Guide

June 13, 2022

0.1 # Opening a File

To open a .ROOT file, the function `uproot.open("path/to/file.root")` is used. For example,

[1]:
```
import uproot

file1 = uproot.open("PyHEP_UPROOTandAWKWARD/data/HiggsZZ4mu.root")
file1
```

[1]:
```
<ReadOnlyDirectory '/' at 0x02927e25c0a0>
```

However, it is safer to use the `with` statement in order to ensure that the files close when the program ends.

[22]:
```
with uproot.open("PyHEP_UPROOTandAWKWARD/data/HiggsZZ4mu.root") as file2:
    print(type(file2))
```

```
<class 'uproot.reading.ReadOnlyDirectory'>
```

The file path can either be a local file path or an URL,

[23]:
```
with uproot.open("https://scikit-hep.org/uproot3/examples/nesteddirs.root") as␣
↪file3:
    print(type(file3))
```

```
<class 'uproot.reading.ReadOnlyDirectory'>
```

`uproot.open` have other parameters such as `num_workers` or `object_cache` (more about them, and other parameters, here). The default parameters attempt to optimze everything but better performance can be obtain by tuning the parameters.

0.2 Navigating ROOT file

0.3 # Finding objects in a file

The object returned by `uproot.open` represents a TDirectory inside the file (a Mapping Python object). To get a list of contents use the method `.keys()`.

[16]:
```
file1.keys()
```

[16]:
```
['Events;5']
```

[26]:
```
file2 = uproot.open("https://scikit-hep.org/uproot3/examples/nesteddirs.root")
file2.keys()
```

[26]:
```
['one;1',
 'one/two;1',
 'one/two/tree;1',
 'one/tree;1',
 'three;1',
 'three/tree;1']
```

To extract an item you use square brackets, omiting the cycle number (everything after ;), as follows,

[27]:
```
file2["one"]
```

[27]:
```
<ReadOnlyDirectory '/one' at 0x016f1e76d9a0>
```

[28]:
```
file2["one"]["two"]
```

[28]:
```
<ReadOnlyDirectory '/one/two' at 0x016f217be9a0>
```

[30]:
```
file2["one"]["two"]["tree"]
```

[30]:
```
<TTree 'tree' (20 branches) at 0x016f2178f250>
```

Or the separations can be showed as slashes,

[31]:
```
file2["one/two/tree"]
```

[31]:
```
<TTree 'tree' (20 branches) at 0x016f2178f250>
```

Data isn't read from the disk until they are explicitly requested with squre brackets. Alternativly, you can use `.classnames()` to get the names of classes without reading the objects first.

[32]:
```
file2.classnames()
```

[32]:
```
{'one;1': 'TDirectory',
 'one/two;1': 'TDirectory',
 'one/two/tree;1': 'TTree',
 'one/tree;1': 'TTree',
 'three;1': 'TDirectory',
 'three/tree;1': 'TTree'}
```

As a shortcut, you can open a file and jump straight to the object by separating the file path and object path with a colon.

[4]:
```
events = uproot.open("https://scikit-hep.org/uproot3/examples/Zmumu.root:
↪events")
```

```
events
```

[4]:

```
<TTree 'events' (20 branches) at 0x0277043da850>
```

1 Extracting histograms from a file

Uproot can read most types of objects but only a few of them have been overloaded with specialized behaviors. Classes unknown to Uproot can be accessed through their members. To se the members you can use `.all_members`.

[36]:

```
file = uproot.open("https://scikit-hep.org/uproot3/examples/hepdata-example.
↪root")
file.classnames()
```

[36]:

```
{'hpx;1': 'TH1F',
 'hpxpy;1': 'TH2F',
 'hprof;1': 'TProfile',
 'ntuple;1': 'TNtuple'}
```

[38]:

```
file["hpx"].all_members
```

[38]:

```
{'@fUniqueID': 0,
 '@fBits': 50331656,
 'fName': 'hpx',
 'fTitle': 'This is the px distribution',
 'fLineColor': 602,
 'fLineStyle': 1,
 'fLineWidth': 1,
 'fFillColor': 0,
 'fFillStyle': 1001,
 'fMarkerColor': 1,
 'fMarkerStyle': 1,
 'fMarkerSize': 1.0,
 'fNcells': 102,
 'fXaxis': <TAxis (version 9) at 0x016f21975ac0>,
 'fYaxis': <TAxis (version 9) at 0x016f21975850>,
 'fZaxis': <TAxis (version 9) at 0x016f219753d0>,
 'fBarOffset': 0,
 'fBarWidth': 1000,
 'fEntries': 75000.0,
 'fTsumw': 74994.0,
 'fTsumw2': 74994.0,
 'fTsumwx': -97.16475860591163,
 'fTsumwx2': 75251.86518025988,
 'fMaximum': -1111.0,
 'fMinimum': -1111.0,
```

```
 'fNormFactor': 0.0,
 'fContour': <TArrayD [] at 0x016f219755e0>,
 'fSumw2': <TArrayD [] at 0x016f21975610>,
 'fOption': <TString '' at 0x016f21965430>,
 'fFunctions': <TList of 1 items at 0x016f21975eb0>,
 'fBufferSize': 0,
 'fBuffer': array([], dtype=float64),
 'fBinStatErrOpt': 0,
 'fN': 102}
```

To see an specific one, you can use `.member("NAME")`.

