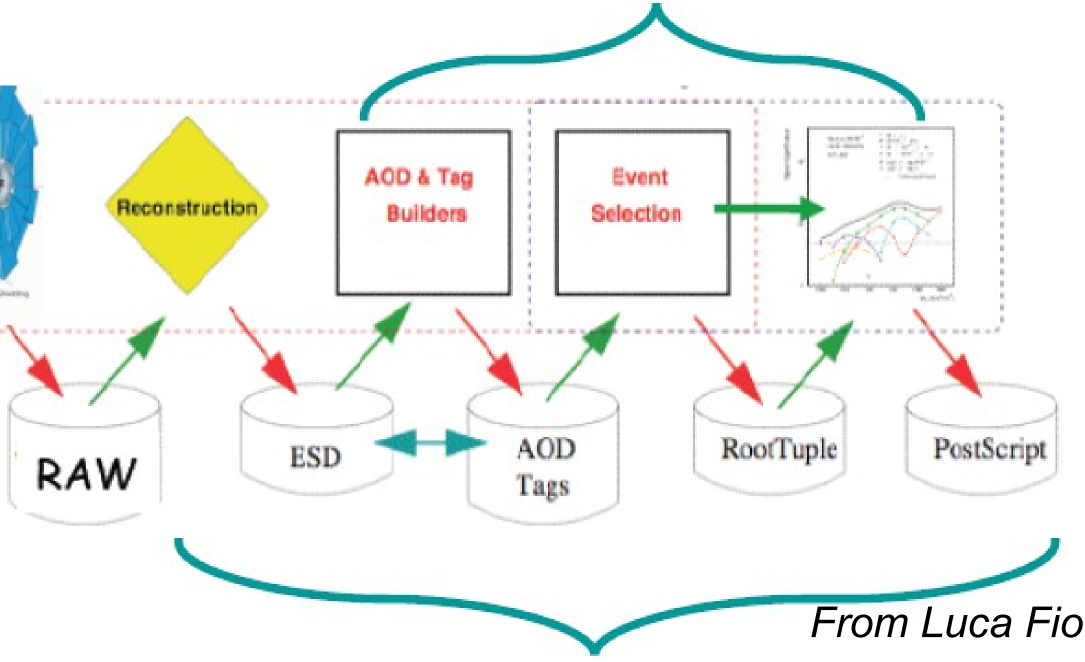
ROOT Application Domains

In modern HEP experiments



General Framework

Data Analysis & Visualization



From Luca Fiorini slides

Data Storage: Local, Network

What is ROOT?

- ROOT is a powerful scientific software framework which is...
- Likely older than many of you (1994, a quarter of a century years old) → Current version is 6.18/00 (frequenet updates)
- Developed by CERN (mostly the EP-SFT group)
- Written in C++, but with interfaces to other languages (python)
- Popular enough to have its own wikipedia page (11 languages)
- Widely used in particle physics, but also used externally (finance, astroparticles)
- Also a data format tailored to particle physics needs (large I/O)

What do you do in ROOT?

- Software framework for data processing, storage, analysis and visualization

→ **Translation:**

In modern HEP experiments:

- Data (real or simulated) are saved in ROOT format (writing ntuples)
- Data re processed (reconstructed, calibrated) in ROOT format (reading/writing ntuples)
- Data are analyzed using ROOT (reading/writing histograms)
- Understand what you have done (plotting histograms)
- Interpreted using ROOT (histogram fitting)

How to start ROOT

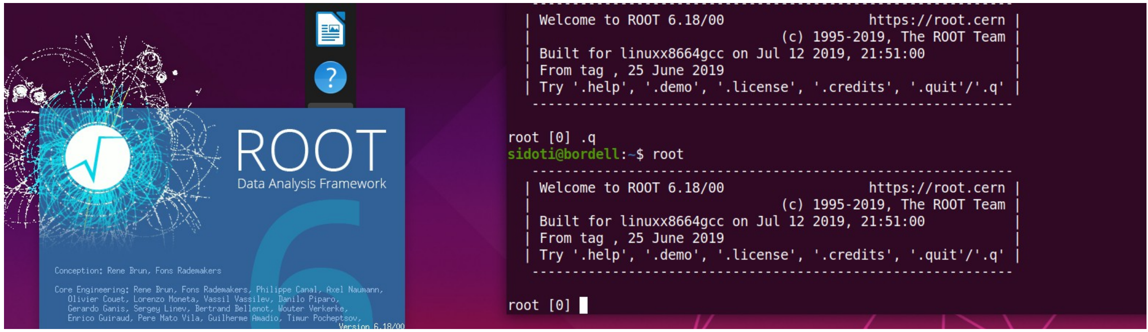
- After installing/setup root on your system (cf here)

$> root

(but this takes a loooong time)

$> root -l (much quicker!)

root [0]



Conception: Rene Brun, Fons Rademakers

Core Engineering: Rene Brun, Fons Rademakers, Philippe Canal, Axel Naumann, Olivier Couet, Lorenzo Moneta, Vassil Vassilev, Danilo Piparo, Gerardo Ganis, Sergey Linev, Bertrand Bellenot, Wouter Verkerke, Enrico Guiraud, Pere Mato Vila, Guilherme Amadio, Timur Pocheptsov,

Version 6.18/00

```
   ------------------------------------------------------------------
  | Welcome to ROOT 6.18/00                        https://root.cern |
  |                               (c) 1995-2019, The ROOT Team |
  | Built for linuxx8664gcc on Jul 12 2019, 21:51:00              |
  | From tag , 25 June 2019                                        |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q' |
   ------------------------------------------------------------------

root [0] .q
sidoti@bordell:~$ root
   ------------------------------------------------------------------
  | Welcome to ROOT 6.18/00                        https://root.cern |
  |                               (c) 1995-2019, The ROOT Team |
  | Built for linuxx8664gcc on Jul 12 2019, 21:51:00              |
  | From tag , 25 June 2019                                        |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q' |
   ------------------------------------------------------------------

root [0] 
```

$> root -b (disable graphics when plotting → “b” is for batch)

$> root -n (does NOT execute rootlogon.C and rootlogoff.C)

$> root -e ‘myCommand’ → executes myCommand

$> root file1.root → loads file1.root

$> root file1.C [file2.C … fileN.C] → executes macro[s]

$>root -h → help

Some starting options can be combined (e.g -q -b, ….)

Ways to use ROOT

How to interface to ROOT?

1) GUI (Graphical User Interface)
2) Command line → quick checks and studies

   CINT (almost C++)

   Python prompt (Python)
3) ROOT macros: simple or moderate programs, in C++
4) PyROOT scripts: simple or moderate programs, in python
5) Compiled ROOT: complex or CPU-intensive programs, in C++

Question

Go to **www.menti.com** and use the code **35 69 0**

How to interface to ROOT?

1) GUI (Graphical User Interface)

2) Command line → quick checks and studies

CINT (almost C++)

Python prompt (Python)

3) ROOT macros: simple or moderate programs, in C++

4) PyROOT scripts: simple or moderate programs, in python

5) Compiled ROOT: complex or CPU-intensive programs, in C++

The C interpreter (CINT/CLING)

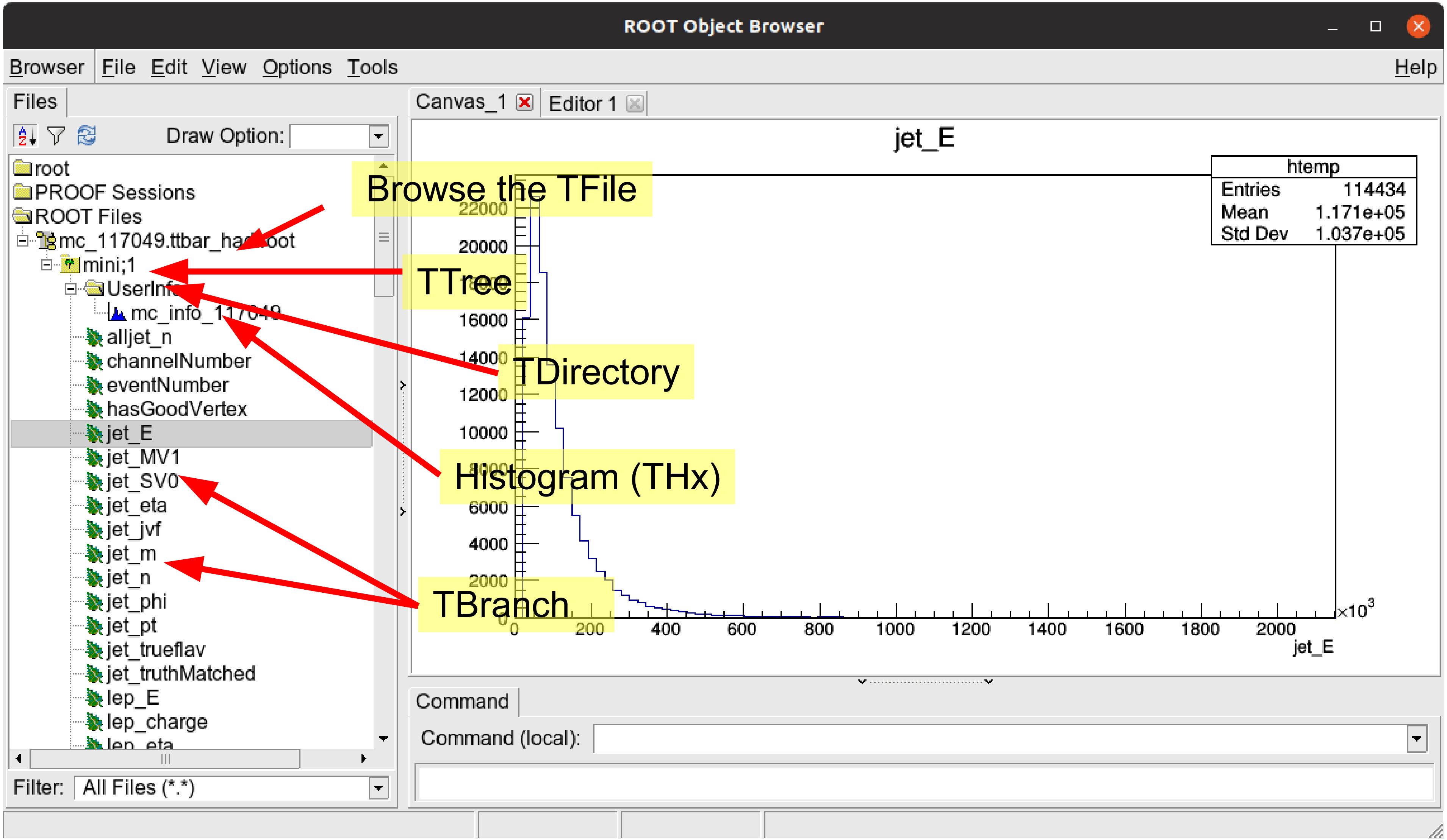
CINT/CLING commands start with "." :
root[0] .q → exit (in case you were wondering how to exit…)
root[0] .qqqq (→ force quit a` la $> kill -9)
root[0] .L myMacro.cpp (load but don't execute myMacro.cpp)
root[0] .x myMacro.cpp (load and execute)
root[0] .> file1.log (redirects output to log file)
root[0] .help → (other options)
Note: CINT/CLING is not "really C++". e.g. pointers and values are treated in the same way (try to do this in C++!)

Shell command starts with ".!" (note, space between prefix and command is irrelevant)
root[0] .!ls -la (list files in directory)
root[0] .! cd <some directory>

Other commands are "almost" C++ commands (tab completion)
root[0] TBrowser *b = new TBrowser() (end of line ";" is optional)

GUI (TBrowser)

root[0] new TBrowser

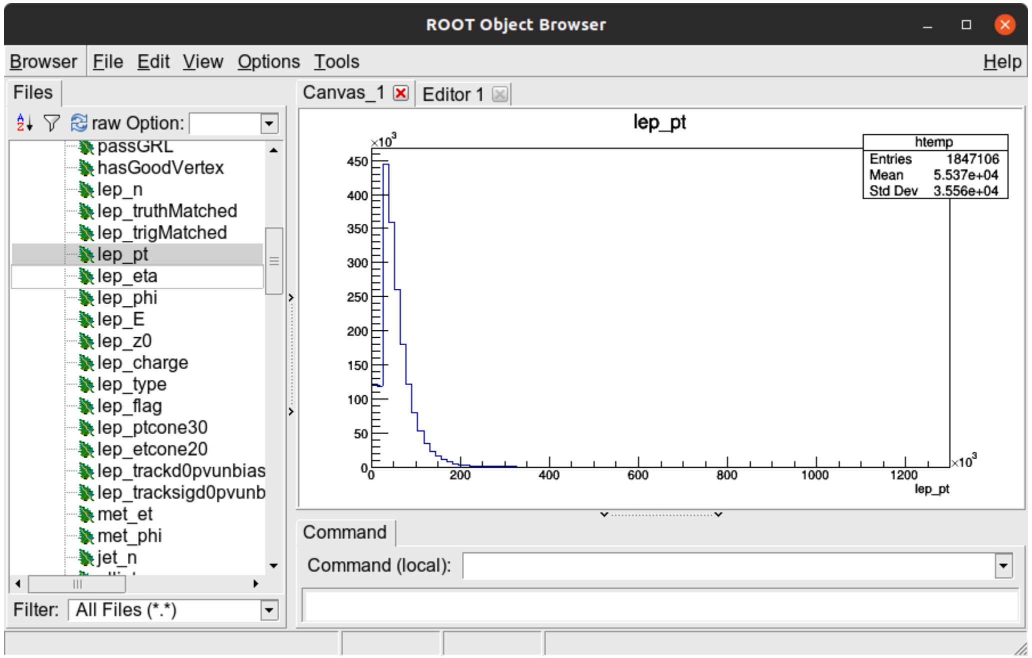


What you can do with GUI?

Inspect TFile contents

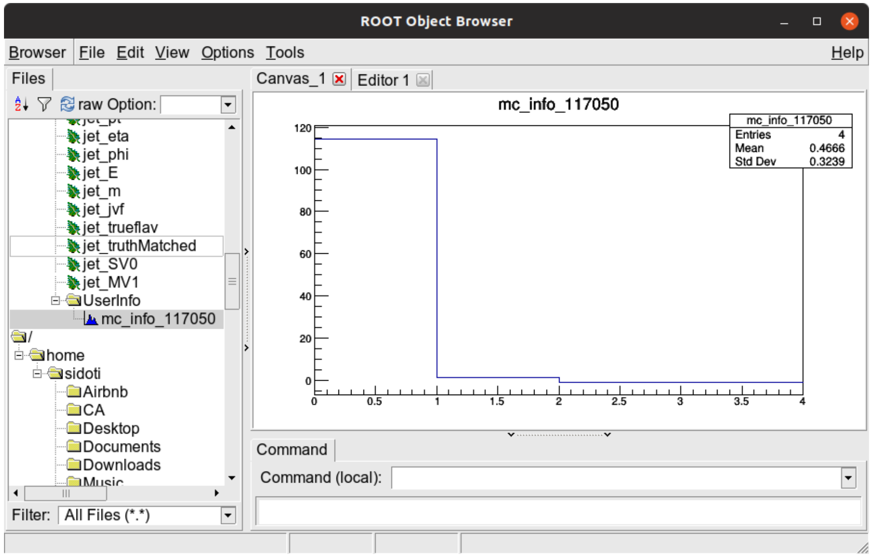
Draw TBranch values

Draw histograms



htemp

Entries	1847106
Mean	5.537e+04
Std Dev	3.556e+04



mc_info_117050

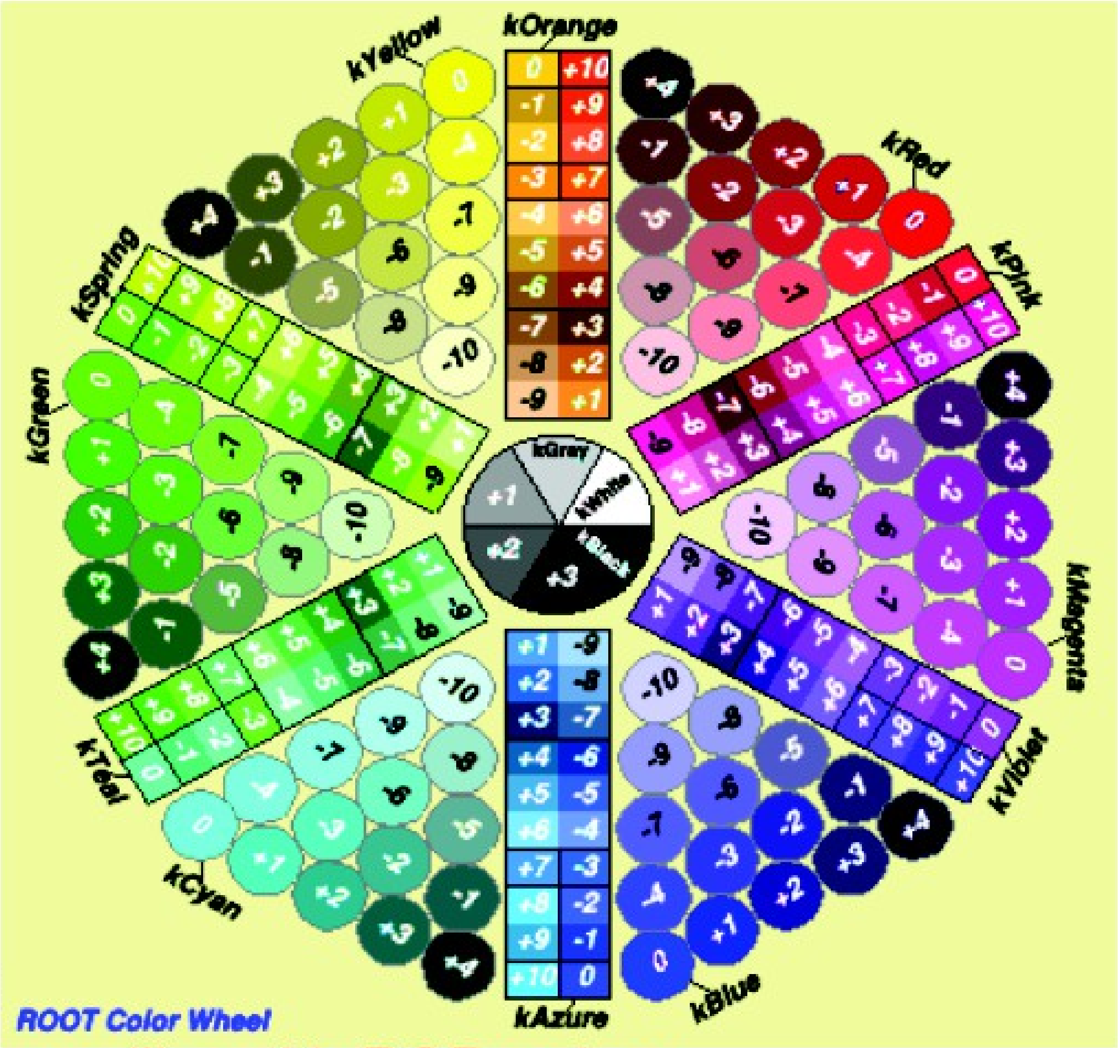
Entries	4
Mean	0.4666
Std Dev	0.3239

Drawing branches and display histograms are **NOT EQUIVALENT!**

You can perform graphical operations (zoom axis, log axis, change color line/style/width, etc...) 2

