Contents

Chapter 1

Python Interface

Python is a popular, open-source, dynamic programming language with an interactive interpreter. Its interoperability with other programming languages, both for extending Python as well as embedding it, is excellent and many existing third-party applications and libraries have therefore so-called "Python bindings." PyROOT provides Python bindings for ROOT: it enables cross-calls from ROOT/Cling into Python and vice versa, the intermingling of the two interpreters, and the transport of user-level objects from one interpreter to the other. PyROOT enables access from ROOT to any application or library that itself has Python bindings, and it makes all ROOT functionality directly available from the python interpreter.

1.1 PyROOT Overview

The Python scripting language is widely used for scientific programming, including high performance and distributed parallel code (see http://www.scipy.org). It is the second most popular scripting language (after Perl) and enjoys a wide-spread use as a "glue language": practically every library and application these days comes with Python bindings (and if not, they can be easily written or generated).

`PyROOT`, a Python extension module, provides the bindings for the ROOT class library in a generic way using the Cling dictionary. This way, it allows the use of any ROOT classes from the Python interpreter, and thus the "glue-ing" of ROOT libraries with any non-ROOT library or applications that provide Python bindings. Further, `PyROOT` can be loaded into the Cling interpreter to allow (as of now still rudimentary) access to Python classes. The best way to understand the benefits of `PyROOT` is through a few examples.

1.1.1 Glue-ing Applications

The `PyQt` library, see http://www.riverbankcomputing.co.uk/pyqt, provides Python bindings for the Qt cross-platform GUI framework (http://www.trolltech.com). With `PyROOT` and `PyQt`, adding ROOT application layer code to a Qt GUI, becomes children play. The following example shows how a Python class can be used to have ROOT code respond to a click on a Qt widget.

```
# Glue-ing Qt and ROOT through Python
import sys, ROOT
from qt import *

theApp = QApplication( sys.argv)
box = QVBox()
box.resize(QSize(40,10).expandedTo(box.minimumSizeHint()))

class myButton(QPushButton):
    def __init__( self,label,master):
        QPushButton.__init__(self,label,master)
        self.setFont( QFont('Times',18,QFont.Bold))

    def browse(self):
        self.b = ROOT.TBrowser()

bb = myButton('browser',box)
```

```
QObject.connect( bb,SIGNAL('clicked()'),bb.browse)

theApp.setMainWidget(box)
box.show()
theApp.exec_loop()
```

When the example is run, a Qt button is displayed, and when the button is clicked, a **`TBrowser`** instance is created and will appear on the screen. `PyROOT` takes care of feeding system events to ROOT widgets, so the **`TBrowser`** instance and the button behave properly when users interact with them.

1.1.2 Access to ROOT from Python

There are several tools for scientific analysis that come with bindings that allow the use of these tools from the Python interpreter. `PyROOT` provides this for users who want to do analysis in Python with ROOT classes. The following example shows how to fill and display a ROOT histogram while working in Python. Of course, any actual analysis code may come from somewhere else through other bindings, e.g. from a C++ program.

When run, the next example will display a 1-dimensional histogram showing a Gaussian distribution. More examples like the one above are distributed with ROOT under the `$ROOTSYS/tutorials` directory.

```
# Example: displaying a ROOT histogram from Python
from ROOT import gRandom,TCanvas,TH1F

c1 = TCanvas('c1','Example',200,10,700,500)
hpx = TH1F('hpx','px',100,-4,4)

for i in xrange(25000):
    px = gRandom.Gaus()
    hpx.Fill(px)

hpx.Draw()
c1.Update()
```

1.1.3 Access to Python from ROOT

Access to Python objects from Cling is not completely fleshed out. Currently, ROOT objects and built-in types can cross the boundary between the two interpreters, but other objects are much more restricted. For example, for a Python object to cross, it has to be a class instance, and its class has to be known to Cling first (i.e. the class has to cross first, before the instance can). All other cross-coding is based on strings that are run on the Python interpreter and vise-versa.

