A New PyROOT: Modern, Interoperable and More Pythonic

Massimiliano Galli[1,3,*], *Enric* Tejedor[1,**], and *Stefan* Wunsch[1,2,***]

[1]CERN
[2]Karlsruhe Institute of Technology (KIT)
[3]Università di Bologna

Abstract. Python is nowadays one of the most widely-used languages for data science. Its rich ecosystem of libraries together with its simplicity and readability are behind its popularity. HEP is also embracing that trend, often using Python as an interface language to access C++ libraries for the sake of performance. PyROOT, the Python bindings of the ROOT software toolkit, plays a key role here, since it allows to automatically and dynamically invoke C++ code from Python without the generation of any static wrappers beforehand. In that sense, this paper presents the efforts to create a new PyROOT with three main qualities: modern, able to exploit the latest C++ features from Python; pythonic, providing Python syntax to use C++ classes; interoperable, able to interact with the most important libraries of the Python data science toolset.

1 Introduction

Python is currently at the top of the most widely-used languages and tools for data science [1]. The language was born with a philosophy that emphasizes code readability and programming productivity, which makes it attractive and more accessible to non-expert programmers. Moreover, there are a number of powerful and fairly mature Python libraries for scientific computing, data manipulation, analysis and visualisation, some prominent examples being NumPy [2], pandas [3], SciPy [4] and Matplotlib [5].

Although the field of High-Energy Physics is still dominated by C++, Python is steadily gaining adoption for data analysis [6] and new libraries are proliferating [7]. In most cases, Python is used as an interface language to C++, which allows to keep the simplicity of the Python programming language without sacrificing performance. On the other hand, interoperability is a primary goal: any Python tool for analysis in HEP must be able to integrate itself into the already existing (and rich) Python data science ecosystem, mainly by building bridges with other tools and understanding the same data formats.

The ROOT software toolkit [8] is meant to play a key role in the interaction of HEP programs with the Python world. Although it is mostly written in C++, ROOT provides bindings for Python, called PyROOT, which are unique: they allow to dynamically access C++ functionality from Python (classes, functions) without any prior generation of static bindings. This is possible thanks to the ROOT C++ interpreter and type system, which can

[*]e-mail: massimiliano.galli@cern.ch
[**]e-mail: etejedor@cern.ch
[***]e-mail: stefan.wunsch@cern.ch

provide information about / instantiate C++ entities as requested by PyROOT at execution time. In practice, this means that a user can call into any ROOT C++ library via PyROOT, or even load a user library and invoke it too, without any wrapper generation beforehand.

This paper describes how the powerful PyROOT bindings are now going through a process of redesign and modernisation in order to better support the needs of HEP analysers. First, PyROOT is being redesigned to operate on top of cppyy [9], a set of libraries that offer automatic Python-C++ bindings and include support for new C++ features. Second, PyROOT is being made more pythonic by adding support for Python syntax in the manipulation of C++ classes (the so-called pythonisations). Third, PyROOT is improving its interoperability with Python data science libraries by making it possible to read ROOT data into NumPy and pandas. Last, as a response to a common request from our users, the build system of PyROOT is being modified to allow the generation of PyROOT libraries for more than one Python version, without having to rebuild the entire ROOT.

The structure of this paper is as follows. Section 2 describes how the redesign of the new PyROOT on top of the cppyy libraries. Section 3 explains how pythonisations are going to make the new PyROOT more pythonic. Section 4 presents the efforts to make PyROOT interoperable with the Python data science ecosystem. Section 5 introduces the new support in PyROOT for multi-Python builds. Finally, Section 6 discusses some conclusions.

2 New Design on Top of Modern Cppyy

After its standardisation in 1998 and until 2011, C++ had only one minor revision. However, since then, revisions are more frequent and new features appear at a higher pace. This means that we are now in busier times for Python-C++ bindings, which need to keep up and support new C++ syntax and functionality being added to the language.

Therefore, it is of utmost importance for PyROOT to be up-to-date and support modern C++ features. For that purpose, PyROOT is being redesigned to work on top of cppyy, a set of libraries for automatic Python-C++ bindings generation, which offers access to the latest C++ features.