Sadly the Americans won the fight over spelling rights

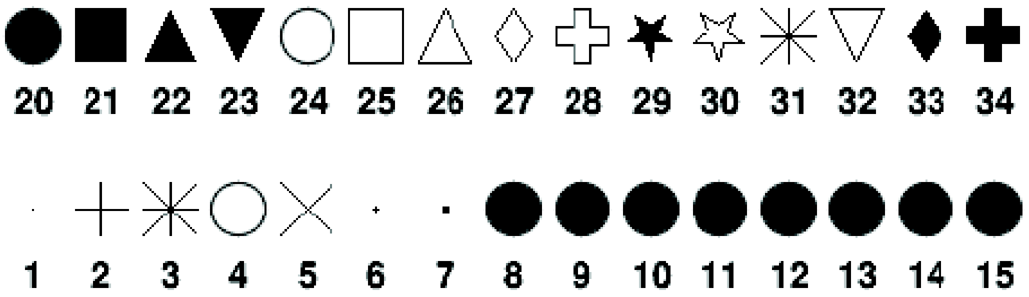
- TColorWheel: **216 colors** as used in web applications
- Special identifiers for colors
  - kWhite, kBlack, kGrey
  - kRed, kBlue, kGreen
  - KYellow, kMagenta, kCyan
  - And more…
- Colors in between
  - Obtained by ± [1,10]
- Colorize objects
  - SetLineColor()
  - SetFillColor()
- **Use colors smartly** (grayscale)

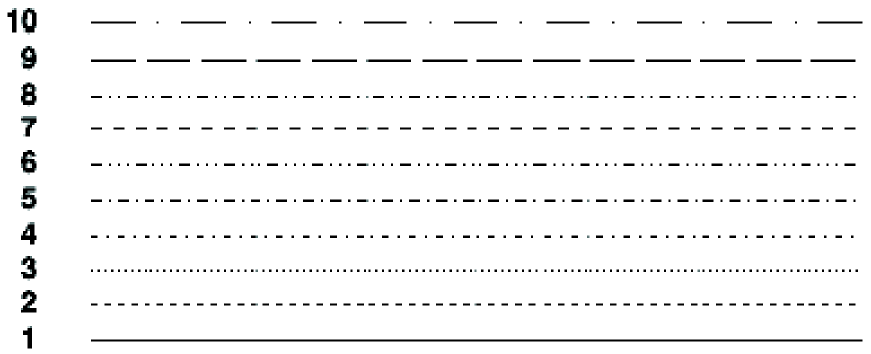


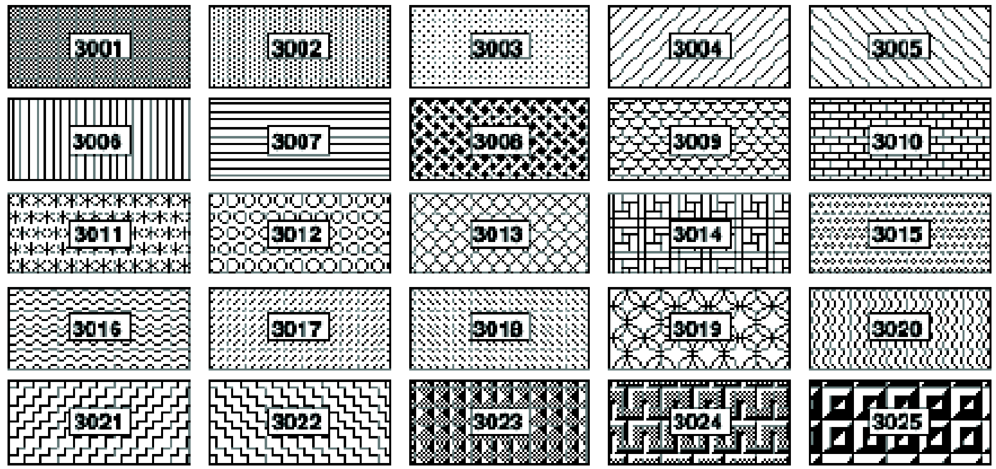
You can also define any colour you like with RGB or hex

From W. Kalderon (HASCO 2017) that was inspired from O. Nackenhorst (2016) slides

- Marker style
  - KDot, kPlus, kStar, kCircle, etc.
  - **SetMarkerStyle()**
- Line style
  - Fixed: 1-10
  - Customizable: SetLineStyleString()
  - **SetLineStyle()**
- Fill area style
  - Fixed: 3000 + 1-25
  - Customizable: FillStyle = 3ijk
    - i=space between each hatch
    - j=angle(<90), k=angle(>90)
  - **SetFillStyle()**







From W. Kalderon (HASCO 2017) that was inspired from O. Nackenhorst (2016) slides

TBranches

- This is where the variables you are interested in are stored.
- TBranches can be primitive types
- ROOT supports primitive types are the types that are native to a language
- You do not need to include any libraries to use them
- They generally have a fixed interpretation
- In C++, the primitive types are very simple:

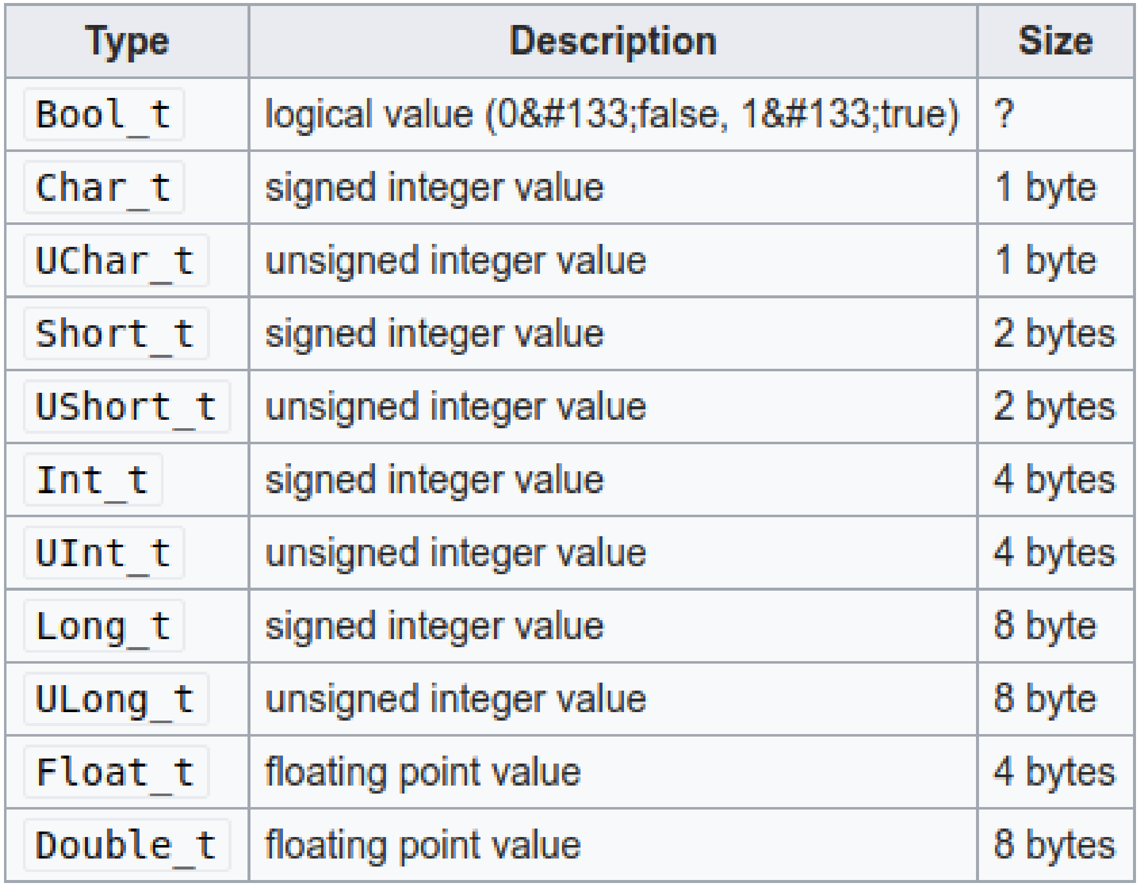
  Integer numbers: (unsigned) char, short, int, long, long long

  Real numbers: float, double, long double

  Conditions: bool = true" or false" = 1 or 0

Primitive Types

To ensure that a type is machine independent (in some machines int are 16 bit whyle in moder ones thay are 32 bits) → Redefinition of primitive types



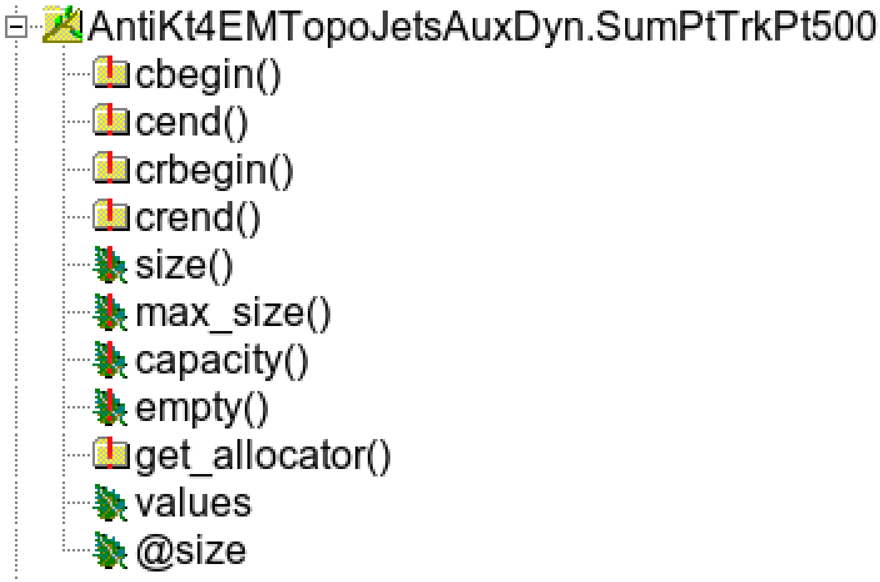
Type	Description	Size
`Bool_t`	logical value (0…false, 1…true)	?
`Char_t`	signed integer value	1 byte
`UChar_t`	unsigned integer value	1 byte
`Short_t`	signed integer value	2 bytes
`UShort_t`	unsigned integer value	2 bytes
`Int_t`	signed integer value	4 bytes
`UInt_t`	unsigned integer value	4 bytes
`Long_t`	signed integer value	8 byte
`ULong_t`	unsigned integer value	8 byte
`Float_t`	floating point value	4 bytes
`Double_t`	floating point value	8 bytes

Note: you have char → not strings!

Complex types

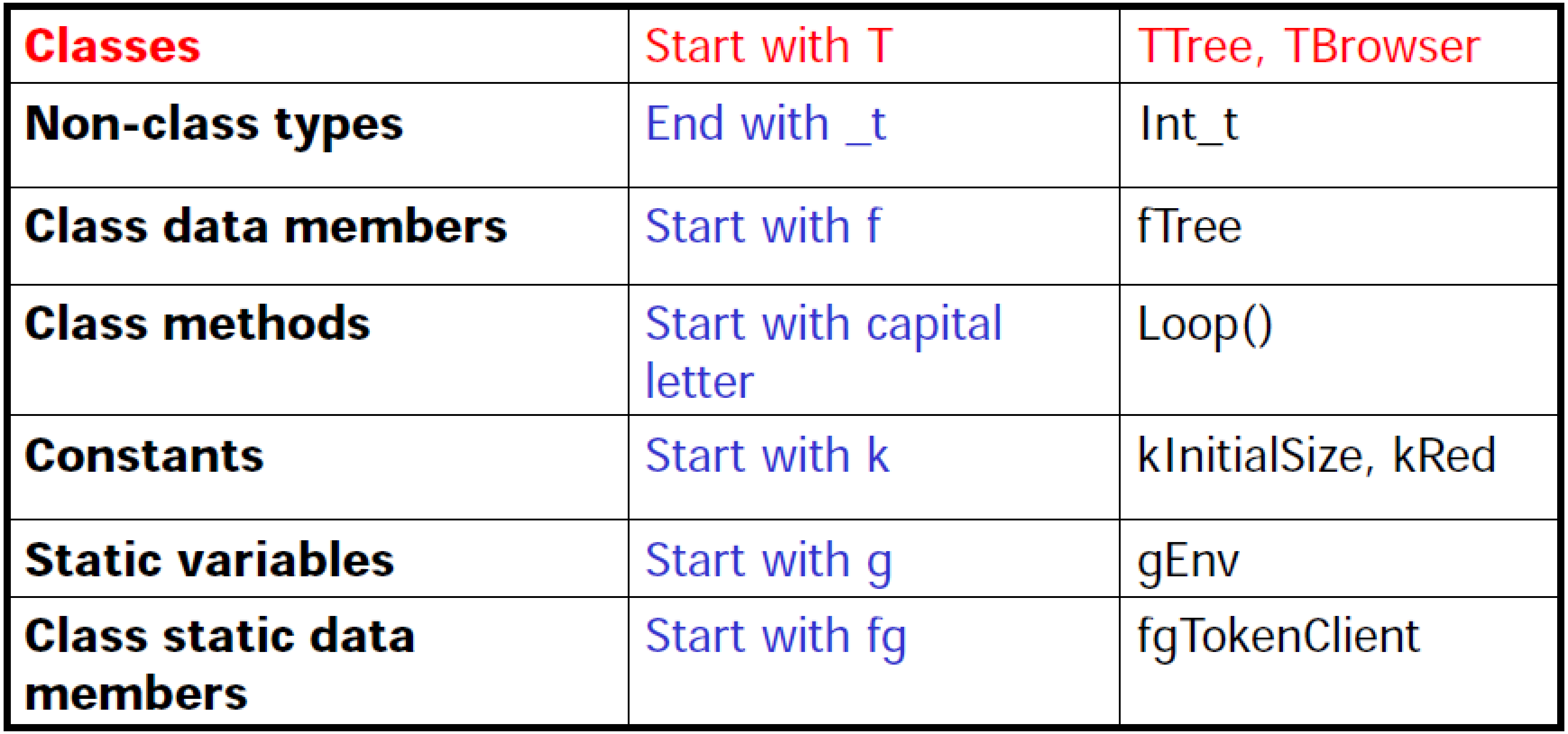
- You can store vectors (a` la C++) of primary types or custom classes
- Even strings are complex types→ you can use std::string or TString (root-specific native)
- And many other derived types you can define

Complex types look like this in TBrowser
How to access them? → see later
In that case this is a std::vector



AntiKt4EMTopoJetsAuxDyn.SumPtTrkPt500
cbegin()
cend()
crbegin()
crend()
size()
max_size()
capacity()
empty()
get_allocator()
values
@size

Coding Conventions



Classes	Start with T	TTree, TBrowser
Non-class types	End with _t	Int_t
Class data members	Start with f	fTree
Class methods	Start with capital letter	Loop()
Constants	Start with k	kInitialSize, kRed
Static variables	Start with g	gEnv
Class static data members	Start with fg	fgTokenClient

How to interface to ROOT?