With the ROOT v4.00/06 and later, the **`TPython`** class will be loaded automatically on use, for older editions, the `libPyROOT.so` needs to be loaded first before use. It is possible to switch between interpreters by calling **`TPython::Prompt()`** on the ROOT side, while returning with `^D` (`EOF`). State is preserved between successive switches, and string based cross calls can nest as long as shared resources are properly handled.

```
// Example: accessing the Python interpreter from ROOT
// either load PyROOT explicitly or rely on auto-loading
root[] gSystem->Load( "libPyROOT" );
root[] TPython::Exec("print1+1");
2

// create a TBrowser on the Python side, and transfer it back and forth
root[] TBrowser* b = (void*)TPython::Eval("ROOT.TBrowser()");
(class TObject*)0x8d1daa0
root[] TPython::Bind(b,"b");

// builtin variables can cross-over (after the call i==2)
root[] int i = TPython::Eval( "1+1" );
root[] i
(int)2
```

1.1.4 Installation

There are several ways of obtaining `PyROOT`, and which is best depends on your specific situation. If you work at CERN, you can use the installation available on `afs`. Otherwise, you will want to build from source, as `PyROOT` is not build by default in the binaries distributed from the ROOT project site. If you download the ROOT binaries, take care to download and install the Python distribution from http://www.python.org/ against which they were built.

1.1.4.1 Environment Settings

ROOT installations with the build of `PyROOT` enabled are available from the CERN `afs` cell `/afs/cern.ch/sw/root/<version>/<p`
To use them, simply modify your shell environment accordingly. For Unix:

```
export PATH=$ROOTSYS/bin:$PYTHONDIR/bin:$PATH
```

```
export LD_LIBRARY_PATH=$ROOTSYS/lib:$PYTHONDIR/lib:$LD_LIBRARY_PATH
```

```
export PYTHONPATH=$ROOTSYS/lib:$PYTHONPATH
```

For Windows:

```
set PATH=%ROOTSYS%/bin;%PYTHONDIR%/bin;%PATH%
```

```
set PYTHONPATH=%ROOTSYS%/bin;%PYTHONPATH%
```

where `$ROOTSYS` should be set to `/afs/cern.ch/sw/root/<version>/<platform>`, and `PYTHONDIR` to `/afs/cern.ch/sw/lcg/ex`
with `<version>` and `<platform>` as appropriate. Note that the latest version of Python is 2.4.1.

1.1.4.2 Building from Source

The standard installation instructions for building ROOT from source apply, with the addition that the build of `PyROOT` needs to be enabled at the configuration step. First, follow the instructions for obtaining and unpacking the source, and setting up the build environment.

Then, use the following command to configure the build process (of course, feel free to add any additional flags you may need):

```
$ ./configure <arch> [--with-python-incdir=<dir>][--with-python-libdir=>dir>]
```

For details on `<arch>` see the official build pages, the Python include directory should point to the directory that contains `Python.h` and the library directory should point to the directory containing `libpythonx.y.so`, where '`x`' and '`y`' are the major and minor version number, respectively. If you do not specify include and library directories explicitly, the configuration process will try the `PYTHONDIR` environment variable or, alternatively, the standard locations.

A recent distribution of Python is required: version 2.4.3 is preferred, but the older 2.2.x and 2.3.x versions suffice and are supported as well. Versions older than 2.2 are not supported and will not work. Note that one problem with 2.2 is that the shared library of the `Python` interpreter core is not build by default and the '–enable-shared' flag should thus be used when building `Python` from source. If the `Python` interpreter that is installed on your system is too old, please obtain a new version from http://www.python.org.

Once configured, you continue the build process the normal way:

```
$ make
```

```
$ make install
```

After some time, a library called `libPyROOT.so` (or `libPyROOT.dll`, on Windows) will be created in the `$ROOTSYS/lib`(`$ROOTSYS/bin on Windows`) directory and a top Python module, `ROOT.py`, will be copied into the same place. The final step is to setup the shell environment, which is similar to what is described in the chapter 'Environment Settings'. Note that the `$ROOTSYS` entries are probably already there if you followed the standard instructions, and that the `PYTHONDIR` entries should be replaced as appropriate by your choice at configuration time, or be left out if you had the configuration script pick up them up from a default location.

1.1.5 Using PyROOT

Since it is an extension module, the usage of `PyROOT` probably comes naturally if you're used to Python. In general, `PyROOT` attempts to allow working in both Python and ROOT style, and although it is succeeding, it isn't perfect: there are edges. The following sections explain in some detail what you can expect, and what you need to watch out for.

