HEP Python

Alexander Richards

Imperial College Sci. & Med. UK
(IC)

July 2019

ROOT bindings for Python: PyROOT

> *PyROOT is a Python extension module that allows the user to interact with any ROOT class from the Python interpreter.*
>
> *https://root.cern.ch/pyroot*

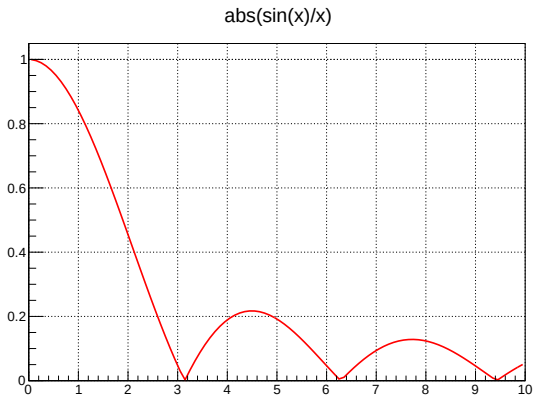
```
>>> import ROOT
```

- Requires C++ ROOT installation
- The top level Python module is *ROOT.py*
- This is the main entry point for any Python script using the ROOT classes
- This module imports the extension module libPyROOT.so [.dll] and does a similar initialization as the ROOT application (i.e. loading common libraries, defining a number of globals, starting a thread to handle GUI events, etc.)

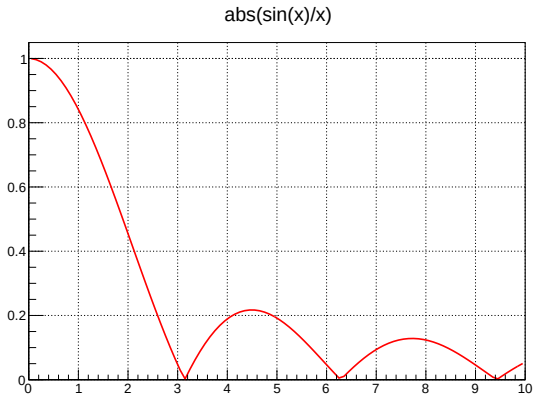
PyROOT: API Example

Syntax very similar to C++ original:

```
>>> from ROOT import gROOT, TCanvas, TF1
>>> gROOT.Reset()
>>> c1 = TCanvas("c1", "Title", 200, 10, 700, 500)
>>> fun1 = TF1("func", "abs(sin(x)/x)", 0, 10)
>>> c1.SetGridx()
>>> c1.SetGridy()
>>> fun1.Draw()
>>> c1.Update()
```

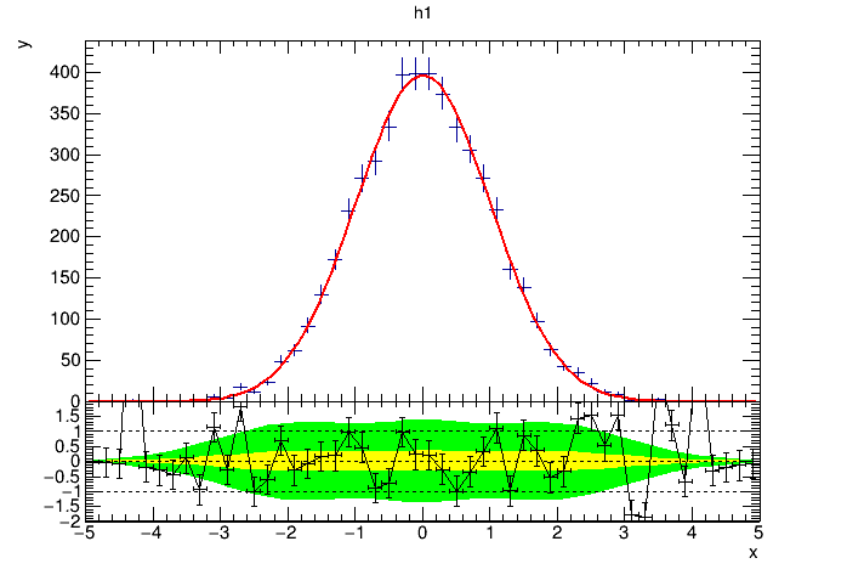



```
root [0] gROOT->Reset();
root [1] TCanvas c1("c1", "Title", 200, 10, 700, 500);
root [2] TF1 fun1("func", "abs(sin(x)/x)", 0, 10);
root [3] c1.SetGridx();
root [4] c1.SetGridy();
root [5] fun1.Draw();
root [6] c1.Update();
```