Figure 1 illustrates the redesign of PyROOT, which now acts as the upper layer or entry point for the user application. It also contains the pythonisations for ROOT classes (more information in Section 3). Below, the cppyy layer provides the automatic bindings and pythonisations for classes of the standard template library of C++. Cppyy operates on top of the ROOT type system and Cling, the C++ interpreter, which make it possible to dynamically obtain information about C++ entities, just-in-time compile code and execute code.

As an example of how the use of cppyy can be advantageous for PyROOT, Figure 2 shows a case of modern C++ syntax that PyROOT now supports thanks to cppyy: variadic templates. This is just one example, but there are many others, such as improved template support, better overload resolution or support for r-value reference parameters and lambda functions.

3 Pythonisations for C++ Classes

The cppyy automatic bindings described in Section 2 provide the core functionality of PyROOT, but they are not enough to offer a fully Python-friendly experience to the user. Sometimes it is not sufficient to be able to invoke some C++ code from Python: it is also important to invoke it in the most pythonic way possible. Python possesses a set of features / artifacts that are characteristic of the language, and those should be made available to the user even if it is C++ code running underneath. Otherwise, the user can feel like they are just writing C++ code in Python. This is why pythonisations are necessary.

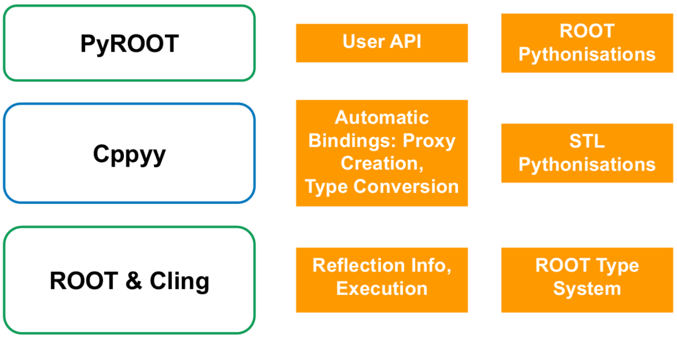
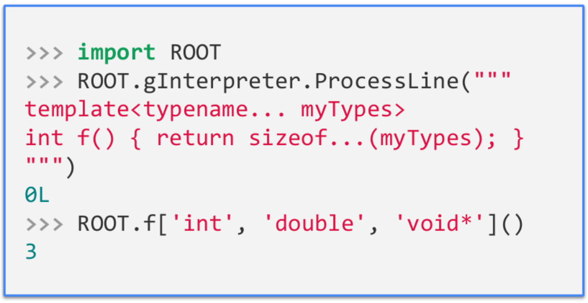


Figure 1: Design of the new PyROOT on top of cppyy.



```
>>> import ROOT
>>> ROOT.gInterpreter.ProcessLine("""
template<typename... myTypes>
int f() { return sizeof...(myTypes); }
""")
0L
>>> ROOT.f['int', 'double', 'void*']()
3
```

Figure 2: This example dynamically defines a C++ function template with variadic template syntax via PyROOT; after that it instantiates the template with three types and calls the resulting function.

A pythonisation is just some logic that is dynamically injected into a Python entity, e.g. a class, which modifies the behaviour of that entity to make it more pythonic or easier to use. An example can be found in Figure 3, which illustrates two ways of accessing a `TTree` that is stored in a ROOT file (in short, `TTree` is the class that represents a columnar dataset in ROOT). In 3a), the `TTree` is retrieved by calling the `GetObject` method, just like it would be done in C++. However, in 3b), the `TTree` is obtained via the attribute syntax of Python. It is important to note how this is not provided by cppyy out-of-the-box, but by a pythonisation: `mytree` is not an attribute of `myfile`, but there is a pythonisation that complements the lookup of attributes in the file by executing `GetObject` under the hood.

The new PyROOT incorporates many pythonisations for ROOT classes, not only for `TTree` as in the example, but also for other key ones in ROOT. Such pythonisations allow for a more pythonic use of ROOT classes, e.g. by making them iterable in Python, customising the attribute access, injecting a new subscript operator or overriding mathematic operators.

It is worth pointing out that such pythonisations are applied lazily: only when a given class is used from the application, the pythonisations for that class are executed.

```
myfile.GetObject('mytree')
```

(a)

```
myfile.mytree
```

(b)

Figure 3: Two ways of getting the TTree stored in a ROOT file in PyROOT: 3a) the C++-like way, 3b) the pythonic way.