1) GUI (Graphical User Interface)

2) Command line → quick checks and studies

CINT/CLING (almost C++)

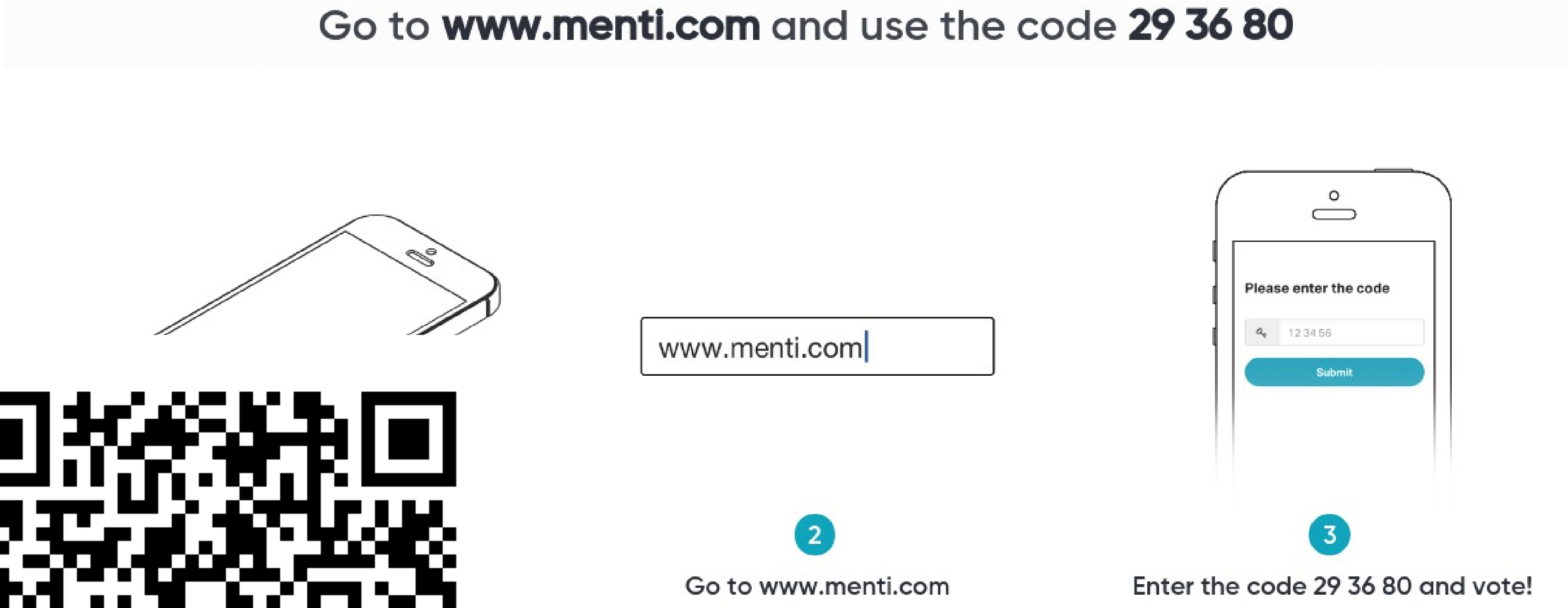
Python prompt (Python)

3) ROOT macros: simple or moderate programs, in C++

4) PyROOT scripts: simple or moderate programs, in python

5) Compiled ROOT: complex or CPU-intensive programs, in C++

Write you CERN user name (if you have one)



Go to **www.menti.com** and use the code **29 36 80**

Go to www.menti.com

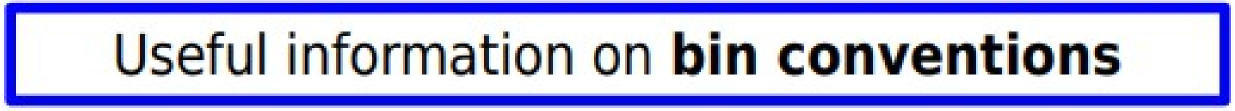
Enter the code 29 36 80 and vote!

I'll share with you my jupyter notebooks

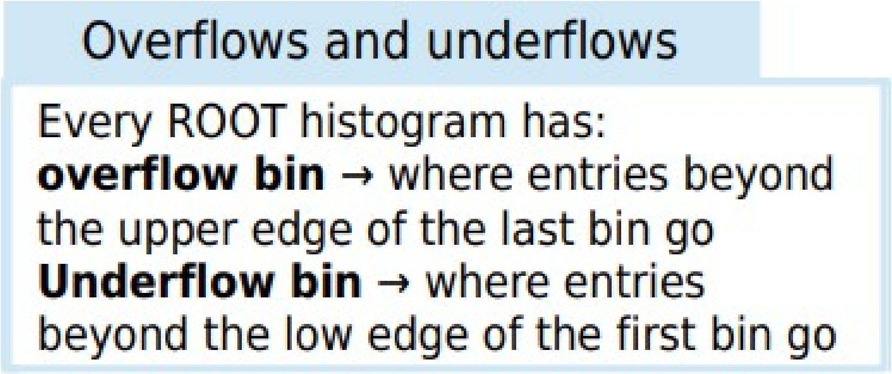
Examples: CINT/CLING (C++)

CINT/CLING notebook

Histogram binning



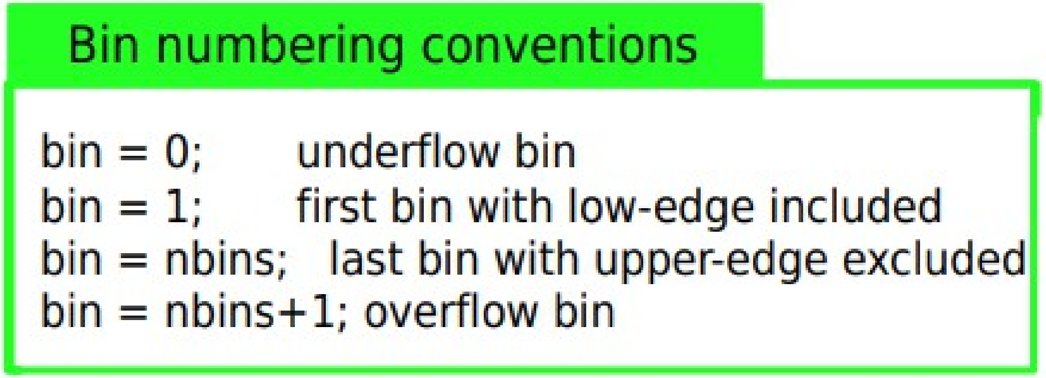
Useful information on **bin conventions**



Overflows and underflows

Every ROOT histogram has:
overflow bin → where entries beyond the upper edge of the last bin go
Underflow bin → where entries beyond the low edge of the first bin go

Beware when you bin integers!



Bin numbering conventions

bin = 0; underflow bin
bin = 1; first bin with low-edge included
bin = nbins; last bin with upper-edge excluded
bin = nbins+1; overflow bin

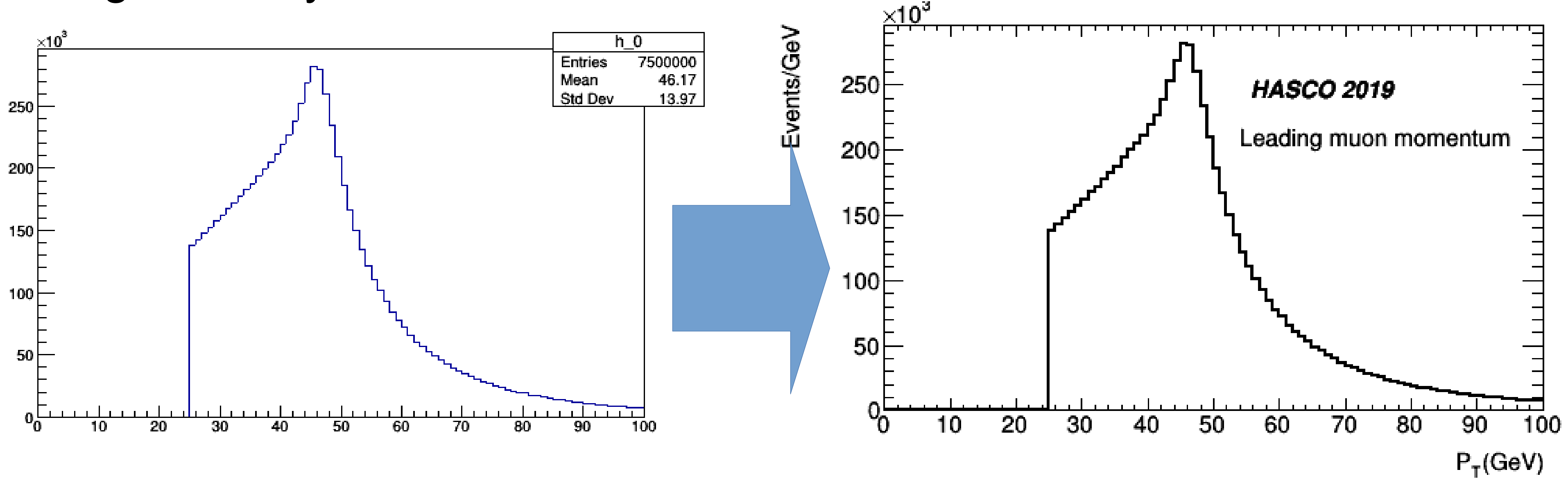
From C. Doglioni and A. Andreazza (2012) slides

More on Style

Default histogram style gives, in general, awful and unreadable histograms. We have seen how to modify graphical attributes of histograms etc, but we find long to do that for all the histograms

rootlogon.C in the directory where you launch ROOT is executed when you start ROOT → use it to set default.

For tomorrow hands download the hascostyle.tgz file and untar it on your working directory



To add labels etc..

```
root>.L HascoLabels.C
root> AtlasLabels(0.6,0.7,”My label”)
```

How to interface to ROOT?

1) GUI (Graphical User Interface)
2) Command line → quick checks and studies
   CINT/CLING (almost C++)
   Python prompt (Python)
3) ROOT macros: simple or moderate programs, in C++
4) PyROOT scripts: simple or moderate programs, in python
5) Compiled ROOT: complex or CPU-intensive programs, in C++

Python in 10 (ehm) minutes

From xkcd

High level interpreted programming language

Object-oriented tool

Some people write entire analyses using pyROOT and derivatives...can be done!

Threats string in a more straightforward way

Useful properties:

Everything is a reference (no pointers...)

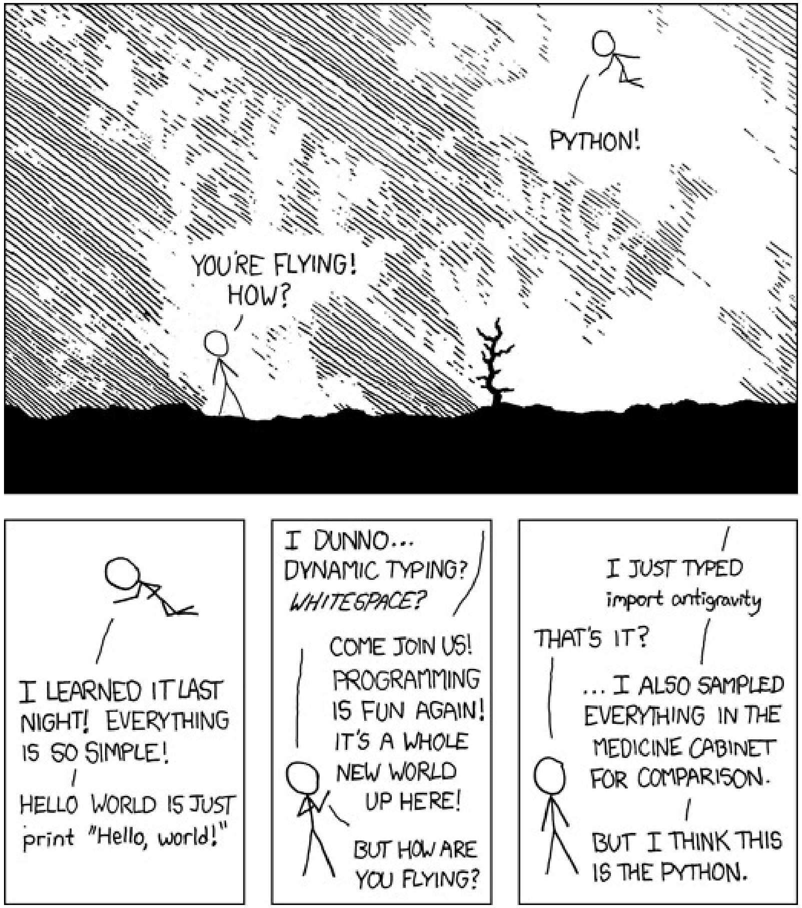
Automatic garbage collection (this sometimes clashes with ROOT's...)

Built-in help and reference listing

Strongly typed

What I don’t like:

Indentation! (you don’t have {} for delimiting blocks of instructions)

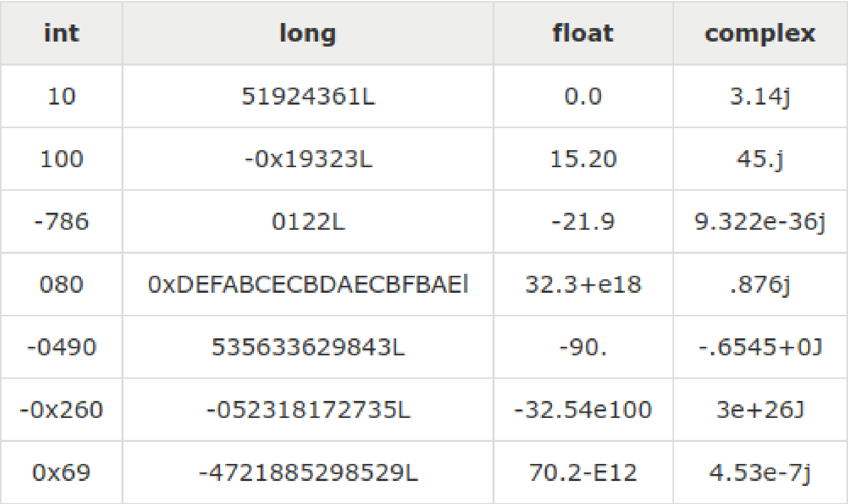


Comments start with #

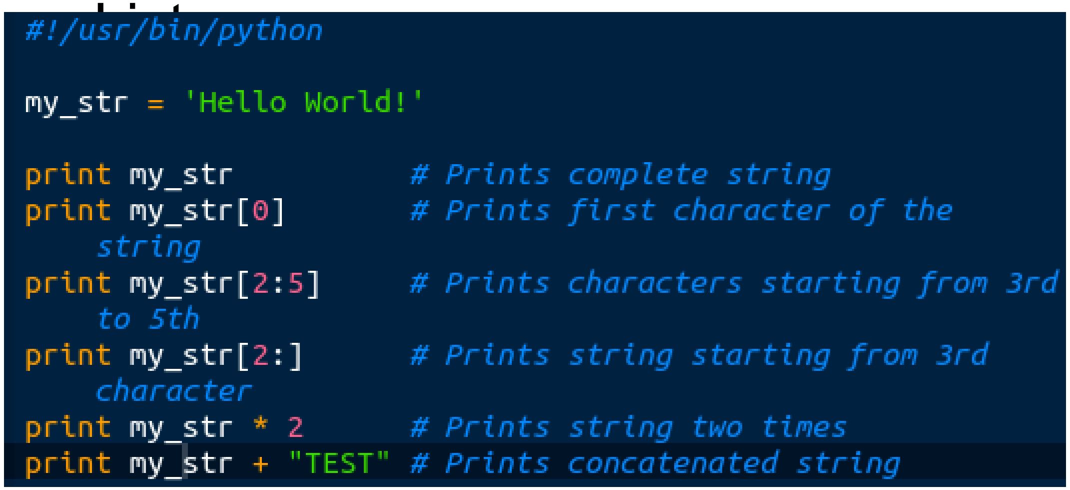
Caution: Two python versions exist: Python 2.x and Python 3
Many of the exercises rely in Python 2.x (2.7) that will NOT be supported after 2020