1.1.5.1 Access to ROOT Classes

Before a ROOT class can be used from Python, its dictionary needs to be loaded into the current process. Starting with ROOT version 4.00/06, this happens automatically for all classes that are declared to the auto-loading mechanism through so-called `rootmap` files. Effectively, this means that all classes in the ROOT distributions are directly available for import. For example:

```
from ROOT import TCanvas          # available at startup
c = TCanvas()

from ROOT import TLorentzVector   # triggers auto-load of libPhysics
l = TLorentzVector()
```

Although it is not recommended, a simple way of working with `PyROOT` is doing a global import:

```
from ROOT import *

c = TCanvas()
l = TLorentzVector()
```

Keeping the ROOT namespace ("`import ROOT`"), or only importing from ROOT those classes that you will actually use (see above), however, will always be cleaner and clearer:

```
import ROOT

c = ROOT.TCanvas()
l = ROOT.TLorentzVector()
```

Since it is foreseen that most people will use the simple approach anyway, the request to copy all from module ROOT will not actually result in copying all ROOT classes into the current namespace. Instead, classes will still be bound (and possibly loaded) on an as-needed basis. Note carefully how this is different from other Python (extension) modules, and what to expect if you use the normal inspection tools (such as e.g. '`dir()`'). This feature prevents the inspection tools from being swamped by an enormous amount of classes, but they can no longer be used to explore unknown parts of the system (e.g. to find out which classes are available). Furthermore, because of this approach, <`tab`>-completion will usually not be available until after the first use (and hence creation) of a class.

Access to class static functions, public data members, enums, etc. is as expected. Many more example uses of ROOT classes from Python can be found in the tutorials directory in the ROOT distribution. The recipes section contains a description on working with your own classes (see "Using Your Own Classes").

1.1.5.2 Access to STL Classes

The STL classes live in the ROOT.std namespace (or, if you prefer to get them from there, in the ROOT module directly, but doing so makes the code less clear, of course). Be careful in their use, because Python already has types called "`string`" and "`list`."

In order to understand how to get access to a templated class, think of the general template as a meta class. By instantiating the meta class with the proper parameters, you get an actual class, which can then be used to create object instances. An example usage:

```
>>> from ROOT import std
>>> v = std.vector(int)()
>>> for i in range(0,10):
...    v.push_back(i)
...
>>> for i in v:
...     print(i, end=' ')
1 2 3 4 5 6 7 8 9
>>>
>>> list(v)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

The parameters to the template instantiation can either be an actual type or value (as is used here, "int"), or a string representation of the parameters (e.g. " 'double' "), or a mixture of both (e.g. " 'TCanvas, 0' " or " 'double', 0"). The "std::vector<int>" class is one of the classes builtin by default into the Cling extension dlls. You will get a non-functional class (instances of which can still be passed around to C++) if the corresponding dictionary doesn't exist.

1.1.5.3 Access to ROOT Globals

Most globals and global functions can be imported directly from the ROOT.py module, but some common ones (most notably **gMinuit**, although that variable now exists at startup from release 5.08 onward) do not exist yet at program startup, as they exist in modules that are loaded later (e.g. through the auto-loading mechanism). An example session should make this clear:

```
>>> from ROOT import *
>>> gROOT                           # directly available
<ROOT.TROOT object at 0x399c30>
>>> gMinuit                         # library not yet loaded: not available
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: name 'gMinuit' is not defined
>>> TMinuit                         # use of TMinuit class forces auto-loading
<class '__main__.TMinuit'>
>>> gMinuit                         # now gMinuit is available
<__main__.TMinuit object at 0x1458c70>
>>> not not gMinuit                 # but it is the null pointer, until set
False
>>> g = TMinuit()
>>> not not gMinuit
True
```

It is also possible to create globals interactively, either by executing a Cling macro, or by a call to `gROOT.ProcessLine()`. These globals are made available in the same way: either use them directly after creation in 'from ROOT import *' more, or get them from the ROOT namespace after an 'import ROOT'.

As of 5.08, the behaviour of ROOT globals is the same as python globals, which is sometimes counterintuitive: since they are references, they can be changed only if done so directly through their containing module. The following session shows that in detail:

```
>>> from ROOT import *
>>> print(gDebug)
0
>>> gROOT.ProcessLine( 'gDebug = 7;' )
>>> print(gDebug)
0                                # local gDebug is unchanged
>>> gDebug = 5                   # changes _local_ reference only
>>> print(gDebug)
5                                # locally correct, but ...
>>> gROOT.ProcessLine( 'cout << gDebug << endl;' )
7                                # ... ROOT global unchanged
>>> import ROOT
>>> print(ROOT.gDebug)
7                                # still the old value (not '5')
>>> ROOT.gDebug = 3              # changes ROOT module reference
>>> gROOT.ProcessLine( 'cout << gDebug << endl;' )
3                                # ROOT global properly changed
>>>
```

The above is another good reason to prefer 'import ROOT' over 'from ROOT import *'.

1.1.5.4 Access to Python

The access to Python from Cling goes through the **TPython** class, or directly if a Python object or class has crossed the border. The **TPython** class, which looks approximately like this:

```
class TPython {

public:
   // load a Python script as if it were a macro
```

```
    static void LoadMacro(const char* name);

   // execute a Python statement (e.g. "import ROOT")
    static void Exec(const char* cmd);

   // evaluate a Python expression (e.g. "1+1")
    static const TPyReturn& Eval(const char* expr);

   // bind a ROOT object with, at the Python side, the name "label"
    static bool Bind(TObject* obj,const char* label);

   // enter an interactive Python session (exit with ^D)
    static void Prompt();
};
```

`LoadMacro(const char* name)` - the argument is a name of a Python file that is to be executed (`'execfile'`), after which any new classes are automatically made available to Cling. Since it is non-selective, use with care.

`ExecScript(const char* name,int argc=0,const char** argv=0)` - the argument is a name of a python file that is to be executed ('execfile') in a private namespace to minimize side-effects. Optionally, you can add CLI-style arguments which are handed to the script through 'sys.argv' in the normal way.

`Exec(const char* cmd)`- the argument is a string of Python code that is executed as a statement. There is no return value, but an error message will be printed if there are problems such as syntax errors.

`Eval(const char* expr)`- the argument is a string of Python code that is evaluated as an expression. The result of the expression is returned, if it is either a builtin type (int, long, float, double, and `const char*` are supported), a Python type that can cross, or a ROOT type. If a ROOT type is returned, an explicit cast to void* is needed to assign the return value to a local pointer (which may have a different type), whereas builtin types will be cast implicitly, if possible, to the type of the local variable to which they are assigned.

`Bind(TObject* obj,const char* label)` - transfer a ROOT object from the Cling to the Python interpreter, where it will be referenced with a variable called "`label`".

`Prompt()` - Transfer the interactive prompt to Python.

With the ROOT v4.00/06 and later, the **`TPython`** class will be loaded automatically on use, for older editions, the `libPyROOT.so` needs to be loaded first with `gSystem->Load()` before use. Refer back to the other example of the use of **`TPython`** that was given in "Access to Python from ROOT".

To show in detail how Python access can be used, an example Python module is needed, as follows:

```
print('creating class MyPyClass ... ')
class MyPyClass:
    def __init__(self):
        print('in MyPyClass.__init__')
        self._browser = None
    def gime(self, what):
        return what
```

This module can now be loaded into a Cling session, the class used to instantiate objects, and their member functions called for showing how different types can cross:

```
root[] TPython::LoadMacro("MyPyClass.py");
creating class MyPyClass ...
root[] MyPyClass m;
in MyPyClass.__init__
root[] char* s = m.gime("aap");
root[] s
(char* 0x41ee7754)"aap"
```

Note that the `LoadMacro()` call makes the class automatically available, such that it can be used directly. Otherwise, a `gROOT->GetClass()` call is required first.

1.1.5.5 Callbacks

The simplest way of setting a callback to Python from Cling, e.g. for a button, is by providing the execution string. See for example `tutorials/pyroot/demo.py` that comes with the ROOT installation:

```
# [..]
bar = ROOT.TControlBar('vertical','Demos')
bar.AddButton('Help on Demos',r'TPython::Exec("execfile('demoshelp.py')");','Click Here For Help on Runnin
bar.AddButton('browser',r'TPython::Exec("b = Tbrowser()");','Start the ROOT browser')
# [..]
```

Here, the callback is a string that will be interpreted by Cling to call `TPython::Exec()`, which will, in turn, interpret and execute the string given to it. Note the use of raw strings (the '`r`' in front of the second argument string), in order to remove the need of escaping the backslashes.

1.1.5.6 Cling Commands

In interactive mode, the Python exception hook is used to mimic some of the Cling commands available. These are: `.q`, `.!`, `.x`, `.L`, `.cd`, `.ls`, `.pwd`, `.?` and `.help`. Note that `.x` translates to Python '`execfile()`' and thus accepts only Python files, not Cling macros.

1.1.6 Memory Handling

The Python interpreter handles memory for the user by employing reference counting and a garbage collector (for new-style objects, which includes `PyROOT` objects). In C++, however, memory handling is done either by hand, or by an application specific, customized mechanism (as is done in ROOT). Although `PyROOT` is made aware of ROOT memory management, there are still a few boundary conditions that need to be dealt with by hand. Also, the heuristics that `PyROOT` employs to deal with memory management are not infallible. An understanding in some detail of the choices that are made is thus important.

1.1.6.1 Automatic Memory Management

There are two global policies that can be set: heuristics and strict. By default, the heuristic policy is used, in which the following rules are observed:

- A ROOT object created on the Python interpreter side is owned by Python and will be deleted once the last Python reference to it goes away. If, however, such an object is passed by non-const address as a parameter to a C++ function (with the exception of the use as "self" to a member function), ownership is relinquished.
- A ROOT object coming from a ROOT call is not owned, but before it passes to the Python interpreter, its "must cleanup" bit is set if its type is a class derived from `TObject`. When the object goes out of scope on the C++ side, the Python object will change type into an object that largely behaves like None.

The strict policy differs in that it will never relinquish ownership when passing an object as a parameter to a function. It is then up to the developer to prevent double deletes. Choosing one or the other policy is done by:

```
ROOT.SetMemoryPolicy( ROOT.kMemoryStrict )
```

for the strict policy, or for the heuristic policy:

```
ROOT.SetMemoryPolicy( ROOT.kMemoryHeuristics )
```

Care must be taken in the case of graphic objects: when drawn on the current pad, a reference to the graphics is kept that `PyROOT` isn't currently aware of, and it is up to the developer to keep at lease one Python reference alive. See `$ROOTSYS/tutorials/pyroot/zdemo.py` (available in the latest release) for an example. Alternatively, one can tell python to give up ownership for individual instances:

```
o = ROOT.TObject()
ROOT.SetOwnership( o, False )      # True to own, False to release
```

1.1.6.2 Memory Management by Hand

If needed, you can explicitly destroy a ROOT object that you own through its associated `TClass`:

```
myobject.IsA().Destructor(myobject)
```

which will send out the deletion notification to the system (thus you do not need to care anymore at this point about Python reference counting, the object will go, even if it's reference count it non-zero), and free the memory.

1.1.7 Performance

The performance of PyROOT when programming with ROOT in Python is similar to that of Cling. Differences occur mainly because of differences in the respective languages: C++ is much harder to parse, but once parsed, it is much easier to optimize. Consequently, individual calls to ROOT are typically faster from PyROOT, whereas loops are typically slower.

When programming in Python, the modus operandi is to consider performance generally "good enough" on the outset, and when it turns out that, it is not good enough; the performance critical part is converted into C/C++ in an extension module. The school of thought where pre-mature optimization is the root of all evil should find this way of working very satisfying. In addition, if you look at their history, you will see that many of the standard Python modules have followed this path.

Your code should always make maximum use of ROOT facilities; such that most of the time is spending in compiled code. This goes even for very simple things: e.g. do not compute invariant masses in Python, use **TLorentzVector** instead. Moreover, before you start optimizing, make sure that you have run a profiler to find out where the bottlenecks are. Some performance, without cost in terms of programmer effort, may be gained by using psyco, see the next link: http://psyco.sourceforge.net, a Python just in time compiler (JIT). Note, however, that psyco is limited to Intel i386 CPUs. Since psyco optimizes Python, not PyROOT calls; it generally does not improve performance that much if most of your code consists of ROOT API calls. Mathematical computations in Python, on the other hand, benefit a lot.

Every call to a Python member function results in a lookup of that member function and an association of this method with 'self'. Furthermore, a temporary object is created during this process that is discarded after the method call. In inner loops, it may be worth your while (up to 30%), to short-cut this process by looking up and binding the method before the loop, and discarding it afterwards. Here is an example:

```
hpx = TH1F('hpx','px',100,-4,4)
hpxFill = hpx.Fill                # cache bound method
for i in xrange(25000):
    px = gRandom.Gaus()
hpxFill(px)                    # use bound method: no lookup needed
del hpxFill                       # done with cached method
```

Note that if you do not discard the bound method, a reference to the histogram will remain outstanding, and it will not be deleted when it should be. It is therefore important to delete the method when you're done with it.

1.1.8 Use of Python Functions

It is possible to mix Python functions with ROOT and perform such operations as plotting and fitting of histograms with them. In all cases, the procedure consists of instantiating a ROOT **TF1**, **TF2**, or **TF3** with the Python function and working with that ROOT object. There are some memory issues, so it is for example not yet possible to delete a **TF1** instance and then create another one with the same name. In addition, the Python function, once used for instantiating the **TF1**, is never deleted.

Instead of a Python function, you can also use callable instances (e.g., an instance of a class that has implemented the `__call__` member function). The signature of the Python callable should provide for one or two arrays. The first array, which must always be present, shall contain the x, y, z, and t values for the call. The second array, which is optional and its size depends on the number given to the **TF1** constructor, contains the values that parameterize the function. For more details, see the **TF1** documentation and the examples below.

1.1.8.1 Plotting Python Function

This is an example of a parameter less Python function that is plotted on a default canvas:

```
from ROOT import TF1, TCanvas

def identity( x ):
    return x[0]

# create an identity function
f = TF1('pyf1', identity, -1., 1.)

# plot the function
c = TCanvas()
f.Draw()
```

Because no number of parameters is given to the **TF1** constructor, '0' (the default) is assumed. This way, the 'identity' function need not handle a second argument, which would normally be used to pass the function parameters. Note that the argument'x' is an array of size 4. The following is an example of a parameterized Python callable instance that is plotted on a default canvas:

```
from ROOT import TF1, TCanvas

class Linear:
    def __call__( self, x, par ):
        return par[0] + x[0]*par[1]

# create a linear function with offset 5, and pitch 2
f = TF1('pyf2',Linear(),-1.,1.,2)
f.SetParameters(5.,2.)

# plot the function
c = TCanvas()
f.Draw()
```

Note that this time the constructor is told that there are two parameters, and note in particular how these parameters are set. It is, of course, also possible (and preferable if you only use the function for plotting) to keep the parameters as data members of the callable instance and use and set them directly from Python.

1.1.8.2 Fitting Histograms

Fitting a histogram with a Python function is no more difficult than plotting: instantiate a **TF1** with the Python callable and supply that **TF1** as a parameter to the `Fit()` member function of the histogram. After the fit, you can retrieve the fit parameters from the **TF1** instance. For example:

```
from ROOT import TF1, TH1F, TCanvas

class Linear:
    def __call__( self, x, par ):
        return par[0] + x[0]*par[1]

# create a linear function for fitting
f = TF1('pyf3',Linear(),-1.,1.,2)

# create and fill a histogram
h = TH1F('h','test',100,-1.,1.)
f2 = TF1('cf2','6.+x*4.5',-1.,1.)
h.FillRandom('cf2',10000)

# fit the histo with the python 'linear' function
h.Fit(f)

# print results
par = f.GetParameters()
print('fit results: const =', par[0], ',pitch =', par[1])
```

1.1.9 Working with Trees

Next to making histograms, working with trees is probably the most common part of any analysis. The **TTree** implementation uses pointers and dedicated buffers to reduce the memory usage and to speed up access. Consequently, mapping **TTree** functionality to Python is not straightforward, and most of the following features are implemented in ROOT release 4.01/04 and later only, whereas you will need 5.02 if you require all of them.

1.1.9.1 Accessing an Existing Tree

Let us assume that you have a file containing **TTrees**, **TChains**, or **TNtuples** and want to read the contents for use in your analysis code. This is commonly the case when you work with the result of the reconstruction software of your experiment (e.g. the combined ntuple in ATLAS). The following example code outlines the main steps (you can run it on the result of the `tree1.C` macro):

```
from ROOT import TFile

# open the file
myfile = TFile('tree1.root')

# retrieve the ntuple of interest
mychain = myfile.Get('t1')
entries = mychain.GetEntriesFast()

for jentry in xrange(entries):
    # get the next tree in the chain and verify
    ientry = mychain.LoadTree(jentry)
    if ientry < 0:
        break