[41]:
```
file["hpx"].member("fName")
```

[41]:
```
'hpx'
```

Some classes, like uproot.behaviors.TH1.TH1, uproot.behaviors.TProfile.TProfile, and uproot.behaviors.TH2.TH2, have high-level "behaviors" defined in uproot.behaviors to make them easier to use.

Histograms have edges, values and errors mehods to extract the content directly to NumPy arrays. To see you use file["name"].axis().edges(), file["name"].values and file["name"].errors() respectivley.

Uproot (since it's an io library) doesn't have methods for plotting/manipulating histogrmas. Instead, it has methods to export them to other libraries such as NumPy, Boost and Hist.

[42]:
```
file["hpxpy"].to_numpy()
```

[42]:
```
(array([[0., 0., 0., …, 0., 0., 0.],
        [0., 0., 0., …, 0., 0., 0.],
        [0., 0., 0., …, 0., 0., 0.],
        …,
        [0., 0., 0., …, 0., 0., 0.],
        [0., 0., 0., …, 0., 0., 0.],
        [0., 0., 0., …, 0., 0., 0.]], dtype=float32),
 array([-4. , -3.8, -3.6, -3.4, -3.2, -3. , -2.8, -2.6, -2.4, -2.2, -2. ,
        -1.8, -1.6, -1.4, -1.2, -1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,
         0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8,  2. ,  2.2,  2.4,
         2.6,  2.8,  3. ,  3.2,  3.4,  3.6,  3.8,  4. ]),
 array([-4. , -3.8, -3.6, -3.4, -3.2, -3. , -2.8, -2.6, -2.4, -2.2, -2. ,
        -1.8, -1.6, -1.4, -1.2, -1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,
         0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8,  2. ,  2.2,  2.4,
         2.6,  2.8,  3. ,  3.2,  3.4,  3.6,  3.8,  4. ]))
```

[43]:
```
file["hpxpy"].to_boost()
```

[43]:
```
Histogram(
  Regular(40, -4, 4),
  Regular(40, -4, 4),
```

```
  storage=Double()) # Sum: 74985.0 (75000.0 with flow)
```

[44]:
```
file["hpxpy"].to_hist()
```

[44]:
```
Hist(
  Regular(40, -4, 4, name='xaxis', label='xaxis'),
  Regular(40, -4, 4, name='yaxis', label='yaxis'),
  storage=Double()) # Sum: 74985.0 (75000.0 with flow)
```

Inspecting a TBranches of a TTree

`uproot.TTree`, which contains TBranches or other nested TTrees, is another type of Mapping objects within Uproot. Like the command `.classnames()`, one can acces the data type of a TBranch without reading them using `.typenames`.

[46]:
```
events.typenames()
```

[46]:
```
{'Type': 'char*',
 'Run': 'int32_t',
 'Event': 'int32_t',
 'E1': 'double',
 'px1': 'double',
 'py1': 'double',
 'pz1': 'double',
 'pt1': 'double',
 'eta1': 'double',
 'phi1': 'double',
 'Q1': 'int32_t',
 'E2': 'double',
 'px2': 'double',
 'py2': 'double',
 'pz2': 'double',
 'pt2': 'double',
 'eta2': 'double',
 'phi2': 'double',
 'Q2': 'int32_t',
 'M': 'double'}
```

More interactively and convinient, one can use `.show()`

[47]:
```
events.show()
```

```
name                 | typename                 | interpretation
---------------------+--------------------------+-------------------------------
Type                 | char*                    | AsStrings()
Run                  | int32_t                  | AsDtype('>i4')
Event                | int32_t                  | AsDtype('>i4')
E1                   | double                   | AsDtype('>f8')
px1                  | double                   | AsDtype('>f8')
```

```
py1                  | double               | AsDtype('>f8')
pz1                  | double               | AsDtype('>f8')
pt1                  | double               | AsDtype('>f8')
eta1                 | double               | AsDtype('>f8')
phi1                 | double               | AsDtype('>f8')
Q1                   | int32_t              | AsDtype('>i4')
E2                   | double               | AsDtype('>f8')
px2                  | double               | AsDtype('>f8')
py2                  | double               | AsDtype('>f8')
pz2                  | double               | AsDtype('>f8')
pt2                  | double               | AsDtype('>f8')
eta2                 | double               | AsDtype('>f8')
phi2                 | double               | AsDtype('>f8')
Q2                   | int32_t              | AsDtype('>i4')
M                    | double               | AsDtype('>f8')
```

2 Reading a TBranch as an array

A TBranch can be turned into an array using `.array()` method.

[48]:
```
events["M"].array()
```

[48]:
```
<Array [82.5, 83.6, 83.3, … 96, 96.5, 96.7] type='2304 * float64'>
```

By defautl, the array is an array from Awkward Array, but a NumPy or a Pandas array (`pandas.Series` in the case of Pandas) can be created using the parameter `library="np"` or `library="pd"`.

[49]:
```
events["M"].array(library="np")
```

[49]:
```
array([82.46269156, 83.62620401, 83.30846467, …, 95.96547966,
       96.49594381, 96.65672765])
```

[50]:
```
events["M"].array(library="pd")
```

[50]:
```
0       82.462692
1       83.626204
2       83.308465
3       82.149373
4       90.469123
          …
2299    60.047138
2300    96.125376
2301    95.965480
2302    96.495944
2303    96.656728
Length: 2304, dtype: float64
```

The array method has multiple parameters such as delimitation, parellelization and others, as seen here.