Standard Data Types

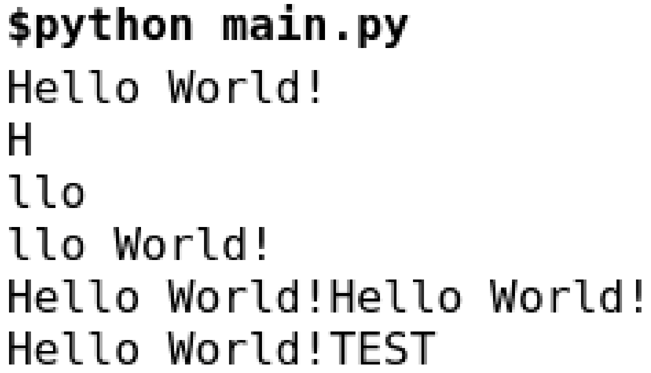
- Numbers
- Int (signed integers)
    - Long (long integers, also octal and hex)
    - Float
    - Complex
- Strings



int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

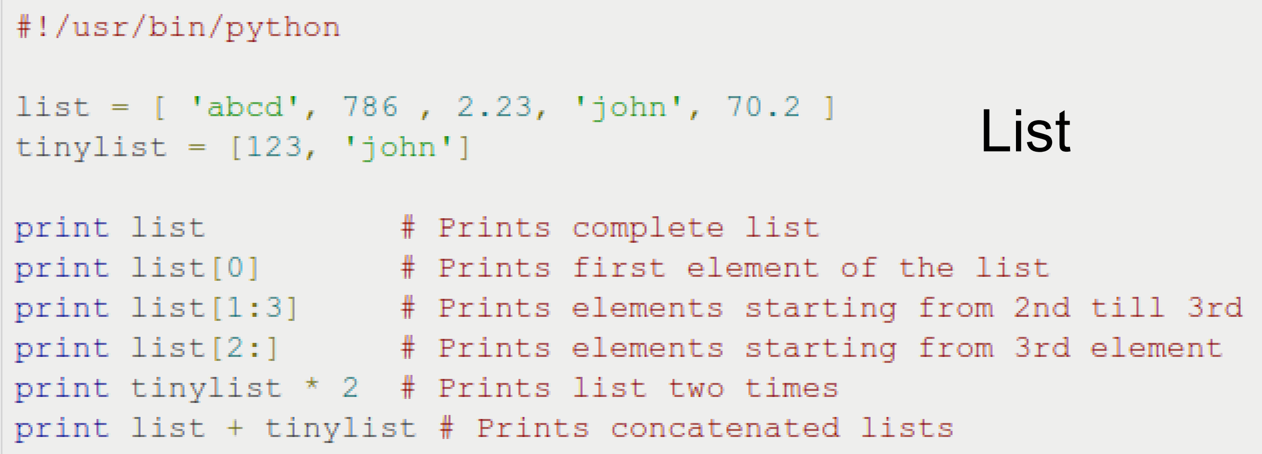


```
#!/usr/bin/python

my_str = 'Hello World!'

print my_str          # Prints complete string
print my_str[0]       # Prints first character of the
    string
print my_str[2:5]     # Prints characters starting from 3rd
    to 5th
print my_str[2:]      # Prints string starting from 3rd
    character
print my_str * 2      # Prints string two times
print my_str + "TEST" # Prints concatenated string
```



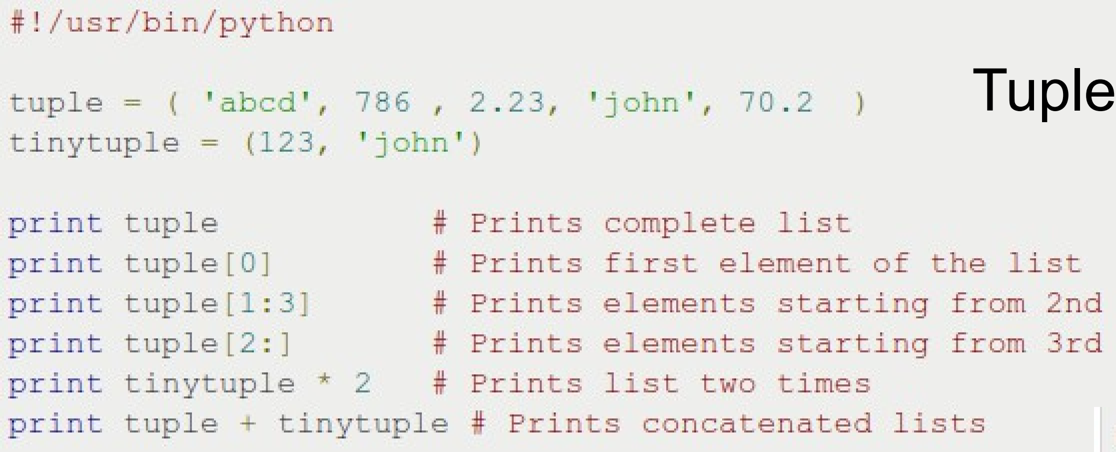
```
$python main.py
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Standard Data Types



```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

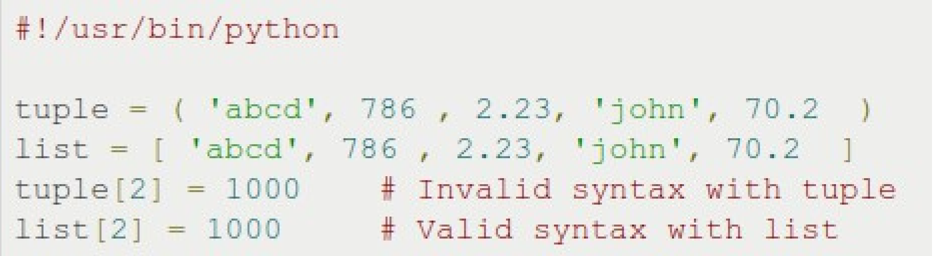
List



```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple           # Prints complete list
print tuple[0]        # Prints first element of the list
print tuple[1:3]      # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

Tuple

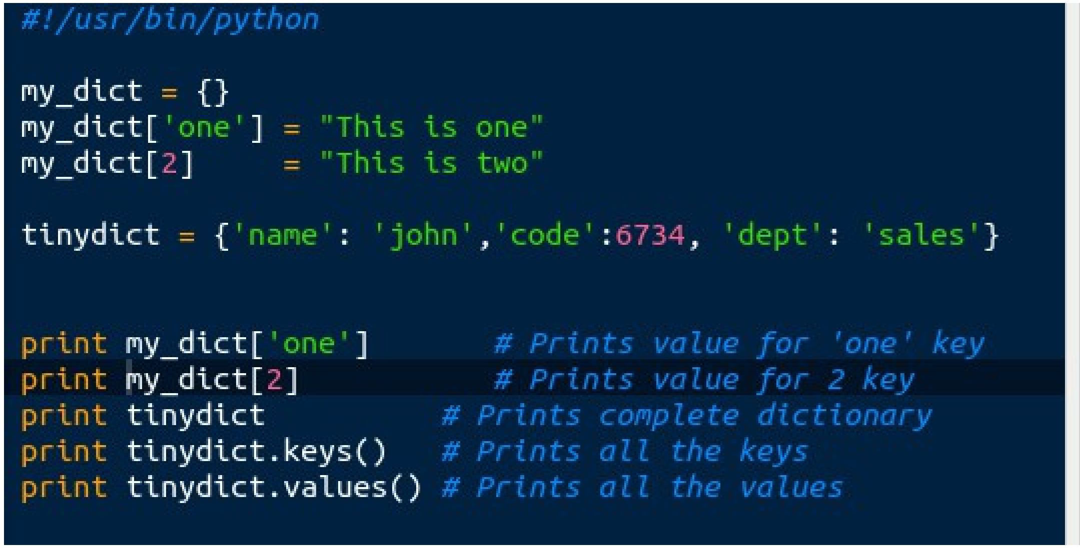
Lists and tuple look the same.
Main difference:
Tuple **CANNOT** be updated !



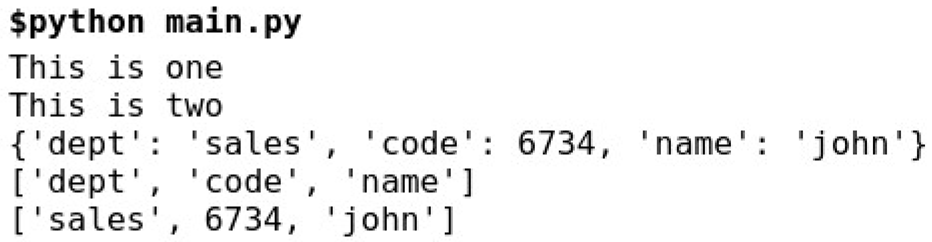
```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Dictionary (Python)

Dictionaries are hash tables (associative arrays). And are composed by **key:values** pairs
keys → can be almost any object
Values → any arbitrary Python objects.
e.g. you can have lists of dictionaries, dictionaries of lists, dictionaries of dictionaries and so on...
Remind that, unlike C++ maps, dictionaris are not ordered !



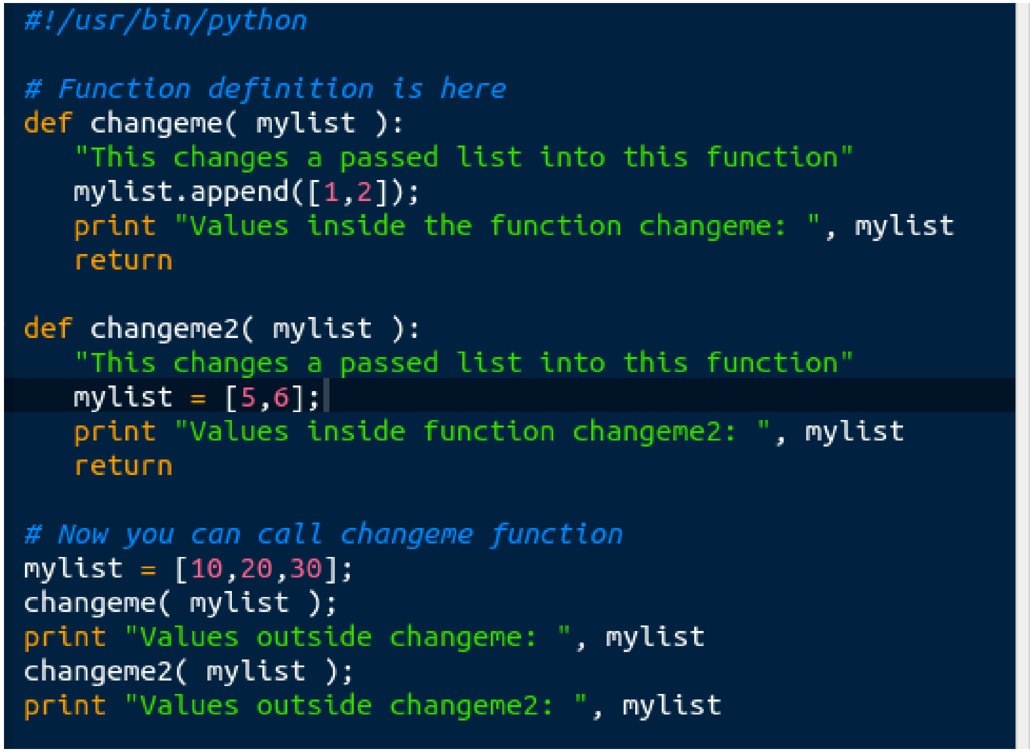
```
#!/usr/bin/python

my_dict = {}
my_dict['one'] = "This is one"
my_dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print my_dict['one']       # Prints value for 'one' key
print my_dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```



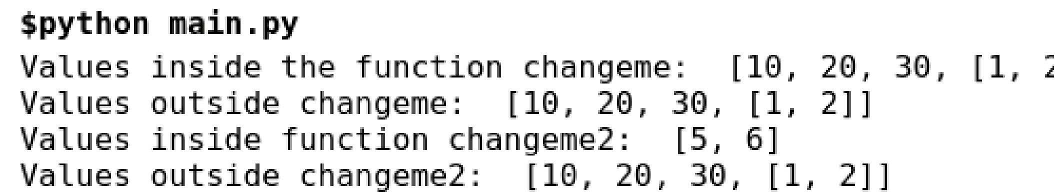
```
$python main.py
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Functions

Functions should be defined BEFORE they are actually called
Arguments are passed by **reference**



```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2]);
   print "Values inside the function changeme: ", mylist
   return

def changeme2( mylist ):
   "This changes a passed list into this function"
   mylist = [5,6];
   print "Values inside function changeme2: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside changeme: ", mylist
changeme2( mylist );
print "Values outside changeme2: ", mylist
```



```
$python main.py
Values inside the function changeme:  [10, 20, 30, [1,
Values outside changeme:  [10, 20, 30, [1, 2]]
Values inside function changeme2:  [5, 6]
Values outside changeme2:  [10, 20, 30, [1, 2]]
```

Modules and Packages

Having all functions in a single file (file.py) becomes quickly cumbersome (thousand of lines of code)
Modules: a file (aModule.py) in python code that contains a function (my_func) that you can call from another file (file.py)

```
In file aModule.py
def my_func(par):
        # do something
        return
```

```
In file file.py
import aModule
aModule.my_func(my_par)
```

```
In file file.py
from aModule import my)func
# you can also use
# from aModule import *
# but use WISELY
my_func(my_par)
```

Python looks for aModule.py in:

- current directory
- directories defined in PYTHONPATH env var
- in /usr/local/lib/python

Packages: hyerarchical file directory structure that defines a single Python application environment consisting of modules

```
$> ls -la MyPackage
MyPackage/__init__.py
MyPackage/module1.py
MyPackage/module2.py
```

```
In file MyPackage/__init__.py
from module1 import Mod1
from module2 import Mod2
```

```
In file.py
import MyPackage
MyPackage.Mod1()
MyPackage.Mod2()
```

And this is the way you can use
Python in ROOT
You need to import package ROOT

PyROOT notebook

How to interface to ROOT?

1) GUI (Graphical User Interface)

2) Command line → quick checks and studies

CINT/CLING (almost C++)

Python prompt (Python)

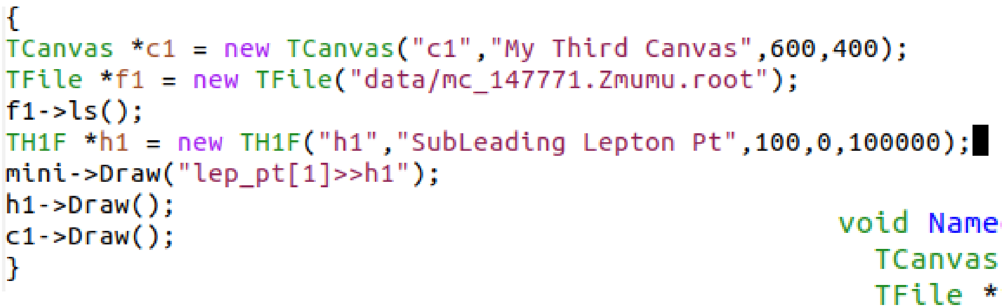
3) ROOT macros: simple or moderate complexity programs, in C++

4) PyROOT scripts: simple or moderate programs, in python

5) Compiled ROOT: complex or CPU-intensive programs, in C++

Macros

Anonymous macros example:

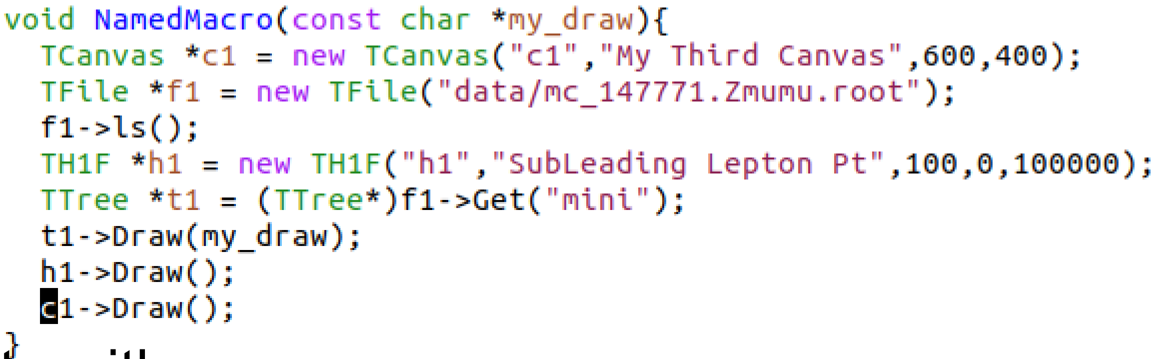


```
{
TCanvas *c1 = new TCanvas("c1","My Third Canvas",600,400);
TFile *f1 = new TFile("data/mc_147771.Zmumu.root");
f1->ls();
TH1F *h1 = new TH1F("h1","SubLeading Lepton Pt",100,0,100000);
mini->Draw("lep_pt[1]>>h1");
h1->Draw();
c1->Draw();
}
```

execute with:
root> .x
scripts/UnNamedMacro.cpp

Named Macros

Pass argument



```
void NamedMacro(const char *my_draw){
  TCanvas *c1 = new TCanvas("c1","My Third Canvas",600,400);
  TFile *f1 = new TFile("data/mc_147771.Zmumu.root");
  f1->ls();
  TH1F *h1 = new TH1F("h1","SubLeading Lepton Pt",100,0,100000);
  TTree *t1 = (TTree*)f1->Get("mini");
  t1->Draw(my_draw);
  h1->Draw();
  c1->Draw();
}
```

execute with:
root> .L scripts/NamedMacro.cpp
root> NamedMacro(“lep_pt[1]>>h1")

But you can still use it as an anomymous macro
root> .x scripts/NamedMacro.cpp(“lep_pt[1]>>h1")

How to interface to ROOT?