PyRoot: More Complicated Example

```
>>> import ROOT
>>> ROOT.gStyle.SetOptStat(0)
>>> c1 = ROOT.TCanvas("c1", "fit␣residual␣simple")
>>> ROOT.gPad.SetFrameFillStyle(0)
>>> h1 = ROOT.TH1D("h1", "h1", 50, -5, 5)
>>> h1.FillRandom("gaus", 5000)
>>> h1.Fit("gaus", "S")
>>> h1.Sumw2()
>>> h1.GetXaxis().SetTitle("x")
>>> h1.GetYaxis().SetTitle("y")
>>> rp1 = ROOT.TRatioPlot(h1, "errfunc")
>>> rp1.SetGraphDrawOpt("L")
>>> rp1.SetSeparationMargin(0.0)
>>> rp1.Draw()
>>> rp1.GetLowerRefGraph().SetMinimum(-2)
>>> rp1.GetLowerRefGraph().SetMaximum(2)
>>> c1.Update()
```



rootpy

“

- *The PyROOT bindings introduced ROOT into the world of Python, however, interacting with ROOT in Python should not feel like you are still writing C++*
- *The rootpy project is a community-driven initiative aiming to provide a more pythonic interface with ROOT on top of the existing PyROOT bindings*
- *Given Python's reflective and dynamic nature, rootpy also aims to improve ROOT design flaws and supplement existing ROOT functionality*
- *rootpy provides an interface and conversion mechanisms (lacking in PyROOT) required to make use of powerful packages such as SciPy, NumPy, matplotlib, scikit-learn, and PyTables (through the* ***root_numpy*** *package not described as part of this lecture)*

”

rootpy.org

rootpy: Simple migration from PyROOT

- rootpy provides via rootpy.ROOT a drop-in replacement for the PyROOT import.
- It mimics the PyROOT interface meaning you can run all your PyROOT code as before.
- It simultaneously provides access to it's own classes under a single import point so you don't need to remember where to import them from.

```
>>> import rootpy.ROOT as ROOT
>>> ROOT.TH1F('name', 'title', 10, 0, 1)
Hist('name')
>>> ROOT.Hist(10, 0, 1, name='name', type='F')
Hist('name')
```

- rootpy classes are essentially wrappers around the underlying PyROOT classes
- Note when using this rootpy.ROOT, interface PyROOT classes are automatically transformed into their rootpy equivalents after construction

rootpy: Histograms

- rootpy simplifies ROOT's histogram class hierarchy into three classes.

```
>>> from rootpy.plotting import Hist, Hist2D, Hist3D
```

- These classes dynamically inherit from the corresponding ROOT histogram class depending on the value of the *type* keyword argument supplied to the constructor.
- Type can be any of the types corresponding to the ROOT histogram classes: "C", "S", "I", "F", "D" (or lower case)
- If type is not specified the default is to use floating-point bins ("F").

```
>>> hist = Hist2D(10, 0, 1, 5, 0, 1)
>>> isinstance(hist, ROOT.TH2F)
True
>>> isinstance(Hist2D(10, 0, 1, 5, 0, 1, type='C'), ROOT.TH2C)
True
>>> isinstance(Hist2D(10, 0, 1, 5, 0, 1, type='D'), ROOT.TH2D)
True
```

rootpy: Histograms

- Unlike with PyROOT and C++ ROOT, the *name* and *title* arguments are optional.
- If not given then a UUID is used for the histogram name to avoid collision with any other objects.

```
>>> h = Hist(10, 0, 1, name="some name", title="some title")
>>> h.GetName(), h.GetTitle()
('some name', 'some title')
```