2.1 Reading multiple TBranches as a group of arrays

If multiple TBranches are going to be used you could use the funtion **arrays**. As shown next,

[5]:
```
events.arrays(["px1", "py1", "pz1"])
```

[5]:
```
<Array [{px1: -41.2, … pz1: -74.8}] type='2304 * {"px1": float64, "py1":
float…'>
```

Just as with **array**, you can alter the default library to Pandas or Numpy. In Numpy, it will be exported as an dict of arrays

[6]:
```
events.arrays(["px1", "py1", "pz1"], library="np")
```

[6]:
```
{'px1': array([-41.19528764,  35.11804977,  35.11804977, …,  32.37749196,
         32.37749196,  32.48539387]),
 'py1': array([ 17.4332439 , -16.57036233, -16.57036233, …,   1.19940578,
          1.19940578,   1.2013503 ]),
 'pz1': array([-68.96496181, -48.77524654, -48.77524654, …, -74.53243061,
        -74.53243061, -74.80837247])}
```

And in Pandas, it will be exported as a `pandas.DataFrame`.

[7]:
```
events.arrays(["px1", "py1", "pz1"], library="pd")
```

[7]:
```
            px1        py1         pz1
0    -41.195288  17.433244  -68.964962
1     35.118050 -16.570362  -48.775247
2     35.118050 -16.570362  -48.775247
3     34.144437 -16.119525  -47.426984
4     22.783582  15.036444  -31.689894
…           …          …           …
2299  19.054651  14.833954   22.051323
2300 -68.041915 -26.105847 -152.235018
2301  32.377492   1.199406  -74.532431
2302  32.377492   1.199406  -74.532431
2303  32.485394   1.201350  -74.808372

[2304 rows x 3 columns]
```

2.1.1 Filtering TBranches

If no filtering arguments are passed to arrays, all TBranches will be read. To avoid this (for any reason) you can use the parameters `filter_name`, `filter_typenames` or `filter_branch` to select TBranches by name, type or other attributes. (This can be used with in other methods such as `keys`, `show` or `typename`). Additionally, `lambda` functions can be used in this parameters.

[8]:
```python
events.keys(filter_name="px*")
```

[8]: ['px1', 'px2']

[9]:
```python
events.arrays(filter_name="px*")
```

[9]: <Array [{px1: -41.2, … px2: -68.8}] type='2304 * {"px1": float64, "px2": float64}'>

[11]:
```python
events.keys(filter_name="/p[xyz][0-9]/i")
```

[11]: ['px1', 'py1', 'pz1', 'px2', 'py2', 'pz2']

[12]:
```python
events.arrays(filter_name="/p[xyz][0-9]/i")
```

[12]: <Array [{px1: -41.2, py1: 17.4, … pz2: -154}] type='2304 * {"px1": float64, "p…'>

[13]:
```python
events.keys(filter_branch=lambda b: b.compression_ratio > 10)
```

[13]: ['Run', 'Q1', 'Q2']

[14]:
```python
events.arrays(filter_branch=lambda b: b.compression_ratio > 10, library="pd")
```

[14]:

	Run	Q1	Q2
0	148031	1	-1
1	148031	-1	1
2	148031	-1	1
3	148031	-1	1
4	148031	1	-1
…	…	..	..
2299	148029	1	-1
2300	148029	-1	1
2301	148029	1	-1
2302	148029	1	-1
2303	148029	1	-1

[2304 rows x 3 columns]

2.2 Selections

2.2.1 Selections from 1D arrays

Another way to filter branches is if they are able to pass some criteria, for exmaple

[53]:
```python
branches = uproot.open("HSF training/uproot-tutorial-file.root:Events").arrays()

branches['nMuon'] == 1
```

[53]:
```
<Array [False, False, True, … False, False] type='100000 * bool'>
```

You can observe that the returned array is a boolean array, called *mask*.

[54]:
```python
single_muon_mask = branches['nMuon'] == 1
```

2.2.2 Applying a mask to an array

If we want to apply a selection to an array, we use the mask as an index, For example, if we want the pT of only those muons in events with exactly one muon,

[56]:
```python
branches['Muon_pt'][single_muon_mask]
```

[56]:
```
<Array [[3.28], [3.84], … [13.3], [9.48]] type='13447 * var * float32'>
```

2.3 Selections from a jagged array

To make a selection of this type of array, you use the absolute value and follow the same steps as in the 1D arrays,

[57]:
```python
eta_mask = abs(branches["Muon_eta"]) < 2
eta_mask
```

[57]:
```
<Array [[True, True], … True, True, True]] type='100000 * var * bool'>
```

2.4 Computing expressions and cuts

So far in the `arrays` method we've used the first argument to pass TBranches names. Additionally, it can also be used to compute expresions.

[16]:
```python
events.arrays("sqrt(px1**2 + py1**2)")
```

[16]:
```
<Array [{'sqrt(px1**2 + py1**2)': 44.7, … ] type='2304 * {"sqrt(px1**2 +
py1**…'>
```

You can use aliases to name the computations.