1) GUI (Graphical User Interface)

2) Command line → quick checks and studies

CINT/CLING (almost C++)

Python prompt (Python)

3) ROOT macros: simple or moderate complexity programs, in C++

4) PyROOT scripts: simple or moderate programs, in python

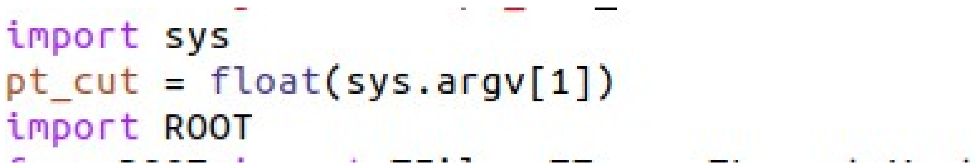
5) Compiled ROOT: complex or CPU-intensive programs, in C++

PyROOT Macros

Not very different from executing from command prompt.

```
$> python scripts/PyRootLoopTree.py  20
```

Passing argument to script



```
import sys
pt_cut = float(sys.argv[1])
import ROOT
```

In principle you can compile also python scripts through py_compile module but it doesn’t speed up things since Python compiles internally if it’s convenient

How to interface to ROOT?

1) GUI (Graphical User Interface)

2) Command line → quick checks and studies

CINT/CLING (almost C++)

Python prompt (Python)

3) ROOT macros: simple or moderate complexity programs, in C++

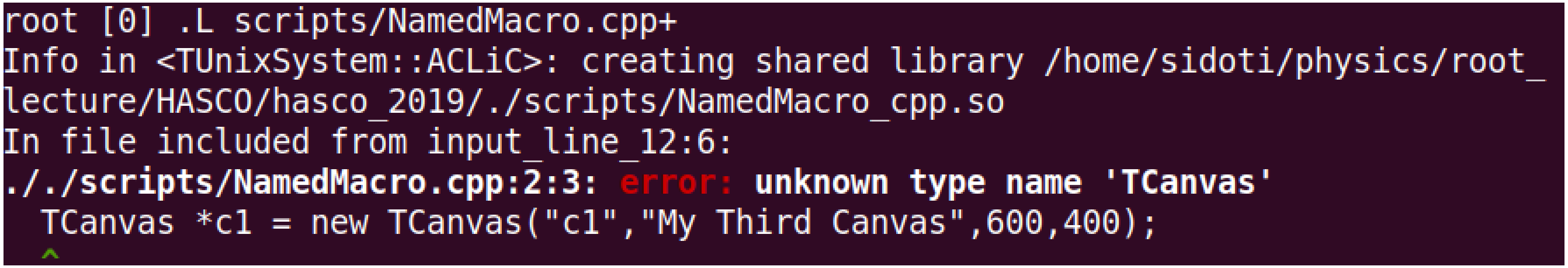
4) PyROOT scripts: simple or moderate programs, in python

5) Compiled ROOT: complex or CPU-intensive programs, in C++

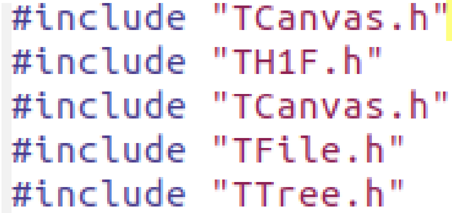
Compiled Macro

Let’s try to compile the macro with AClic

A bunch of errors !



```
root [0] .L scripts/NamedMacro.cpp+
Info in <TUnixSystem::ACLiC>: creating shared library /home/sidoti/physics/root_
lecture/HASCO/hasco_2019/./scripts/NamedMacro_cpp.so
In file included from input_line_12:6:
././scripts/NamedMacro.cpp:2:3: error: unknown type name 'TCanvas'
  TCanvas *c1 = new TCanvas("c1","My Third Canvas",600,400);
  ^
```

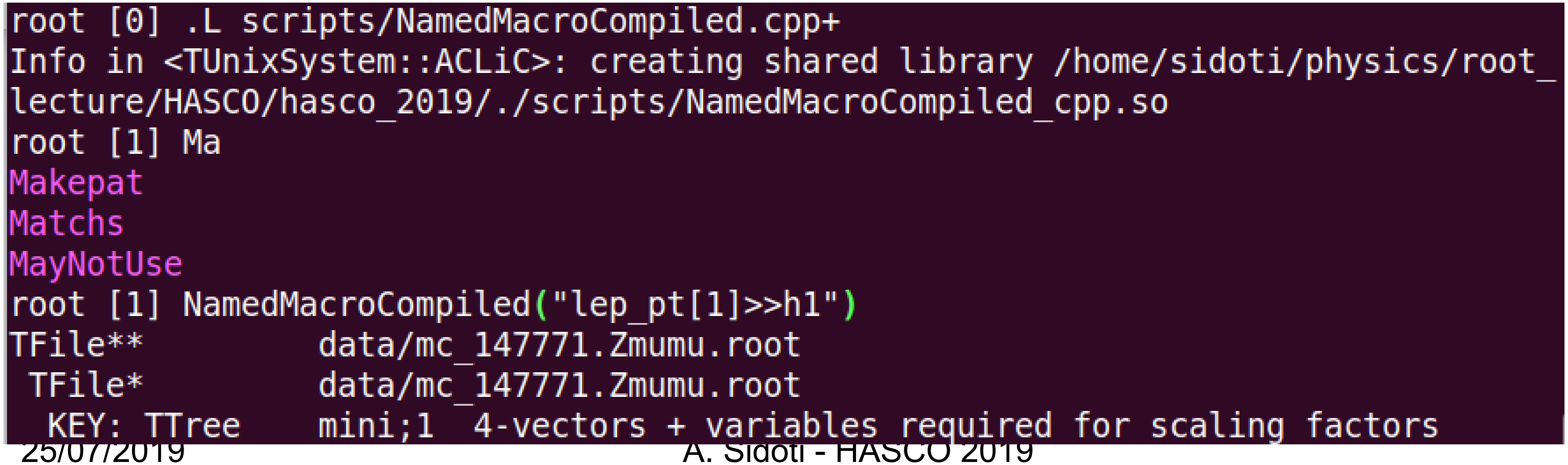


```
#include "TCanvas.h"
#include "TH1F.h"
#include "TCanvas.h"
#include "TFile.h"
#include "TTree.h"
```

Include headers!

.L macroname++ → Forces recompilation

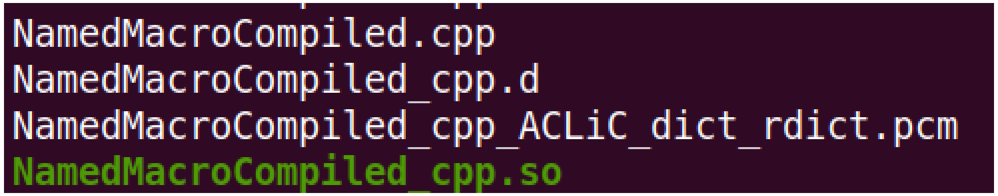
When running compiled macro is faster !



```
root [0] .L scripts/NamedMacroCompiled.cpp+
Info in <TUnixSystem::ACLiC>: creating shared library /home/sidoti/physics/root_
lecture/HASCO/hasco_2019/./scripts/NamedMacroCompiled_cpp.so
root [1] Ma
Makepat
Matchs
MayNotUse
root [1] NamedMacroCompiled("lep_pt[1]>>h1")
TFile**         data/mc_147771.Zmumu.root
 TFile*         data/mc_147771.Zmumu.root
  KEY: TTree    mini;1  4-vectors + variables required for scaling factors
```

Compiled Macros

Produced files



```
NamedMacroCompiled.cpp
NamedMacroCompiled_cpp.d
NamedMacroCompiled_cpp_ACLiC_dict_rdict.pcm
NamedMacroCompiled_cpp.so
```

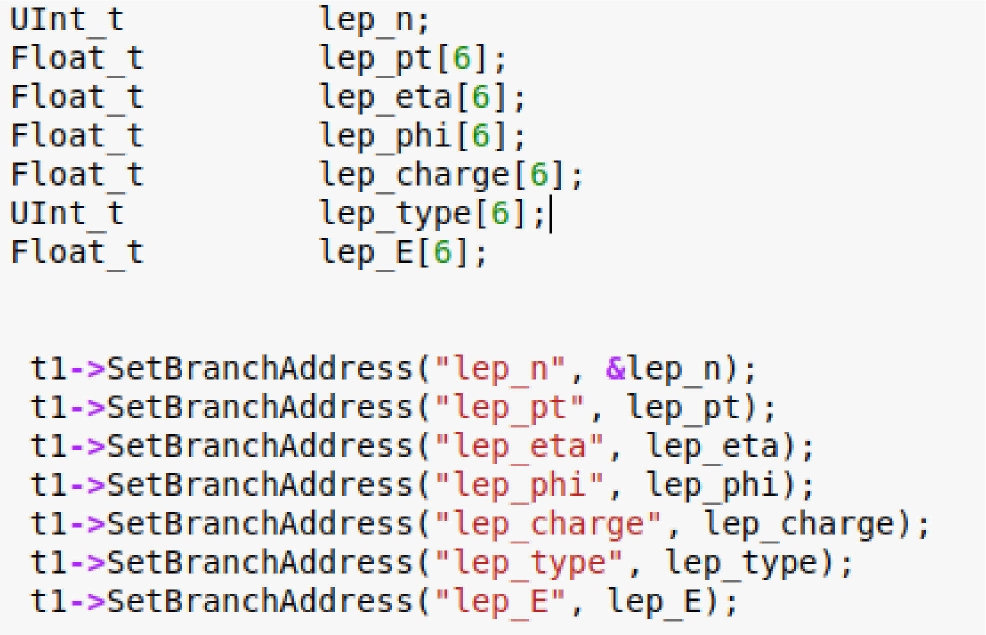
You can load your shared library without recompiling

root[0] gSystem→Load("scripts/NamedMacroCompiled_cpp.so")

root[1] NamedMacroCompiled("lep_pt[1]>>h1")

MakeClass

Remember our example to loop on ROOT-tuple event?
→ You have to figure out which variables are in the ntuple →
tedious/long/and error prone



```
UInt_t          lep_n;
Float_t         lep_pt[6];
Float_t         lep_eta[6];
Float_t         lep_phi[6];
Float_t         lep_charge[6];
UInt_t          lep_type[6];
Float_t         lep_E[6];


 t1->SetBranchAddress("lep_n", &lep_n);
 t1->SetBranchAddress("lep_pt", lep_pt);
 t1->SetBranchAddress("lep_eta", lep_eta);
 t1->SetBranchAddress("lep_phi", lep_phi);
 t1->SetBranchAddress("lep_charge", lep_charge);
 t1->SetBranchAddress("lep_type", lep_type);
 t1->SetBranchAddress("lep_E", lep_E);
```

TTree::MakeClass builds a skeleton code for you

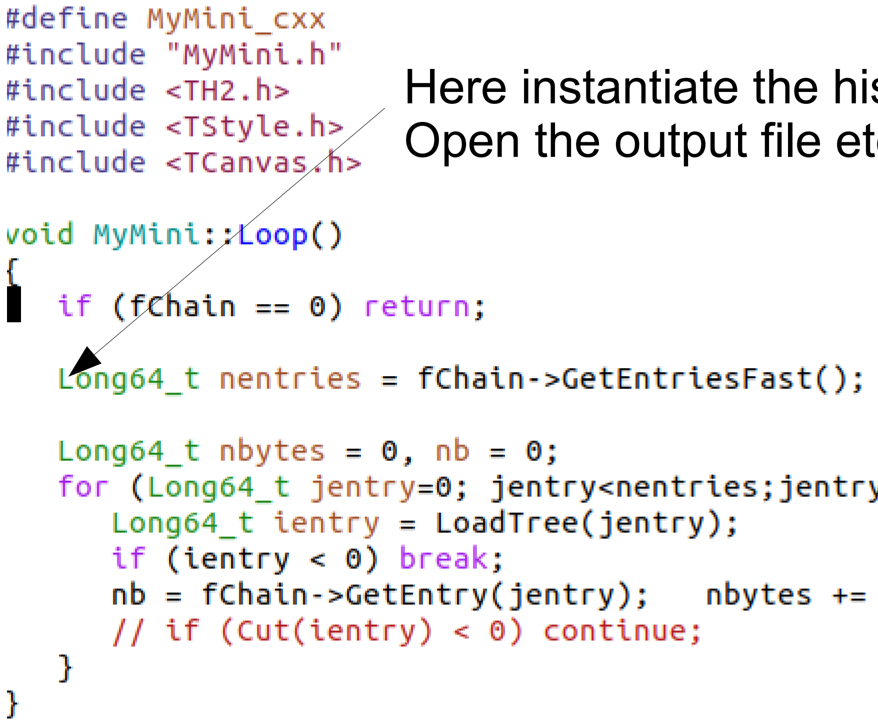
```
$>root -l data/mc_147771.Zmumu.root
root[0] mini->MakeClass("MyMini")
Info in <TTreePlayer::MakeClass>:
Files: MyMini.h and MyMini.C generated
from
TTree: mini
```

Snippet of MyMini.h (it's a class)

```
   // Declaration of leaf types
   Int_t           runNumber;
   Int_t           eventNumber;
   UInt_t          lep_n;
   Float_t         lep_pt[6];   //[lep_n]
   Float_t         lep_eta[6];   //[lep_n]
   Float_t         lep_phi[6];   //[lep_n]
   // List of branches
   TBranch        *b_runNumber;   //!
   TBranch        *b_eventNumber;   //!
   TBranch        *b_lep_n;   //!
   TBranch        *b_lep_pt;   //!
   TBranch        *b_jet_eta;   //!
   TBranch        *b_jet_phi;   //!

   MyMini(TTree *tree=0);
   virtual ~MyMini();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};

#endif
```

Here goes the stuff you need
Histogram declaration TH1F *h1;
Function declaration
Output file declaration

Snippet of MyMini.C



```
#define MyMini_cxx
#include "MyMini.h"
#include <TH2.h>
#include <TStyle.h>
#include <TCanvas.h>

void MyMini::Loop()
{
   if (fChain == 0) return;

   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      // if (Cut(ientry) < 0) continue;
   }
}
```

Here instantiate the histograms
Open the output file etc...

Control the Loop()
Here you are accessing the events.

Note! There is no main() !

How to Use it?

1) Compile with AcliC (from root command line). **RECOMENDED**

```
Root> .L MyMini.C+
Root> MyMini my_ana
Root> my_ana.Loop()
```

2) Makefile (obsolete) → but will show it

3) cmake (new default) → Will not show it (far from being an expert)

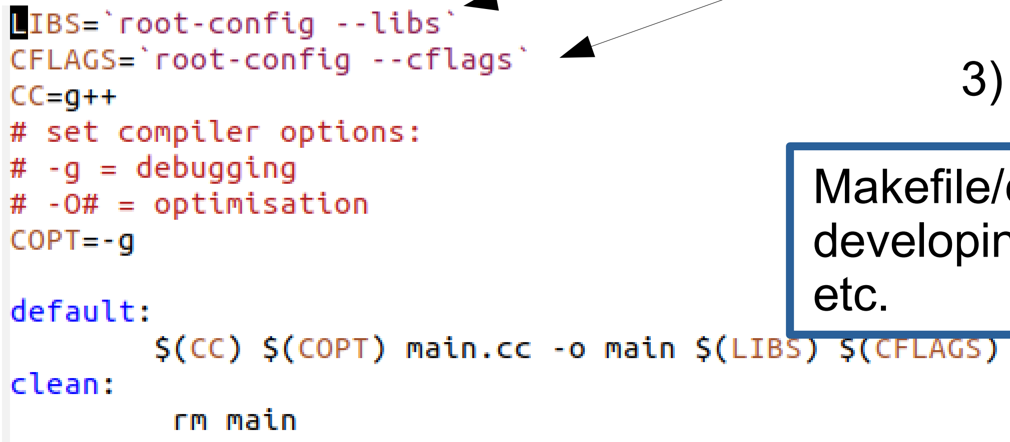
Makefile

1) Implement a file (main.cc) with main() that will call the code you've implemented with MakeClass

```
#include <iostream>
#include "MyMini.cxx"
int main(int argc, char * argv[]);
int main(int argc, char * argv[]) {
  MyMini *pippo;
  TFile *f1 = new TFile("../data/mc_147771.Zmumu.root");
  TTree *t1 = (TTree*)f1->Get("mini");
  pippo = new MyMini(t1);
  pippo->Loop();
  return 0;
}
```

2) Write a Makefile

Root commands to determine flags and libraries for compiler



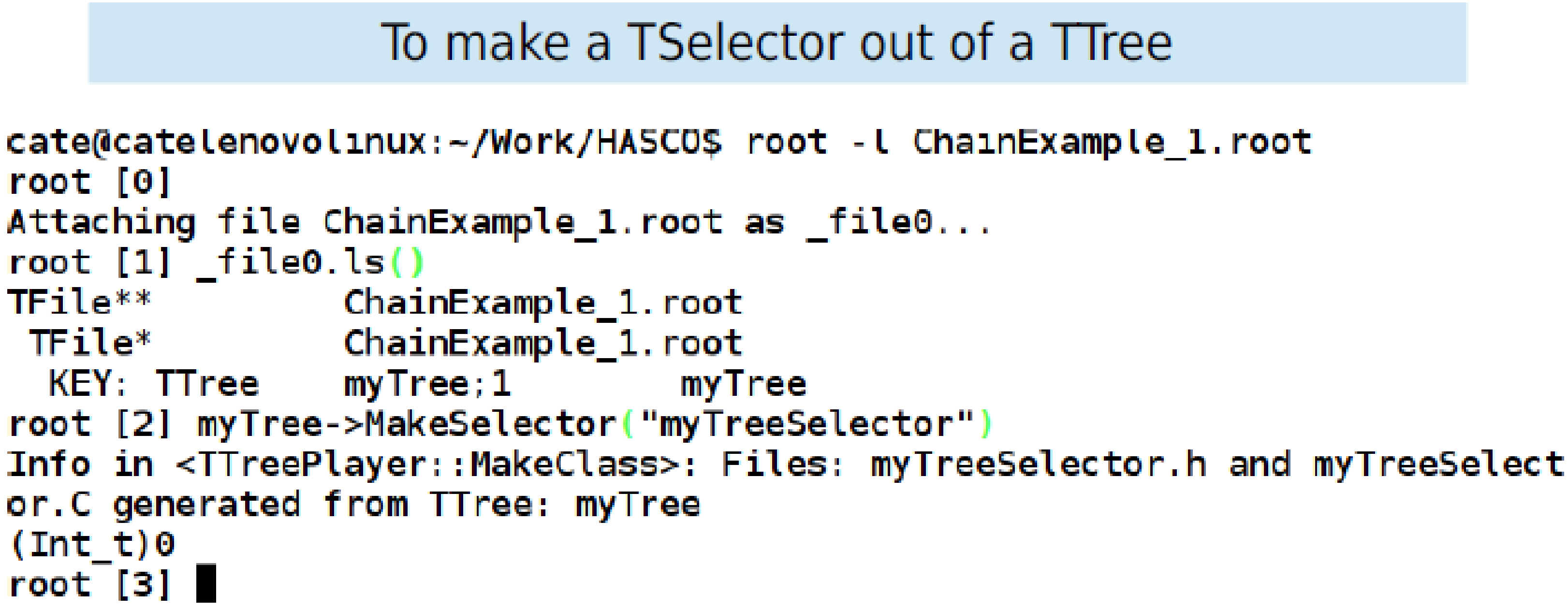
```
LIBS=`root-config --libs`
CFLAGS=`root-config --cflags`
CC=g++
# set compiler options:
# -g = debugging
# -O# = optimisation
COPT=-g

default:
        $(CC) $(COPT) main.cc -o main $(LIBS) $(CFLAGS)
clean:
         rm main
```

3) Execute witn $> ./main

Makefile/cmake starts to become useful if you are developing a large project with many source files etc.