3.1 User Pythonisations

In addition to providing pythonisations for ROOT classes, PyROOT will allow soon to define pythonisations for user classes too. Such pythonisations will also be lazy and only run when the application references the corresponding class for the first time.

The envisaged API for the user to define their own pythonisations is depicted in Figure 4. In order to pythonise a given class, the user needs to provide a function that is decorated with the `pythonization` decorator, which specifies which class is being pythonised. Such function will receive one parameter, the class to be pythonised, in which new behaviour can be injected as desired.

```
@pythonization('MyCppClass')
def my_pythonizor_function(klass):
    # Inject new behaviour in the class
    klass.__iter__ = ...
```

Figure 4: Function that pythonises the `MyCppClass` user C++ class.

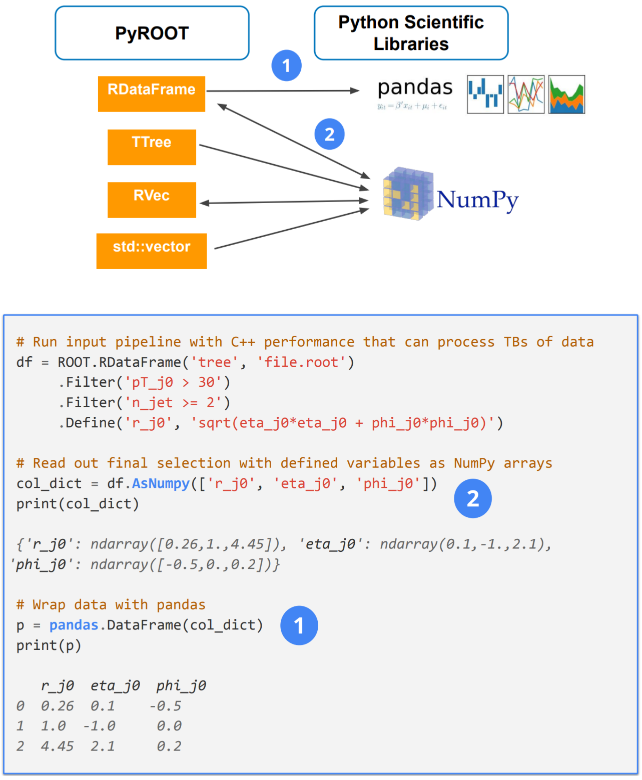
4 Interoperability with the Python Ecosystem

The Python ecosystem for scientific applications is centered around the SciPy [4] packages, which are using multidimensional arrays implemented by NumPy [2] as interface for the data. This data interface is adopted widely in the scientific community for example in the machine-learning ecosystem with scikit-learn [10] or TensorFlow [11]. Therefore, interoperability in the scientific Python ecosystem is primarily defined by providing data processed by ROOT in C++ in Python as NumPy arrays and also vice versa.

PyROOT implements the adoption of data in C++ containers such as `std::vector` by providing pythonizations, which attach the NumPy array interface to these objects in Python. The array interface states the data type, shape and memory address so that NumPy can build a zero-copy view on the data. Going from data owned by NumPy in Python to C++ is also supported by the ROOT container `ROOT::RVec`, which follows the interface of a `std::vector` but allows to adopt memory. Using this mechanism, PyROOT is able to move data originating from Python libraries to C++ without copying or error-prone raw pointer interfaces.

On a higher level, interoperability is also provided for larger data structures such as dataframes. On the ROOT side, the `ROOT::RDataFrame` [12] facility provides efficient data processing of columnar datasets. To provide the dataset in Python, the