    # copy next entry into memory and verify
    nb = mychain.GetEntry(jentry)
    if nb<=0:
        continue

    # use the values directly from the tree
    nEvent = int(mychain.ev)
    if nEvent<0:
        continue

    print(mychain.pz, '=', mychain.px*mychain.px, '+', mychain.py*mychain.py)
```

Access to arrays works the same way as access to single value tree elements, where the size of the array is determined by the number of values actually read from the file. For example:

```
# loop over array tree element
for d in mychain.mydoubles:
    print(d)

# direct access into an array tree element
i5 = mychain.myints[5]
```

1.1.9.2 Writing a Tree

Writing a ROOT **`TTree`** in a Python session is a little convoluted, if only because you will need a C++ class to make sure that data members can be mapped, unless you are working with built-in types. Here is an example for working with the latter only:

```
from ROOT import TFile, TTree
from array import array

h = TH1F('h1','test',100,-10.,10.)
f = TFile('test.root','recreate')
t = TTree('t1','tree with histos')
maxn = 10
n = array('i',[0])
d = array('f',maxn*[0.])
t.Branch('mynum',n,'mynum/I')
t.Branch('myval',d,'myval[mynum]/F')

for i in range(25):
    n[0] = min(i,maxn)
    for j in range(n[0]):
        d[j] = i*0.1+j
        t.Fill()
```

```
f.Write()
f.Close()
```

The use of arrays is needed, because the pointer to the address of the object that is used for filling must be given to the **`TTree::Branch()`** call, even though the formal argument is declared a ’`void*`'. In the case of ROOT objects, similar pointer manipulation is unnecessary, because the full type information is available, and `TTree::Branch()` has been Pythonized to take care of the call details. However, data members of such objects that are of built-in types, still require something extra since they are normally translated to Python primitive types on access and hence their address cannot be taken. For that purpose, there is the `AddressOf()` function. As an example:

```
from ROOT import TFile, TTree
from ROOT import gROOT, AddressOf

gROOT.ProcessLine(
"struct MyStruct { Int_t fMyInt1; Int_t fMyInt2; Int_t fMyInt3; Char_t fMyCode[4]; };" );

from ROOT import MyStruct
mystruct = MyStruct()
f = TFile('mytree.root','RECREATE')
tree = TTree('T','Just A Tree')
tree.Branch('myints',mystruct,'MyInt1/I:MyInt2:MyInt3')
tree.Branch('mycode',AddressOf(mystruct,'fMyCode'),'MyCode/C')
for i in range(0,10):
    mystruct.fMyInt1 = i
    mystruct.fMyInt2 = i*i
    mystruct.fMyInt3 = i*i*i
    mystruct.fMyCode = "%03d" % i      # note string assignment

    tree.Fill()

f.Write()
f.Close()
```

The C++ class is defined through the `gROOT.ProcessLine()` call, and note how the `AddressOf()` function is used for data members of built-in type. Most of the above is for ROOT version 5.02 and later only. For older releases, and without further support, here is an example as to how you can get hold of a pointer-to-pointer to a ROOT object:

```
h = TH1F()
addressofobject = array('i',[h.IsA().DynamicCast(h.IsA(),h)])
```

1.1.10 Using Your Own Classes

A user’s own classes can be accessed after loading, either directly or indirectly, the library that contains the dictionary. One easy way of obtaining such a library, is by using ACLiC:

```
$ cat MyClass.C
class MyClass {
public:

MyClass(int value = 0) {
m_value = value;
}

void SetValue(int value) {
m_value = value;
}

int GetValue() {
return m_value;
}

private:
    int m_value;
};
```

```
$ echo .L MyClass.C+ | root.exe -b
[...]
Info in <TUnixSystem::ACLiC>: creating shared library [..]/./MyClass_C.so
$
```

Then you can use it, for example, like so:

```
from ROOT import gSystem

# load library with MyClass dictionary
gSystem.Load('MyClass_C')

# get MyClass from ROOT
from ROOT import MyClass
# use MyClass
m = MyClass(42)
print(m.GetValue())
```

You can also load a macro directly, but if you do not use ACLiC, you will be restricted to use the default constructor of your class, which is otherwise fully functional. For example:

```
from ROOT import gROOT

# load MyClass definition macro (append '+' to use ACLiC)
gROOT.LoadMacro('MyClass.C')

# get MyClass from ROOT
from ROOT import MyClass

# use MyClass
m = MyClass()
m.SetValue(42)
print(m.GetValue())
```