MakeSelector

Another possibility is to create a Selector

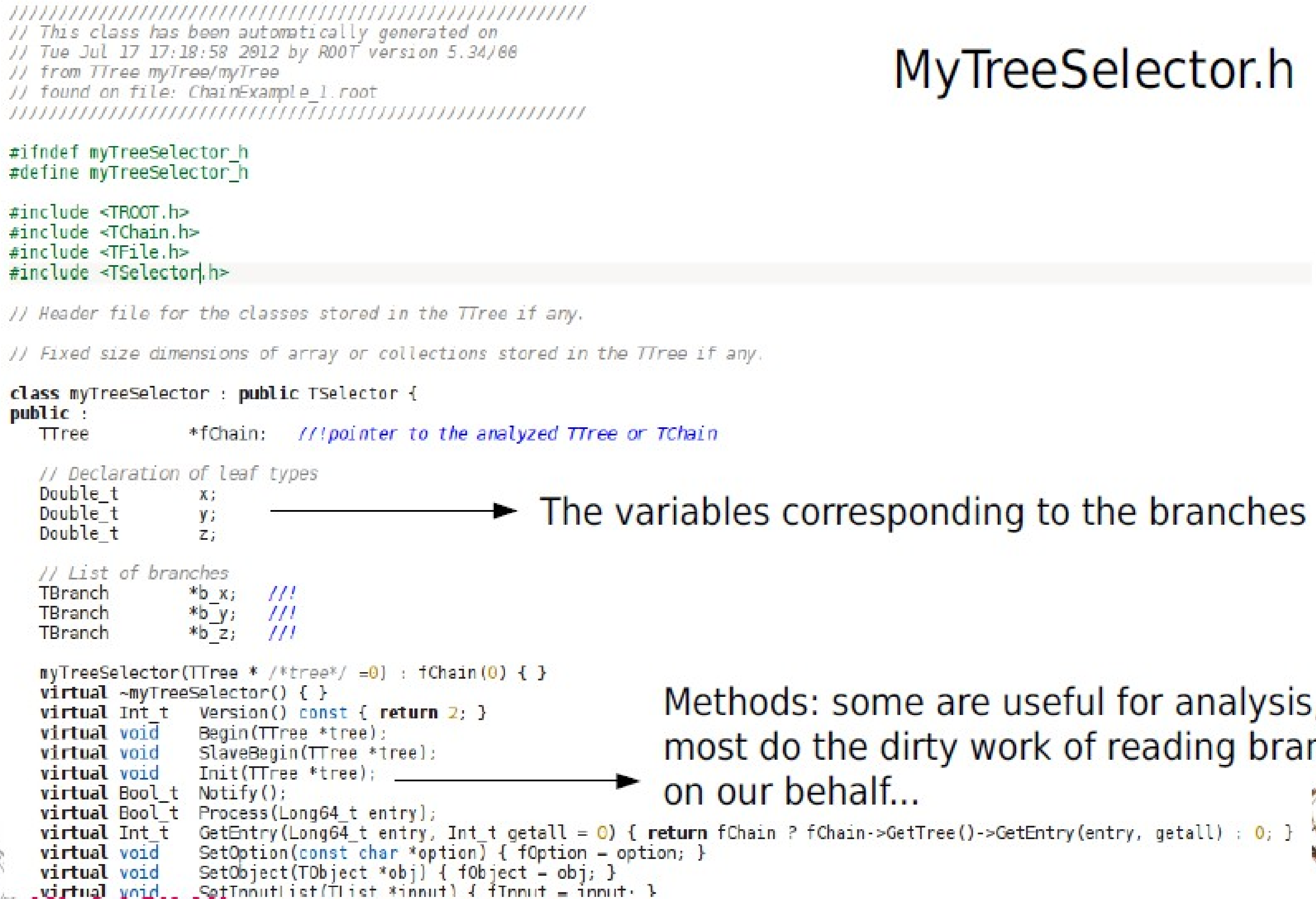


To make a TSelector out of a TTree

```
cate@catelenovolinux:~/Work/HASCO$ root -l ChainExample_1.root
root [0]
Attaching file ChainExample_1.root as _file0...
root [1] _file0.ls()
TFile**         ChainExample_1.root
 TFile*         ChainExample_1.root
  KEY: TTree    myTree;1        myTree
root [2] myTree->MakeSelector("myTreeSelector")
Info in <TTreePlayer::MakeClass>: Files: myTreeSelector.h and myTreeSelect
or.C generated from TTree: myTree
(Int_t)0
root [3] 
```

Following slides are from C. Doglioni and A. Andreazza 2012 HASCO Slides

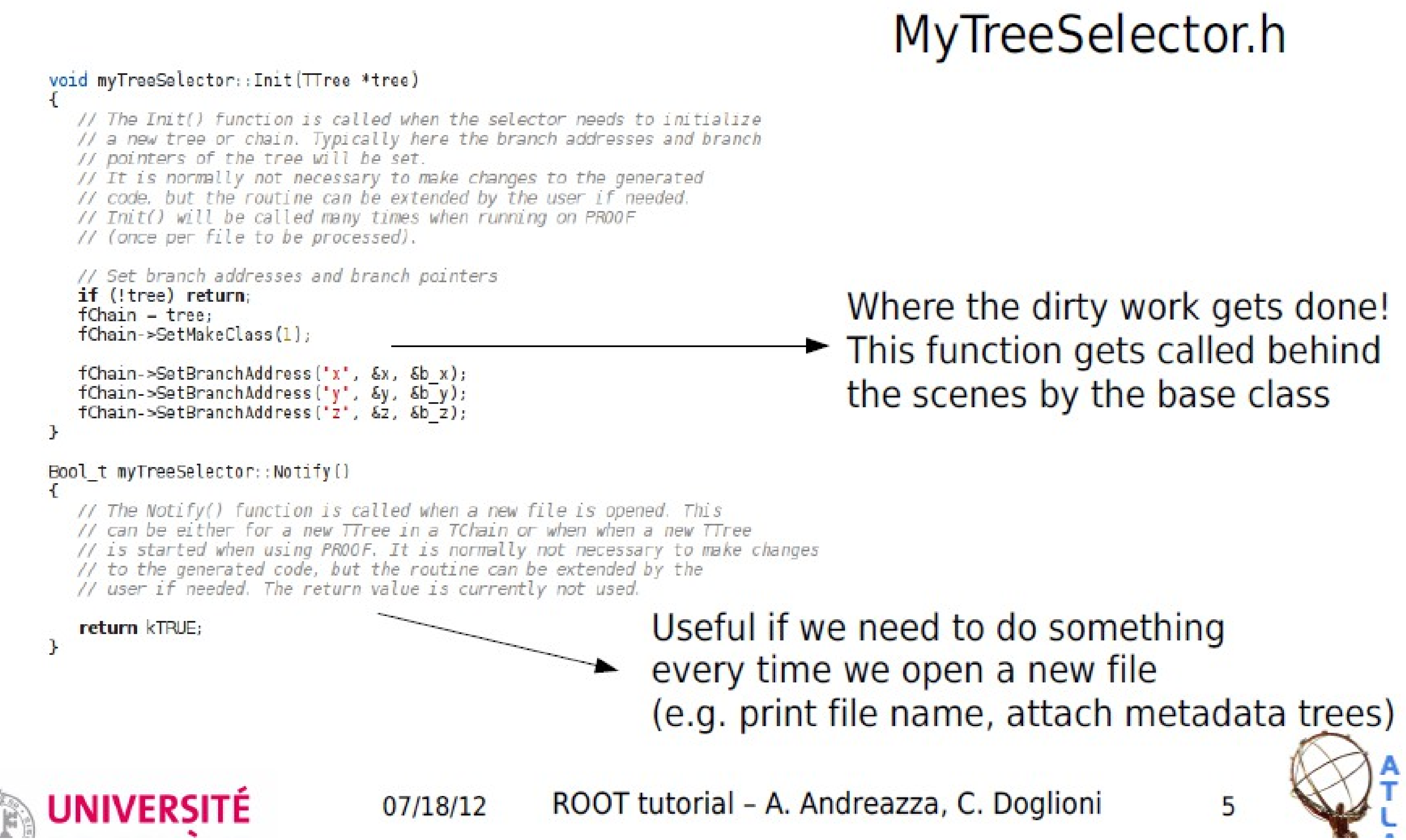
MyTreeSelector.h



```cpp
//////////////////////////////////////////////////////////
// This class has been automatically generated on
// Tue Jul 17 17:18:58 2012 by ROOT version 5.34/00
// from TTree myTree/myTree
// found on file: ChainExample_1.root
//////////////////////////////////////////////////////////

#ifndef myTreeSelector_h
#define myTreeSelector_h

#include <TROOT.h>
#include <TChain.h>
#include <TFile.h>
#include <TSelector.h>

// Header file for the classes stored in the TTree if any.

// Fixed size dimensions of array or collections stored in the TTree if any.

class myTreeSelector : public TSelector {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain

   // Declaration of leaf types
   Double_t        x;
   Double_t        y;
   Double_t        z;

   // List of branches
   TBranch        *b_x;   //!
   TBranch        *b_y;   //!
   TBranch        *b_z;   //!

   myTreeSelector(TTree * /*tree*/ =0) : fChain(0) { }
   virtual ~myTreeSelector() { }
   virtual Int_t   Version() const { return 2; }
   virtual void    Begin(TTree *tree);
   virtual void    SlaveBegin(TTree *tree);
   virtual void    Init(TTree *tree);
   virtual Bool_t  Notify();
   virtual Bool_t  Process(Long64_t entry);
   virtual Int_t   GetEntry(Long64_t entry, Int_t getall = 0) { return fChain ? fChain->GetTree()->GetEntry(entry, getall) : 0; }
   virtual void    SetOption(const char *option) { fOption = option; }
   virtual void    SetObject(TObject *obj) { fObject = obj; }
   virtual void    SetInputList(TList *input) { fInput = input; }
```

The variables corresponding to the branches

Methods: some are useful for analysis, most do the dirty work of reading branches on our behalf...



MyTreeSelector.h

```
void myTreeSelector::Init(TTree *tree)
{
   // The Init() function is called when the selector needs to initialize
   // a new tree or chain. Typically here the branch addresses and branch
   // pointers of the tree will be set.
   // It is normally not necessary to make changes to the generated
   // code, but the routine can be extended by the user if needed.
   // Init() will be called many times when running on PROOF
   // (once per file to be processed).

   // Set branch addresses and branch pointers
   if (!tree) return;
   fChain = tree;
   fChain->SetMakeClass(1);

   fChain->SetBranchAddress("x", &x, &b_x);
   fChain->SetBranchAddress("y", &y, &b_y);
   fChain->SetBranchAddress("z", &z, &b_z);
}

Bool_t myTreeSelector::Notify()
{
   // The Notify() function is called when a new file is opened. This
   // can be either for a new TTree in a TChain or when when a new TTree
   // is started when using PROOF. It is normally not necessary to make changes
   // to the generated code, but the routine can be extended by the
   // user if needed. The return value is currently not used.

   return kTRUE;
}
```

Where the dirty work gets done!
This function gets called behind
the scenes by the base class

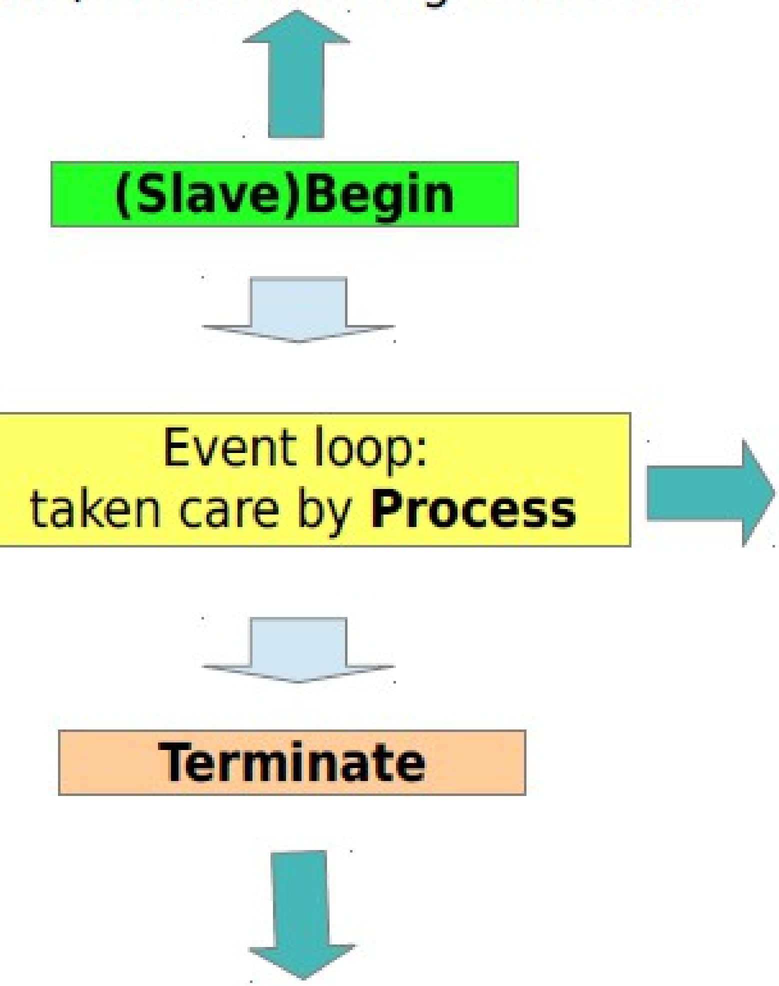
Useful if we need to do something
every time we open a new file
(e.g. print file name, attach metadata trees)

MyTreeSelector.C

```
#include "myTreeSelector.h"
#include <TH2.h>
#include <TStyle.h>


void myTreeSelector::Begin(TTree * /*tree*/)
{
   // The Begin() function is called at the start of the query.
   // When running with PROOF Begin() is only called on the client.
   // The tree argument is deprecated (on PROOF 0 is passed).

   TString option = GetOption();

}

void myTreeSelector::SlaveBegin(TTree * /*tree*/)
{

Bool_t myTreeSelector::Process(Long64_t entry)
{

void myTreeSelector::SlaveTerminate()
{

void myTreeSelector::Terminate()
{
```

All methods are empty!
Up to the user to do what he/she wants in them...