- The arguments that specify the binning are consumed from left to right as either three values for the number of bins, left bound, and right bound, or a list for bins of variable width.
- Fixed and variable width bins may be mixed for 2D and 3D histograms

```
# 1D hist with fixed width bins
>>> h1d=Hist(5, -2, 4)
# variable width bins
>>> h1d=Hist([-10,-3.2, 5.2, 35.])
```

```
# 2D hist, fixed width x & y
>>> h2d = Hist2D(10, 0, 1, 5, -2, 4)
# variable width x, fixed width y
>>> h2d = Hist2D([10, 30, 100, 1000],
                 4, 10, 33.5)
```

rootpy: Histogram Indexing and Slicing

- Histograms support slicing by global bin index or along each axis separately.
- This returns either a BinProxy, HistIndexView, Hist[2D|3D]View
- Can get/set bin content and errors with value and error attributes

```
>>> h1d[1].value, h1d[1].error
(2.0, 1.5)
```

- Can set content/error for an entire slice at once.

```
>>> a = Hist3D(10, 0, 1, 10, 0, 1, 10, 0, 1)
>>> a[:,:, 5] = a[40] # set with a BinProxy
>>> a[:,:,:] = (2, 4) # set content and error with a 2-tuple
>>> a[:,:,:] = 2 # only set the content
```

- Can construct another histogram from a view, in this case the step member of a slice translates to a rebinning along the associated axis

```
>>> b = Hist3D(a[3:5 ,:: 2 ,:])
```

rootpy: Cuts

- The rootpy rootpy.tree.Cut class inherits from ROOT.TCut
- It implements logical operators so cuts can be easily combined

```
>>> from rootpy.tree import Cut
>>> cut1 = Cut('a < 10')
>>> cut2 = Cut('b % 2 == 0')
>>> cut = cut1 & cut2
>>> print(cut)
(a<10)&&(b%2==0)
# expansion of ternary conditions
>>> cut3 = Cut('10 < a < 20')
>>> print(cut3)
(10<a)&&(a<20)
# easily combine cuts arbitrarily
>>> cut = ((cut1 & cut2) | -cut3)
>>> print(cut)
((a<10)&&(b%2==0 ))||(!(( 10<a)&&(a<20)))
```

rootpy: Trees

- rootpy provides pythonized subclass for ROOT's TTrees
- rootpy trees add additional API to ease their creation in Python

```
>>> from rootpy.tree import Tree
>>> from random import gauss
>>> tree = Tree("test")
>>> tree.create_branches(
        {'x': 'F',
         'y': 'F',
         'z': 'F',
         'i': 'I'})
>>> for i in range(10000):
        tree.x = gauss(.5, 1.)
        tree.y = gauss(.3, 2.)
        tree.z = gauss(13., 42.)
        tree.i = i
        tree.fill()
>>> tree.write()
```

rootpy: Trees

- rootpy provides pythonized subclass for ROOT's TTrees
- rootpy trees add additional API to ease their creation in Python
- TTree's Draw method is overridden to support returning and styling the created histogram

```
>>> mytree = myfile.treename
>>> hist = mytree.Draw('x_expr:y_expr',
                       selection='a < 20',
                       linecolor='red',
                       fillstyle='/')
```