[17]:
```python
events.arrays("pt1", aliases={"pt1": "sqrt(px1**2 + py1**2)"})
```

[17]:
```
<Array [{pt1: 44.7}, … {pt1: 32.4}] type='2304 * {"pt1": float64}'>
```

The second argument is a filter ("cut") on entries. It is shown in the previous example that there is 2304 entries, while with a cut,

[20]:
```python
events.arrays("pt1", "pt1 > 50", aliases={"pt1": "sqrt(px1**2 + py1**2)"})
```

[20]:
```
<Array [{pt1: 77}, … {pt1: 72.9}] type='290 * {"pt1": float64}'>
```

there are only 290 entries. More filters can be applied using "&" or/and a pipe ("|"),

```
[23]: events.arrays(["pt1"], "(pt1 > 50) & ((E1>100) | (E1<90))", aliases={"pt1":␣
       ↪"sqrt(px1**2 + py1**2)"})
      # & for "and" and | for "or" (?)
```

```
[23]: <Array [{pt1: 77}, … {pt1: 72.9}] type='269 * {"pt1": float64}'>
```

As it has been said Uproot is uniquely used as an io library, so this filter funnels the command to Numpy. Nevertheless, if the computation requires more than one expression, you'll have to move it out of strings into Python.

3 Nested Data Structures

Not all entries have one value per entry. Take a look to the following array,

```
[25]: events = uproot.open("https://scikit-hep.org/uproot3/examples/HZZ.root:events")
      events.show()
```

```
name                 | typename                 | interpretation
---------------------+--------------------------+-------------------------------
NJet                 | int32_t                  | AsDtype('>i4')
Jet_Px               | float[]                  | AsJagged(AsDtype('>f4'))
Jet_Py               | float[]                  | AsJagged(AsDtype('>f4'))
Jet_Pz               | float[]                  | AsJagged(AsDtype('>f4'))
Jet_E                | float[]                  | AsJagged(AsDtype('>f4'))
Jet_btag             | float[]                  | AsJagged(AsDtype('>f4'))
Jet_ID               | bool[]                   | AsJagged(AsDtype('bool'))
NMuon                | int32_t                  | AsDtype('>i4')
Muon_Px              | float[]                  | AsJagged(AsDtype('>f4'))
Muon_Py              | float[]                  | AsJagged(AsDtype('>f4'))
Muon_Pz              | float[]                  | AsJagged(AsDtype('>f4'))
Muon_E               | float[]                  | AsJagged(AsDtype('>f4'))
Muon_Charge          | int32_t[]                | AsJagged(AsDtype('>i4'))
Muon_Iso             | float[]                  | AsJagged(AsDtype('>f4'))
NElectron            | int32_t                  | AsDtype('>i4')
Electron_Px          | float[]                  | AsJagged(AsDtype('>f4'))
Electron_Py          | float[]                  | AsJagged(AsDtype('>f4'))
Electron_Pz          | float[]                  | AsJagged(AsDtype('>f4'))
Electron_E           | float[]                  | AsJagged(AsDtype('>f4'))
Electron_Charge      | int32_t[]                | AsJagged(AsDtype('>i4'))
Electron_Iso         | float[]                  | AsJagged(AsDtype('>f4'))
NPhoton              | int32_t                  | AsDtype('>i4')
Photon_Px            | float[]                  | AsJagged(AsDtype('>f4'))
Photon_Py            | float[]                  | AsJagged(AsDtype('>f4'))
Photon_Pz            | float[]                  | AsJagged(AsDtype('>f4'))
Photon_E             | float[]                  | AsJagged(AsDtype('>f4'))
Photon_Iso           | float[]                  | AsJagged(AsDtype('>f4'))
MET_px               | float                    | AsDtype('>f4')
```

```
MET_py               | float                    | AsDtype('>f4')
MChadronicBottom_px  | float                    | AsDtype('>f4')
MChadronicBottom_py  | float                    | AsDtype('>f4')
MChadronicBottom_pz  | float                    | AsDtype('>f4')
MCleptonicBottom_px  | float                    | AsDtype('>f4')
MCleptonicBottom_py  | float                    | AsDtype('>f4')
MCleptonicBottom_pz  | float                    | AsDtype('>f4')
MChadronicWDecayQ… | float                    | AsDtype('>f4')
MChadronicWDecayQ… | float                    | AsDtype('>f4')
MChadronicWDecayQ… | float                    | AsDtype('>f4')
MChadronicWDecayQ… | float                    | AsDtype('>f4')
MChadronicWDecayQ… | float                    | AsDtype('>f4')
MChadronicWDecayQ… | float                    | AsDtype('>f4')
MClepton_px          | float                    | AsDtype('>f4')
MClepton_py          | float                    | AsDtype('>f4')
MClepton_pz          | float                    | AsDtype('>f4')
MCleptonPDGid        | int32_t                  | AsDtype('>i4')
MCneutrino_px        | float                    | AsDtype('>f4')
MCneutrino_py        | float                    | AsDtype('>f4')
MCneutrino_pz        | float                    | AsDtype('>f4')
NPrimaryVertices     | int32_t                  | AsDtype('>i4')
triggerIsoMu24       | bool                     | AsDtype('bool')
EventWeight          | float                    | AsDtype('>f4')
```

[26]:
```
events.keys(filter_name="/(Jet|Muon)_P[xyz]/")
```

[26]:
```
['Jet_Px', 'Jet_Py', 'Jet_Pz', 'Muon_Px', 'Muon_Py', 'Muon_Pz']
```

[28]:
```
ak_arrays = events.arrays(filter_name="/(Jet|Muon)_P[xyz]/")
ak_arrays[:2].tolist()
```

[28]:
```
[{'Jet_Px': [],
  'Jet_Py': [],
  'Jet_Pz': [],
  'Muon_Px': [-52.89945602416992, 37.7377815246582],
  'Muon_Py': [-11.654671669006348, 0.6934735774993896],
  'Muon_Pz': [-8.16079330444336, -11.307581901550293]},
 {'Jet_Px': [-38.87471389770508],
  'Jet_Py': [19.863452911376953],
  'Jet_Pz': [-0.8949416279792786],
  'Muon_Px': [-0.8164593577384949],
  'Muon_Py': [-24.404258728027344],
  'Muon_Pz': [20.199968338012695]}]
```

See Awkward array documentation for data analysis techniques using these types.