ook/initialise histograms here

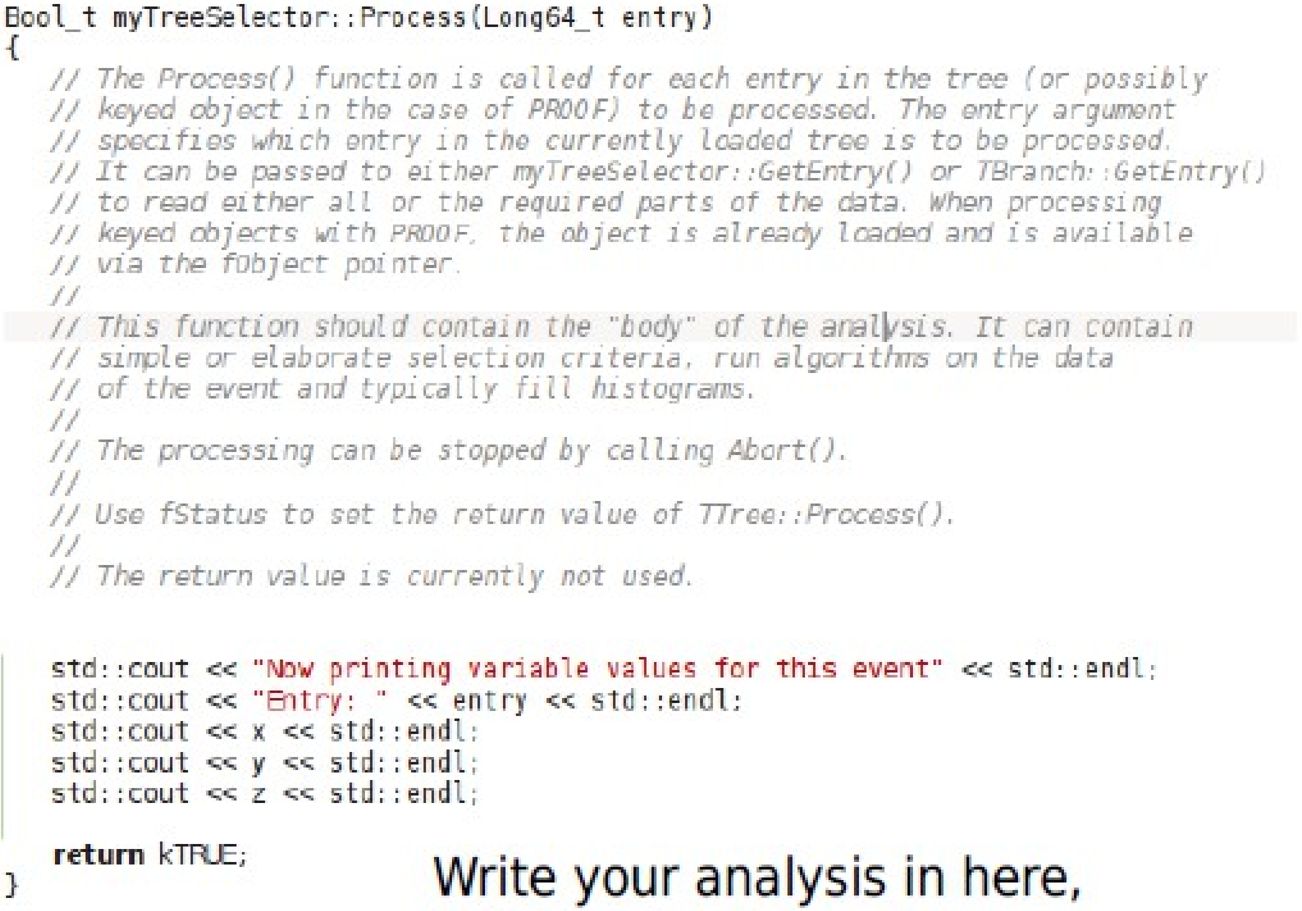


(Slave)Begin

Event loop:
taken care by **Process**

Terminate

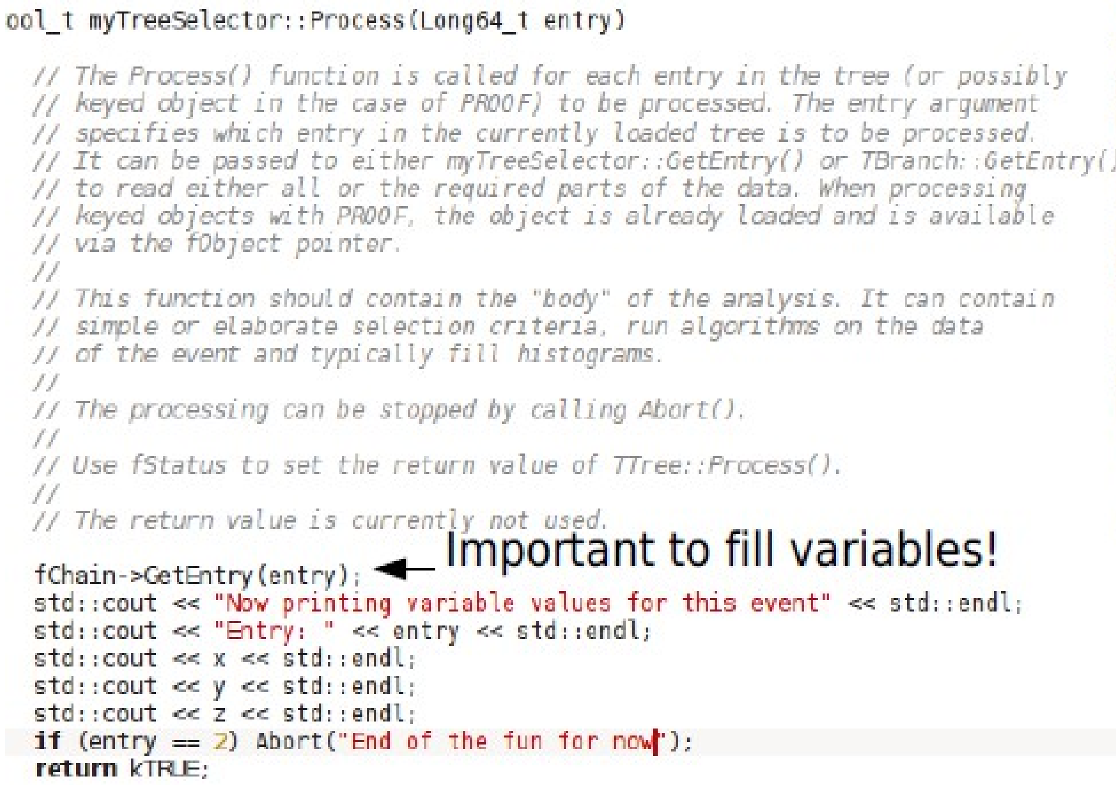
Write out histograms to file here



```
Bool_t myTreeSelector::Process(Long64_t entry)
{
   // The Process() function is called for each entry in the tree (or possibly
   // keyed object in the case of PROOF) to be processed. The entry argument
   // specifies which entry in the currently loaded tree is to be processed.
   // It can be passed to either myTreeSelector::GetEntry() or TBranch::GetEntry()
   // to read either all or the required parts of the data. When processing
   // keyed objects with PROOF, the object is already loaded and is available
   // via the fObject pointer.
   //
   // This function should contain the "body" of the analysis. It can contain
   // simple or elaborate selection criteria, run algorithms on the data
   // of the event and typically fill histograms.
   //
   // The processing can be stopped by calling Abort().
   //
   // Use fStatus to set the return value of TTree::Process().
   //
   // The return value is currently not used.


   std::cout << "Now printing variable values for this event" << std::endl;
   std::cout << "Entry: "  << entry << std::endl;
   std::cout << x << std::endl;
   std::cout << y << std::endl;
   std::cout << z << std::endl;

   return kTRUE;
}
```

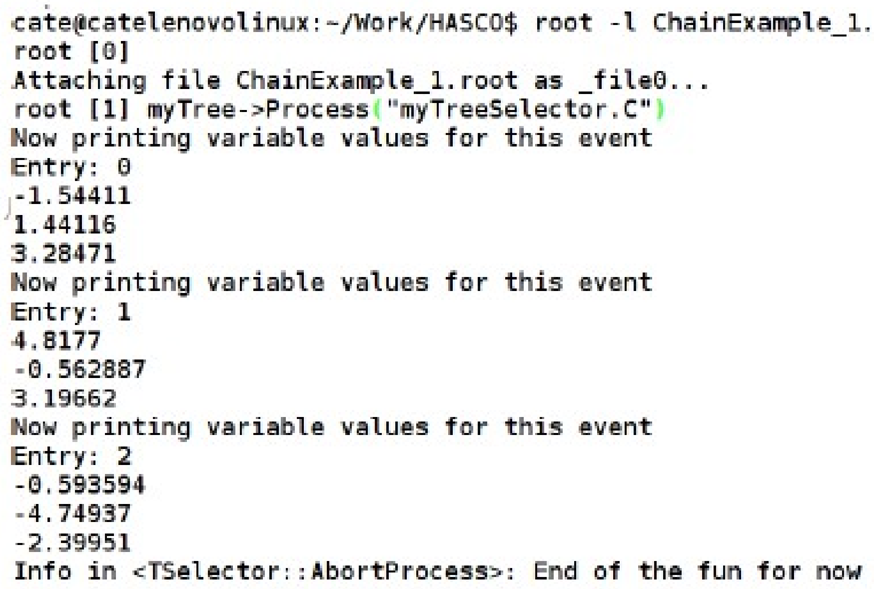
Write your analysis in here, taking advantage of easy access of variables: they will be filled for you in the same way we did when filling the TTree

Interactive: very simple (as written in .C file)
Can also use makefile…recommended (faster)

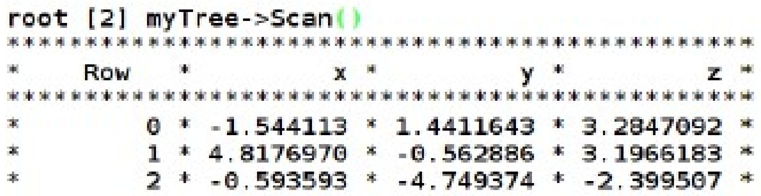


```
ool_t myTreeSelector::Process(Long64_t entry)

  // The Process() function is called for each entry in the tree (or possibly
  // keyed object in the case of PROOF) to be processed. The entry argument
  // specifies which entry in the currently loaded tree is to be processed.
  // It can be passed to either myTreeSelector::GetEntry() or TBranch::GetEntry()
  // to read either all or the required parts of the data. When processing
  // keyed objects with PROOF, the object is already loaded and is available
  // via the fObject pointer.
  //
  // This function should contain the "body" of the analysis. It can contain
  // simple or elaborate selection criteria, run algorithms on the data
  // of the event and typically fill histograms.
  //
  // The processing can be stopped by calling Abort().
  //
  // Use fStatus to set the return value of TTree::Process().
  //
  // The return value is currently not used.

  fChain->GetEntry(entry);
  std::cout << "Now printing variable values for this event" << std::endl;
  std::cout << "Entry: " << entry << std::endl;
  std::cout << x << std::endl;
  std::cout << y << std::endl;
  std::cout << z << std::endl;
  if (entry == 2) Abort("End of the fun for now");
  return kTRUE;
```

← Important to fill variables!



```
cate@catelenovolinux:~/Work/HASCO$ root -l ChainExample_1.
root [0]
Attaching file ChainExample_1.root as _file0...
root [1] myTree->Process("myTreeSelector.C")
Now printing variable values for this event
Entry: 0
-1.54411
1.44116
3.28471
Now printing variable values for this event
Entry: 1
4.8177
-0.562887
3.19662
Now printing variable values for this event
Entry: 2
-0.593594
-4.74937
-2.39951
Info in <TSelector::AbortProcess>: End of the fun for now
```



```
root [2] myTree->Scan()
************************************************
*    Row   *         x *         y *         z *
************************************************
*        0 * -1.544113 * 1.4411643 * 3.2847092 *
*        1 * 4.8176970 * -0.562886 * 3.1966183 *
*        2 * -0.593593 * -4.749374 * -2.399507 *
```

TODO

That's almost all you need to know for tomorrow's hands on

(after social dinner...)

References

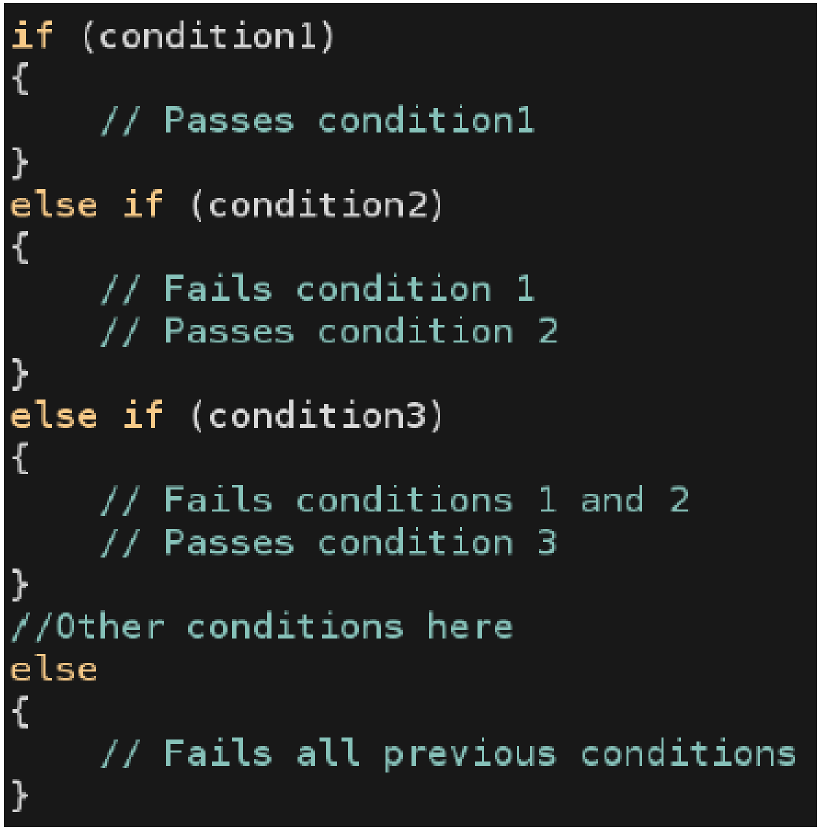
- Notebook folders https://cernbox.cern.ch/index.php/s/oTRjgmouHr9Lf4a password: hasco2019
- Python Crash Course https://github.com/MrAlex6204/Books/blob/master/python-crash-course.
- Root tutorials at previous HASCO editions
- Root forum (was roottalk in the past)
- Root web site http://www.root.cern.ch/

BackUp

Conditional statements

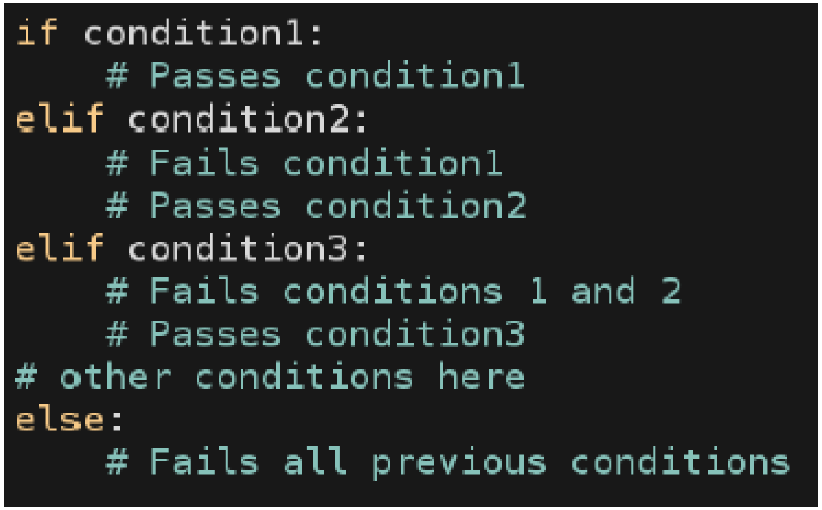
- The conditional statement allows you to control code flow
  - Code no longer needs to be designed for a single fully-determined task
  - Arguably the most important development in programming

C++



```cpp
if (condition1)
{
    // Passes condition1
}
else if (condition2)
{
    // Fails condition 1
    // Passes condition 2
}
else if (condition3)
{
    // Fails conditions 1 and 2
    // Passes condition 3
}
//Other conditions here
else
{
    // Fails all previous conditions
}
```