RDataFrame.AsNumpy method allows to access the data as a dictionary with the column names as keys and the data as NumPy arrays. This data structure can also be used to construct directly a pandas [3] dataframe for further processing of the columnar dataset in the Python ecosystem. A graphical overview over the described features is shown in Figure 5.



```
# Run input pipeline with C++ performance that can process TBs of data
df = ROOT.RDataFrame('tree', 'file.root')
      .Filter('pT_j0 > 30')
      .Filter('n_jet >= 2')
      .Define('r_j0', 'sqrt(eta_j0*eta_j0 + phi_j0*phi_j0)')

# Read out final selection with defined variables as NumPy arrays
col_dict = df.AsNumpy(['r_j0', 'eta_j0', 'phi_j0'])
print(col_dict)

{'r_j0': ndarray([0.26,1.,4.45]), 'eta_j0': ndarray(0.1,-1.,2.1),
'phi_j0': ndarray([-0.5,0.,0.2])}

# Wrap data with pandas
p = pandas.DataFrame(col_dict)
print(p)

   r_j0  eta_j0  phi_j0
0  0.26  0.1     -0.5
1  1.0  -1.0      0.0
2  4.45  2.1      0.2
```

Figure 5: Overview over the interoperability features provided by PyROOT.

5 Building the New PyROOT

Despite Python2 being deprecated starting from 2020 [13], there is still a significant amount of HEP code written in Python2, and the transition to Python3 is expected to take some more time. For this reason the possibility to build PyROOT libraries for multiple Python versions without rebuild the entire ROOT was introduced.

When building ROOT, the user can now specify both a Python2_EXECUTABLE and a Python3_EXECUTABLE parameters to CMake, which point to the location of the Python2 and Python3 binaries to be used, respectively. If such parameters are not specified, the system directories will be searched for Python installations. If both Python2 and Python3 installations have been specified/found, the PyROOT libraries will be generated for both versions.

Concerning the installation, the PyROOT libraries will be installed by default in the same directory as the rest of ROOT libraries. However, the future work includes adding an option to install the PyROOT libraries in Python's site-packages directory, just like any other Python package, so that Python can find them with no need to manually update PYTHONPATH.

6 Conclusions

Due to the growing importance of Python for data analysis in HEP, ROOT is going through a redesign and modernisation of its Python bindings, PyROOT, to better support the use cases of physicists. The use of the cppyy libraries to support modern C++ features, the addition of pythonisations to make it more attractive and easier to use for Python programmers, the improved interoperability with the Python data science ecosystem and the possibility to build PyROOT for multiple Python versions are currently the main areas of work.

It is foreseen that the new PyROOT will become the default during 2020, most probably for the ROOT 6.22 release. The work on the aforementioned areas will continue after that milestone, with the objective of keeping it up to date with the requirements of the HEP Python community.

References

[1] *Python leads the 11 top data science, machine learning platforms: Trends and analysis* (2019), `https://www.kdnuggets.com/2019/05/poll-top-data-science-machine-learning-platforms.html`

[2] T.E. Oliphant, *A guide to NumPy*, Vol. 1 (Trelgol Publishing USA, 2006)

[3] W. McKinney, *Data Structures for Statistical Computing in Python*, in *Proceedings of the 9th Python in Science Conference*, edited by S. van der Walt, J. Millman (2010), pp. 51 – 56

[4] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright et al., Nature Methods (2020)

[5] J.D. Hunter, Computing in Science & Engineering **9**, 90 (2007)

[6] *User feedback from LHCb* (2018), `https://indico.cern.ch/event/697389/contributions/3114285/attachments/1712589/2761508/lhcb_user_survey.pdf`

[7] *Scikit-HEP*, https://scikit-hep.org/

[8] R. Brun, F. Rademakers, *ROOT Object Oriented Data Analysis Framework*, in *New computing techniques in physics research V. Proceedings, 5th International Workshop, AIHENP '96, Lausanne, Switzerland* (1996)

[9] W.T.L.P. Lavrijsen, A. Dutta, *High-performance Python-C++ Bindings with PyPy and Cling*, in *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing* (IEEE Press, Piscataway, NJ, USA, 2016), PyHPC '16, pp. 27–35, ISBN 978-1-5090-5220-2, `https://ieeexplore.ieee.org/abstract/document/7836841/`

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg et al., Journal of Machine Learning Research **12**, 2825 (2011)

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., *Tensorflow: A system for large-scale machine learning*, in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 265–283

[12] E. Guiraud, A. Naumann, D. Piparo, *TDataFrame: functional chains for ROOT data analyses* (2017), `https://doi.org/10.5281/zenodo.260230`

[13] *Sunsetting-Python2*, https://www.python.org/doc/sunset-python-2/