Just as with everything you can read the array with Numpy (array with `dtype="O"`) and Pandas (DataFrame with MultiIndex rows)

[29]:
```python
events.arrays(filter_name="/(Jet|Muon)_P[xyz]/", library="np")
```

[29]:
```
{'Jet_Px': array([array([], dtype=float32), array([-38.874714], dtype=float32),
        array([], dtype=float32), …, array([-3.7148185], dtype=float32),
        array([-36.361286, -15.256871], dtype=float32),
        array([], dtype=float32)], dtype=object),
 'Jet_Py': array([array([], dtype=float32), array([19.863453], dtype=float32),
        array([], dtype=float32), …, array([-37.202377], dtype=float32),
        array([ 10.173571, -27.175364], dtype=float32),
        array([], dtype=float32)], dtype=object),
 'Jet_Pz': array([array([], dtype=float32), array([-0.8949416], dtype=float32),
        array([], dtype=float32), …, array([41.012222], dtype=float32),
        array([226.42921  ,  12.119683], dtype=float32),
        array([], dtype=float32)], dtype=object),
 'Muon_Px': array([array([-52.899456,  37.73778 ], dtype=float32),
        array([-0.81645936], dtype=float32),
        array([48.98783   ,  0.8275667], dtype=float32), …,
        array([-29.756786], dtype=float32),
        array([1.1418698], dtype=float32),
        array([23.913206], dtype=float32)], dtype=object),
 'Muon_Py': array([array([-11.654672 ,   0.6934736], dtype=float32),
        array([-24.404259], dtype=float32),
        array([-21.723139,  29.800508], dtype=float32), …,
        array([-15.303859], dtype=float32),
        array([63.60957], dtype=float32),
        array([-35.665077], dtype=float32)], dtype=object),
 'Muon_Pz': array([array([ -8.160793, -11.307582], dtype=float32),
        array([20.199968], dtype=float32),
        array([11.168285, 36.96519 ], dtype=float32), …,
        array([-52.66375], dtype=float32),
        array([162.17632], dtype=float32),
        array([54.719437], dtype=float32)], dtype=object)}
```

[30]:
```python
events.arrays(filter_name="/(Jet|Muon)_P[xyz]/", library="pd")
```

[30]:
```
(                   Jet_Px     Jet_Py      Jet_Pz
 entry subentry
 1     0        -38.874714  19.863453   -0.894942
 3     0        -71.695213  93.571579  196.296432
       1         36.606369  21.838793   91.666283
       2        -28.866419   9.320708   51.243221
 4     0          3.880162 -75.234055 -359.601624
 …                     …          …          …
 2417  0        -33.196457 -59.664749  -29.040150
       1        -26.086025 -19.068407   26.774284
 2418  0         -3.714818 -37.202377   41.012222
 2419  0        -36.361286  10.173571  226.429214
```

```
      1        -15.256871 -27.175364   12.119683

[2773 rows x 3 columns],
                    Muon_Px    Muon_Py     Muon_Pz
entry subentry
0     0        -52.899456 -11.654672   -8.160793
      1         37.737782   0.693474  -11.307582
1     0         -0.816459 -24.404259   20.199968
2     0         48.987831 -21.723139   11.168285
      1          0.827567  29.800508   36.965191
…                  …          …          …
2416  0        -39.285824 -14.607491   61.715790
2417  0         35.067146 -14.150043  160.817917
2418  0        -29.756786 -15.303859  -52.663750
2419  0          1.141870  63.609570  162.176315
2420  0         23.913206 -35.665077   54.719437

[3825 rows x 3 columns])
```

Each row of the DataFrame represents one particle and the row index is broken down into "entry" and "subentry" levels. If the selected TBranches include data with different numbers of values per entry, then the return value is not a DataFrame, but a tuple of DataFrames, one for each multiplicity. See the Pandas documentation on joining for tips on how to analyze DataFrames with partially shared keys ("entry" but not "subentry"). (?)

4 Iterating over intervals of entries

If files are too large, it is better to iterate over an intervarl in order to not run out of memory. For this, you use a `for` loop and indicate a step size,

[31]:
```
events = uproot.open("https://scikit-hep.org/uproot3/examples/Zmumu.root:
↪events")

for batch in events.iterate(step_size=500):
    print(repr(batch))
```

```
<Array [{Type: 'GT', Run: 148031, … M: 87.7}] type='500 * {"Type": string,
"Ru…'>
<Array [{Type: 'GT', Run: 148031, … M: 72.5}] type='500 * {"Type": string,
"Ru…'>
<Array [{Type: 'TT', Run: 148031, … M: 92.9}] type='500 * {"Type": string,
"Ru…'>
<Array [{Type: 'GT', Run: 148031, … M: 94.6}] type='500 * {"Type": string,
"Ru…'>
<Array [{Type: 'TT', Run: 148029, … M: 96.7}] type='304 * {"Type": string,
"Ru…'>
```

You can also add a filter as it shown previously.