Python

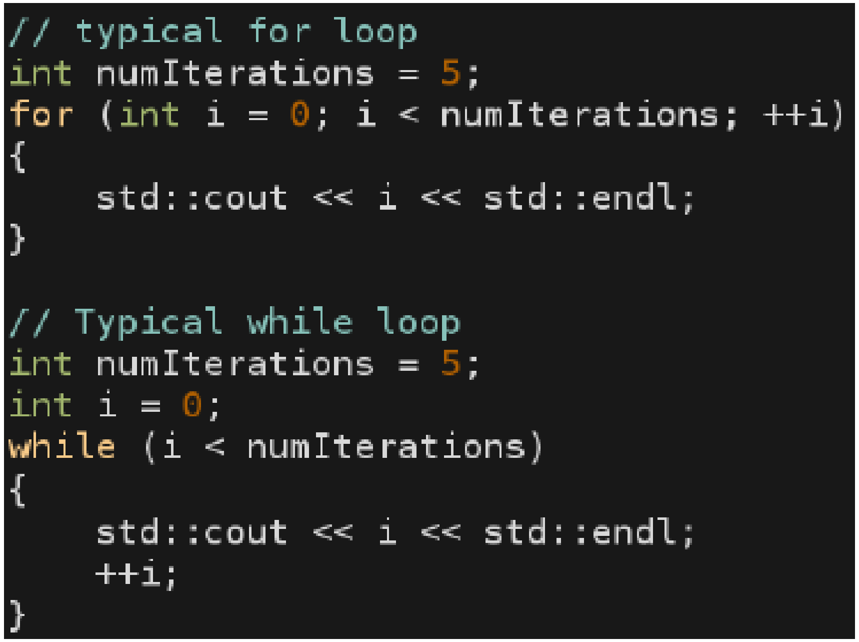


```python
if condition1:
    # Passes condition1
elif condition2:
    # Fails condition1
    # Passes condition2
elif condition3:
    # Fails conditions 1 and 2
    # Passes condition3
# other conditions here
else:
    # Fails all previous conditions
```

Iterative statements (loops)

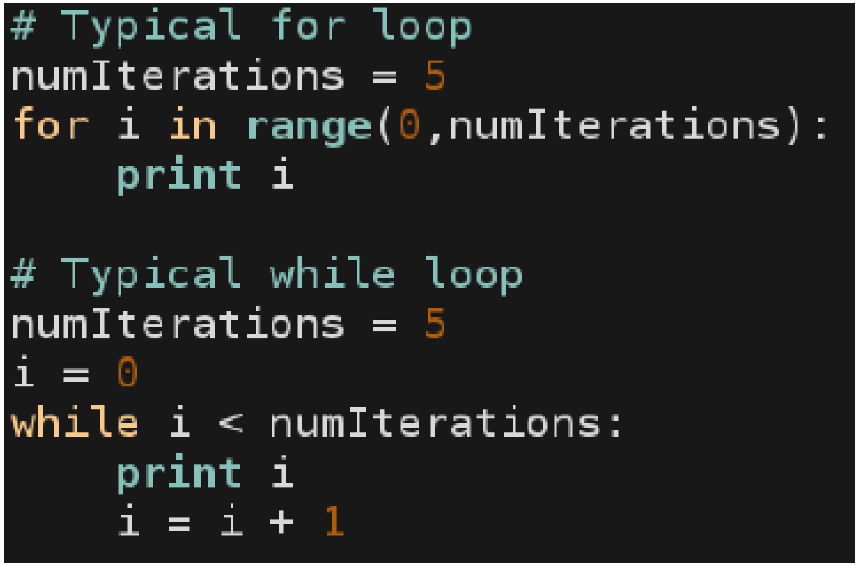
- The iterative statement allows you to simplify code
  - No need to write the same thing over and over
  - Combined with conditions, you can iterate over many items

C++



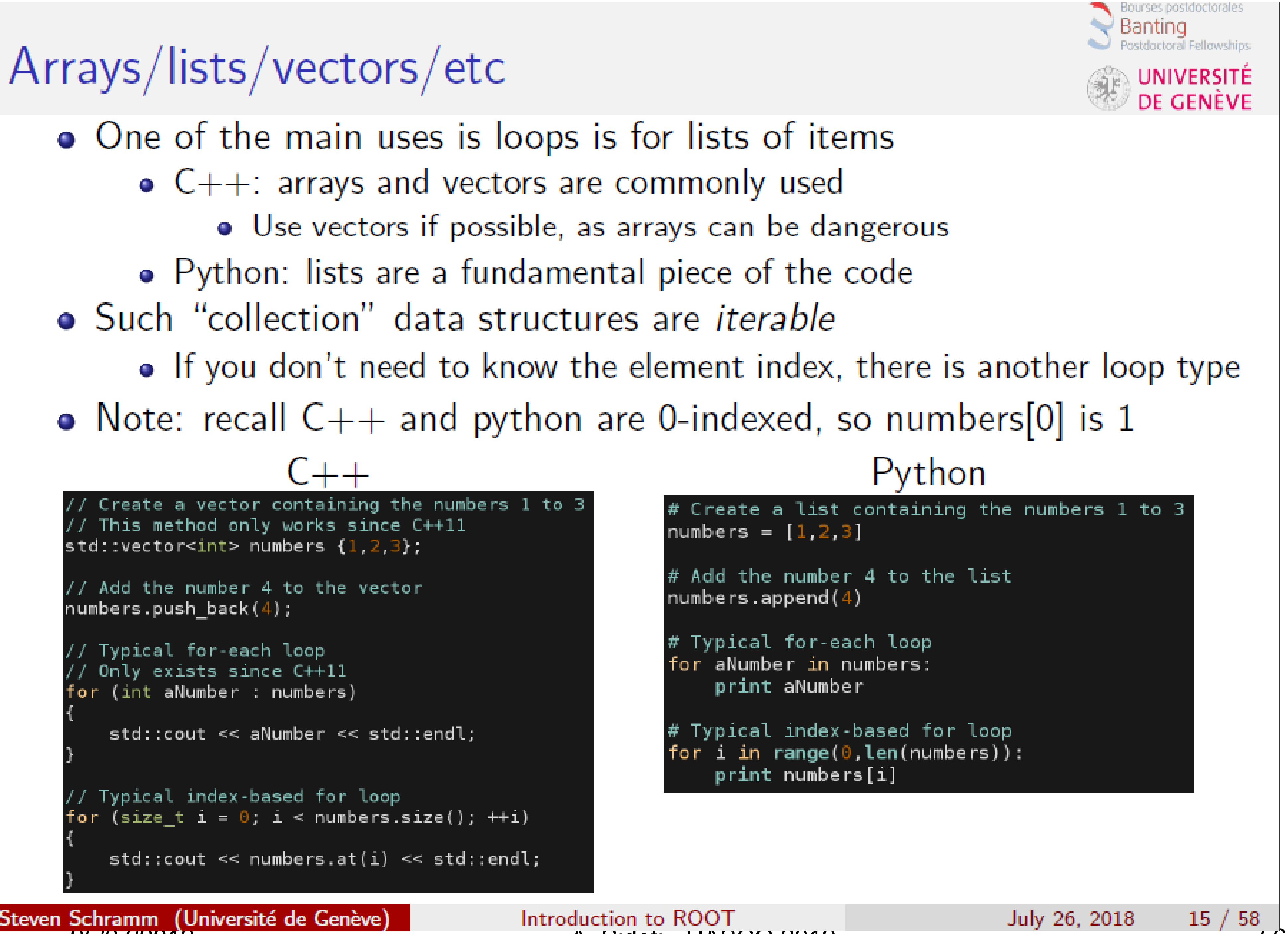
```cpp
// typical for loop
int numIterations = 5;
for (int i = 0; i < numIterations; ++i)
{
    std::cout << i << std::endl;
}

// Typical while loop
int numIterations = 5;
int i = 0;
while (i < numIterations)
{
    std::cout << i << std::endl;
    ++i;
}
```

Python



```python
# Typical for loop
numIterations = 5
for i in range(0,numIterations):
    print i

# Typical while loop
numIterations = 5
i = 0
while i < numIterations:
    print i
    i = i + 1
```

- All of the above will print out the numbers between 0 and 4, with each number on a separate line



Arrays/lists/vectors/etc

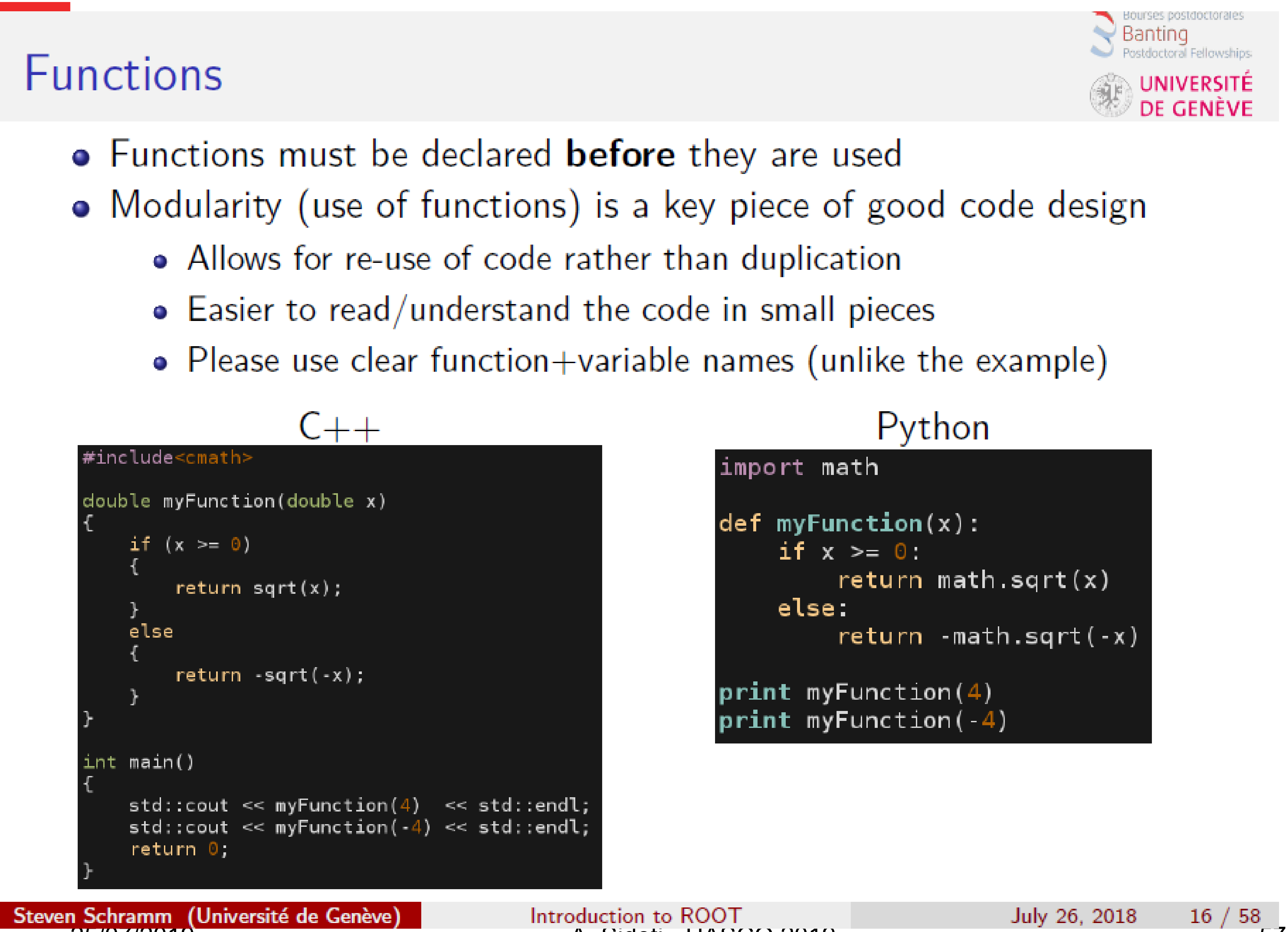
- One of the main uses is loops is for lists of items
  - C++: arrays and vectors are commonly used
    - Use vectors if possible, as arrays can be dangerous
  - Python: lists are a fundamental piece of the code
- Such "collection" data structures are *iterable*
  - If you don't need to know the element index, there is another loop type
- Note: recall C++ and python are 0-indexed, so numbers[0] is 1

C++

```cpp
// Create a vector containing the numbers 1 to 3
// This method only works since C++11
std::vector<int> numbers {1,2,3};

// Add the number 4 to the vector
numbers.push_back(4);

// Typical for-each loop
// Only exists since C++11
for (int aNumber : numbers)
{
    std::cout << aNumber << std::endl;
}

// Typical index-based for loop
for (size_t i = 0; i < numbers.size(); ++i)
{
    std::cout << numbers.at(i) << std::endl;
}
```

Python

```python
# Create a list containing the numbers 1 to 3
numbers = [1,2,3]

# Add the number 4 to the list
numbers.append(4)

# Typical for-each loop
for aNumber in numbers:
    print aNumber

# Typical index-based for loop
for i in range(0,len(numbers)):
    print numbers[i]
```



Functions

Bourses postdoctorales Banting Postdoctoral Fellowships

UNIVERSITÉ DE GENÈVE

- Functions must be declared **before** they are used
- Modularity (use of functions) is a key piece of good code design
    - Allows for re-use of code rather than duplication
    - Easier to read/understand the code in small pieces
    - Please use clear function+variable names (unlike the example)

C++

```cpp
#include<cmath>

double myFunction(double x)
{
    if (x >= 0)
    {
        return sqrt(x);
    }
    else
    {
        return -sqrt(-x);
    }
}

int main()
{
    std::cout << myFunction(4)  << std::endl;
    std::cout << myFunction(-4) << std::endl;
    return 0;
}
```

Python

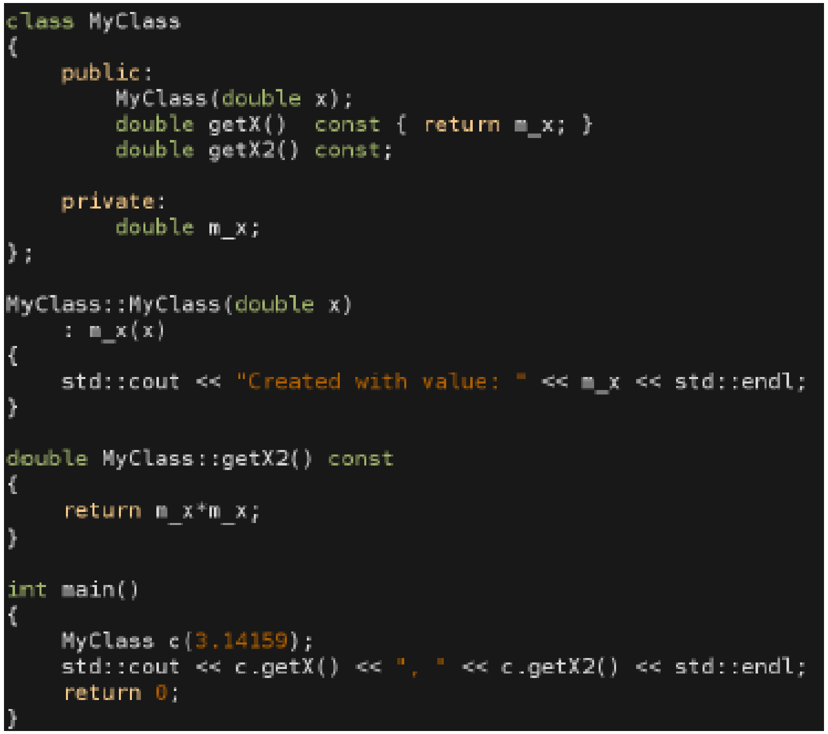
```python
import math

def myFunction(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return -math.sqrt(-x)

print myFunction(4)
print myFunction(-4)
```

Steven Schramm (Université de Genève) Introduction to ROOT July 26, 2018 16 / 58

Classes

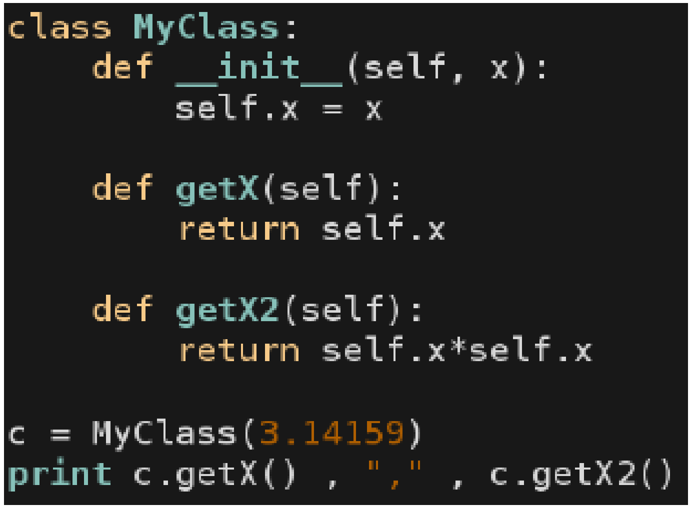
- Object-oriented programming is based on the notion of classes
  - Useful way of grouping similar concepts/information
  - Classes have both a state (variables) and behaviour (functions)
- You may not need to write classes too often (depends on your usage)
- However, all of the ROOT objects you work with are classes
  - histograms, files, trees, fits, etc

C++



```cpp
class MyClass
{
    public:
        MyClass(double x);
        double getX()  const { return m_x; }
        double getX2() const;

    private:
        double m_x;
};

MyClass::MyClass(double x)
    : m_x(x)
{
    std::cout << "Created with value: " << m_x << std::endl;
}

double MyClass::getX2() const
{
    return m_x*m_x;
}

int main()
{
    MyClass c(3.14159);
    std::cout << c.getX() << ", " << c.getX2() << std::endl;
    return 0;
}
```

Python



```python
class MyClass:
    def __init__(self, x):
        self.x = x

    def getX(self):
        return self.x

    def getX2(self):
        return self.x*self.x

c = MyClass(3.14159)
print c.getX() , "," , c.getX2()
```