A better method to iterate entries is to select instead a number of bytes,

[32]:
```
for batch in events.iterate(step_size="50 kB"):
    print(repr(batch))
```

```
<Array [{Type: 'GT', Run: 148031, … M: 89.6}] type='667 * {"Type": string,
"Ru…'>
<Array [{Type: 'TT', Run: 148031, … M: 18.1}] type='667 * {"Type": string,
"Ru…'>
<Array [{Type: 'GT', Run: 148031, … M: 94.7}] type='667 * {"Type": string,
"Ru…'>
<Array [{Type: 'GT', Run: 148029, … M: 96.7}] type='303 * {"Type": string,
"Ru…'>
```

Again, Pandas and Numpy can be used.

5 Iterating over many files

Often, larger data sets consist of many files and other abstractions such as ROOT's TChain. So, in order to iterate many files you can use the functino `uproot.iterate` that takes a list of files as its first argument,

[3]:
```
for batch in uproot.iterate(["https://scikit-hep.org/uproot3/examples/Zmumu.root:events", "https://scikit-hep.org/uproot3/examples/HZZ.root:events"]):
    # do something
    pass
```

The specification of file names has to include paths to the TTree objects, so the colon isn't exactly optional. Since it is possible for file paths to include colons as part of the file or directory name, the following alternate syntax can also be used,

[7]:
```
for batch in uproot.iterate([{"PyHEP_UPROOTandAWKWARD/data/Zmumu.root": "events"}]):
    # do something
    pass
```

6 Reading many files into big arrays

`uproot.iterate` function is nota a direct analogy of ROOT's TChain because it does not make multifile workflows look like a single file workflows.

The simplest way to access many files is to chain them into one array. The `uproot.concatenate` function is a multi-file analogue of the arrays method, in that it returns a single array group,

[8]:
```
uproot.concatenate(["https://scikit-hep.org/uproot3/examples/Zmumu.root:events", "https://scikit-hep.org/uproot3/examples/HZZ.root:events"])
```

[8]: <Array [{Type: 'GT', … EventWeight: 0.00876}] type='4725 * union[{"Type":
stri…'>

A down side, is that the array is entirely read into memory, so this is only posible if, - the files are smanll, - the number of files is small, or - the selected branches do not represent a large fraction of the files

decent RAM memory is adviced.

7 Reading on demand with lazy-arrays

Lazy-loading is a third way to access multifile datasets. The interface to `uproot.lazy` is like `uproot.concatenate` in that it returns a single object, not an iterator that you have to iterate through, but it is like `uproot.iterate` in that the data are not loaded immediately and do not need to reside in memory all at once.

[11]:
```
array = uproot.lazy(["https://scikit-hep.org/uproot3/examples/Zmumu.root:
↪events", "PyHEP_UPROOTandAWKWARD/data/Zmumu.root:events"])
array
```

[11]: <Array [{Type: 'GT', Run: 148031, … M: 96.7}] type='4608 * {"Type": string,
"R…'>

When `uproot.lazy` is called, it opens all of the specified files and TTree metadata, but none of the TBranch data. It uses the TBranch names and types, as well as the TTree num_entries, to define the data type and prepare batches for reading. Only when you access items in the array, such as printing them to the screen or performing a calculation on them, are the relevant TBranches read (in batches).

This lazy-loading uses an Awkward Array feature, so `library="ak"` is the only library option.

The data being loaded is intentionally hidden, if you're interested in watching filling up you can use the `uproot.LRUAArrayCache` function.

[13]:
```
cache = uproot.LRUArrayCache("1 GB")
array = uproot.lazy("https://scikit-hep.org/uproot3/examples/Zmumu.root:events",
                    step_size=100,
                    array_cache=cache)

cache
```

[13]: <LRUArrayCache (0/1000000000 bytes full) at 0x029205b383d0>

As it was previosly said, the lazy array doesn't load any data (untill it is called). If we then ask for a single element from a single field, it loads one TBranch-batch. Since we specified the `step_size=100` (much too small for a real case; the default is "100 MB"), this TBranch-bath is 100 entries ($\approx$ 800 bytes).

```
[15]: array["px1", 1]
      cache
```

```
[15]: <LRUArrayCache (800/1000000000 bytes full) at 0x029205b383d0>
```

If we request another item from the same batch, it doesn't load anything else,

```
[16]: array["px1", 2]
      cache
```

```
[16]: <LRUArrayCache (800/1000000000 bytes full) at 0x029205b383d0>
```

It will, nevertheless, load more data when an item is either outside of the batch or if another TBranch is indicated,

```
[17]: array["px1", 100]
      print(cache)
      array["py1", 0]
      print(cache)
```

```
<LRUArrayCache (1600/1000000000 bytes full) at 0x029205b383d0>
<LRUArrayCache (2400/1000000000 bytes full) at 0x029205b383d0>
```

Although lazy arrays combine the convenience of uproot.concatenate with the gradual loading of uproot.iterate, it is not always the most efficient way to process data. Derived quantities are fully resident in memory, and most data analyses compute more quantities than they read.

Moreover, if a lazy array is larger than its cache, reading the last batches will cause the first batches to be evicted from the cache. If it is accessed again, the first batches will need to be fully re-read, which evicts the last batches, guaranteeing that data will never be found in the cache when it's needed.

On the other hand, if you make the cache(s) large enough to accommodate all the arrays you'll be loading, then you might as well load them entirely into memory. Avoiding the overhead of managing lazy batch-loading can only streamline a workflow.

8 Caching and memory management

Each file has an associated `object_cache` and `array_cache`, which stramline interactive use but could track down memory use.

The `object_cache` stores a number of objects like TDirectories, histograms and TTrees. The main effect of this is that,

```
[18]: file = uproot.open("https://scikit-hep.org/uproot3/examples/hepdata-example.
      ↪root")
      hist = file["hpx"]
      (hist, hist)
```

```
[18]: (<TH1F (version 1) at 0x0292005c8640>, <TH1F (version 1) at 0x0292005c8640>)
```

and

[19]:
```
(file["hpx"], file["hpx"])
```

[19]:
```
(<TH1F (version 1) at 0x0292005c8640>, <TH1F (version 1) at 0x0292005c8640>)
```

have identical performance. In other words, not having to declare names for things that are already referenced by name simplifies bookkeeping

The `array_cache` stores array output up to a maximum of bytes. The `array_cache` ensures that,

[20]:
```
events = uproot.open("https://scikit-hep.org/uproot3/examples/Zmumu.root:
↪events")
array = events["px1"].array()
(array, array)
```

[20]:
```
(<Array [-41.2, 35.1, 35.1, … 32.4, 32.5] type='2304 * float64'>,
 <Array [-41.2, 35.1, 35.1, … 32.4, 32.5] type='2304 * float64'>)
```

and

[21]:
```
(events["px1"].array(), events["px1"].array())
```

[21]:
```
(<Array [-41.2, 35.1, 35.1, … 32.4, 32.5] type='2304 * float64'>,
 <Array [-41.2, 35.1, 35.1, … 32.4, 32.5] type='2304 * float64'>)
```

have the same performance, assuming that the caches are not overrun.

By default, each file has a separate cache of 100 objects and "100 MB" of arrays. However, these can be overridden by passing an object_cache or array_cache argument to uproot.open or setting the object_cache and array_cache properties.

9 Parallel processing

Data are or can be read in parallel in each of the following three stages.

- Physically reading bytes from disk or remote sources : the parallel processing or single-thread background processing is handled by the specific `uproot.source.chunk.Source` type, which can be influenced with `uproot.open` options (particularly `num_workers` and `num_fallback_workers`).
- Decompressing TBasket (uproot.models.TBasket.Model_TBasket) data: depends on the `decompression_executor`.
- Interpreting decompressed data with an array `uproot.interpretation.Interpretation` : depends on the `interpretation_executor`.

Like the caches, the default values for the last two are global `uproot.decompression_executor` and `uproot.interpretation_executor objects`. The default `decompression_executor` is a `uproot.ThreadPoolExecutor` with as many workers as your computer has CPU cores. Decompression workloads are executed in compiled extensions with the Python GIL released, so they can afford to run with full parallelism. The default `interpretation_executor` is a `uproot.TrivialExecutor` that behaves like an distributed executor, but actually runs sequentially. Most interpretation workflows are not computationally intensive or are currently implemented in Python, so they would not currently benefit from parallelism.

If, however, you're working in an environment that puts limits on parallel processing, you may want to modify the defaults, either locally through a `decompression_executor` or `interpretation_executor` function parameter, or globally by replacing the global object.

10 Opening a file for writing

To write ROOT files, you can open them using,

```
new_file = uproot.recreate("path/to/new-file.root")

existing_file = uproot.update("path/to/existing-file.root")
```

This functions should be used like this,

```
with uproot.recreate("path/to/new-file.root") as file:
    do_something...
```

It should be noted that this functions return a `uproot.WritableDirectory` instead of an `uproot.ReadOnlyDirectory` that `uproot.open` returns, and these objects have different methods.

10.1 Writing objects to a file

The object returned by uproot.recreate or uproot.update represents a TDirectory inside the file.

[22]:
```
file = uproot.recreate("example.root")
file
```

[22]: `<WritableDirectory '/' at 0x02920062a430>`

This is a python MutableMapping, wich means you can write date just by assigning it,

[24]:
```
import numpy as np

file["hist"] = np.histogram(np.random.normal(0, 1, 1000000))
file["hist"]
```

[24]: `<TH1D (version 3) at 0x02920062ac70>`

It also works to add a nested directory by adding slashes ("/") in the name,

[25]:
```
file["subdir/hist"] = np.histogram(np.random.normal(0, 1, 100000))
file["subdir/hist"]
```

[25]: `<TH1D (version 3) at 0x029200649be0>`

[26]:
```
file.keys()
```

```
[26]: ['hist;1', 'subdir;1', 'subdir/hist;1']
```

```
[27]: file.classnames()
```

```
[27]: {'hist;1': 'TH1D', 'subdir;1': 'TDirectory', 'subdir/hist;1': 'TH1D'}
```

Empty directories can be made with the mkdir method.

> **Note**
>
> A small but growing list of data types can be written to files:
>
> - strings: TObjString
> - histograms: TH1*, TH2*, TH3*
> - profile plots: TProfile, TProfile2D, TProfile3D
> - NumPy histograms created with np.histogram, np.histogram2d, and np.histogramdd with 3 dimensions or fewer
> - histograms that satisfy the Universal Histogram Interface (UHI) with 3 dimensions or fewer; this includes boost-histogram and hist
> - PyROOT objects

10.2 Removing objects from a file

You can use the `del` operator,

```
[29]: del file["hist"]
```

```
[30]: file.keys()
```

```
[30]: ['subdir;1', 'subdir/hist;1']
```

11 Writing TTrees to a file

To create a TTree, you can use the object `uproot.WritableTree`, which can be created using various methods. First, you can use a directory,

```
[31]: file["tree1"] = {"branch1": np.arange(1000), "branch2": np.arange(1000)*1.1}
      file["tree1"].show()
```

```
name                 | typename                 | interpretation
---------------------+--------------------------+-------------------------------
```

```
branch1                  | int32_t                  | AsDtype('>i4')
branch2                  | double                   | AsDtype('>f8')
```

You can also use Numpy arrays and Pandas DataFrames (of equal length),

[33]:
```python
import pandas as pd

df = pd.DataFrame({"x": np.arange(1000),
                   "y": np.arange(1000)*1.1})
df
```

[33]:
```
       x       y
0      0     0.0
1      1     1.1
2      2     2.2
3      3     3.3
4      4     4.4
..   ...     ...
995  995  1094.5
996  996  1095.6
997  997  1096.7
998  998  1097.8
999  999  1098.9

[1000 rows x 2 columns]
```

[34]:
```python
file["tree2"] = df
file["tree2"].show()
```

```
name                 | typename                 | interpretation
---------------------+--------------------------+-------------------------------
index                | int64_t                  | AsDtype('>i8')
x                    | int32_t                  | AsDtype('>i4')
y                    | double                   | AsDtype('>f8')
```

In Awkward, nonetheless, arrays can contain a variable number of values per entry,

[36]:
```python
import awkward as ak

file["tree3"] = {"branch": ak.Array([[1.1, 2.2, 3.3], [], [4.4, 5.5]])}
file["tree3"].show()
```

```
name                 | typename                 | interpretation
---------------------+--------------------------+-------------------------------
nbranch              | int32_t                  | AsDtype('>i4')
branch               | double[]                 | AsJagged(AsDtype('>f8'))
```

And Awkward record arrays, constructed with ak.zip, can consolidate arrays to ensure that there is only one “counter” TBranch.

Note

The small but growing list of data types can be written as TTrees is:

- dict of NumPy arrays (flat, multidimensional, and/or structured), Awkward Arrays containing one level of variable-length lists and/or one level of records, or a Pandas DataFrame with a numeric index
- a single NumPy structured array (one level deep)
- a single Awkward Array containing one level of variable-length lists and/or one level of records
- a single Pandas DataFrame with a numeric index

Just as empty directories can be made with the `mkdir` method, empty TTrees can be made with `mktree`.

[40]:
```
file.mktree("tree5", {"x": ("f4", (3,)), "y": "var * int64"}, title="A title")
```

[40]:
```
<WritableTree '/tree5' at 0x029205f0ed30>
```

[41]:
```
file["tree5"].show()
```

```
name                 | typename                 | interpretation
---------------------+--------------------------+-------------------------------
x                    | float[3]                 | AsDtype("('>f4', (3,))")
ny                   | int32_t                  | AsDtype('>i4')
y                    | int64_t[]                | AsJagged(AsDtype('>i8'))
```

This method also provides control over the naming convention for counter TBranches and subfield TBranches

12 Extanding TTrees with large datasets

In order to weite more data to dhte disk than can fit in the memory, you can use the extend method.

[42]:
```
file["tree5"].num_entries, file["tree5"].num_baskets
```

[42]:
```
(0, 0)
```

[43]:
```
file["tree5"].extend({
    "x": np.arange(15).reshape(5, 3),
    "y": ak.Array([[0.0, 1.1, 2.2], [], [3.3, 4.4], [5.5], [6.6, 7.7, 8.8, 9.
 ↪9]])
})
```

```
file["tree5"].num_entries, file["tree5"].num_baskets
```

[43]: (5, 1)

```
[44]: file["tree5"].extend({
    "x": np.arange(15).reshape(5, 3),
    "y": ak.Array([[0.0, 1.1, 2.2], [], [3.3, 4.4], [5.5], [6.6, 7.7, 8.8, 9.
 ↪9]])
})

file["tree5"].num_entries, file["tree5"].num_baskets
```

[44]: (10, 2)

The `extend` method always adds one TBasket to each TBranch in the TTree. The data you provide must have the types that have been established in the first write or mktree call: exactly the same set of TBranch names and the same data type for each TBranch.

The arrays also have to have the same lengths as each other, though only in the first dimension. Above, the "x" NumPy array has shape (5, 3): the first dimension has length 5. The "y" Awkward array has type 5 * var * float64: the first dimension has length 5. This is why they are compatible; the inner dimensions don't matter (except inasmuch as they have the right type).

Note: Make sure the `extend` method includes at least 100 kb per branch. If uproot writes very small baskets it will spend more time working on the TBasket overhead than actually writting data.

13 Specifying the compression

You can specify the compression for a whole file while opening it. It is also mutable.

```
[47]: file = uproot.recreate("example.root", compression=uproot.ZLIB(4))
file.compression
```

[47]: ZLIB(4)

The `uproot.WritableTree` object also have a compression setting that can overide the global one. Additionaly, each TBranch can have a different compression setting.