rootpy: Trees

- rootpy provides pythonized subclass for ROOT's TTrees
- rootpy trees add additional API to ease their creation in Python
- TTree's Draw method is overridden to support returning and styling the created histogram
- You can quickly access the names of branches in a Tree by using .b on a tree
- They can be directly used in comparisons to make rootpy.tree.Cut expressions

```
>>> tree.b.evt_number
'evt_number'
>>> tree.b.evt_number > 3
'(evt_number > 3)'
>>> tree.Draw(tree.b.evt_number*2,
              tree.b.run_num > 2)
>>> tree.Draw(tree.b.evt_number
              >> (100,0,2))
```

rootpy: Files

- rootpy provides the rootpy.io.root_open() function
- Internally it uses ROOT's TFile::Open,
- But returns a rootpy.io.File object

```
>>> from rootpy.io import root_open
>>> myfile = root_open('some_file.root', 'recreate')
>>> myfile
File('some_file.root')
>>> isinstance(myfile, ROOT.TFile)
True
```

rootpy: Files

- rootpy provides the rootpy.io.root_open() function
- Internally it uses ROOT's TFile::Open,
- But returns a rootpy.io.File object
- This file object can act as a context manager
- Any objects retrieved from a rootpy File are automatically cast to the appropriate subclass in rootpy, if one exists

```
>>> with root_open('some_file.root') as myfile:
...     # the file is open in this context
...     myhist = myfile.somedirectory.histname.Clone()
...     myhist.SetDirectory(0)
...     myhist
Hist2D('hist2d')
>>> # when the context is left the file is closed
```

rootpy: Files

- rootpy provides the rootpy.io.root_open() function
- Internally it uses ROOT's TFile::Open,
- But returns a rootpy.io.File object
- This file object can act as a context manager
- Any objects retrieved from a rootpy File are automatically cast to the appropriate subclass in rootpy, if one exists
- rootpy files can be "walked" in a similar way to Python's os.walk()

```
>>> myfile = root_open('some_file.root')
>>> # recursively walk through the file
>>> for path, dirs, objects in myfile.walk():
...     # do something
...     print(path, dirs, objects)
```

ROOT I/O in pure Python: Uproot

> *Uproot (originally µproot, for "micro-Python ROOT") is a reader and a writer of the ROOT file format using only Python and Numpy.*
> *Unlike PyROOT and root_numpy, uproot does not depend on C++ ROOT. Instead, it uses Numpy to cast blocks of data from the ROOT file as Numpy arrays.*
>
> *github.com/scikit-hep/uproot*

```
>>> import uproot as ur
```

Uproot: Opening/Exploring files

- uproot.open is the entrypoint for reading a single file
- Can read local file or remote files over HTTP or XRootD (http:// or root:// respectively)
- HTTP access requires installation of the requests package while XRootD requires pyxrootd.
- All directories, including the open file itself, expose a dict-like (ROOTDirectory) structure)
- Deep nesting can be accessed as a path such that the following are equivalent:

```
>>> root_file["one"]["two"]["tree"]
>>> root_file["one/two/tree"]
```

- In addition to the usual *keys*, *values* and *items* methods, we have *allkeys*, *allvalues* and *allitems* which recursively search through the subdirectories
- The above methods can all be filtered using unary predicate functions as the *filtername* and/or *filterclass* arguments

Uproot: Exploring TTrees

- TTrees are also dict-like containing name: TBranch mappings
- In addition they have the following self-explanatory attributes:

```
>>> ttree.name, ttree.title, ttree.numentries
(b'events', b'Z->mumu events', 2304)
```

- The *show* method can be used to print a list of branches and their interpretations (types)

```
>>> ttree.show()
Type    (no streamer)   asstring()
Run     (no streamer)   asdtype('>i4')
Event   (no streamer)   asdtype('>i4')
E1      (no streamer)   asdtype('>f8')
px1     (no streamer)   asdtype('>f8')
py1     (no streamer)   asdtype('>f8')
pz1     (no streamer)   asdtype('>f8')
pt1     (no streamer)   asdtype('>f8')
eta1    (no streamer)   asdtype('>f8')
```

Uproot: Reading data arrays from a TTree

The bulk data in a TTree are not read until requested. There are many ways to do that:

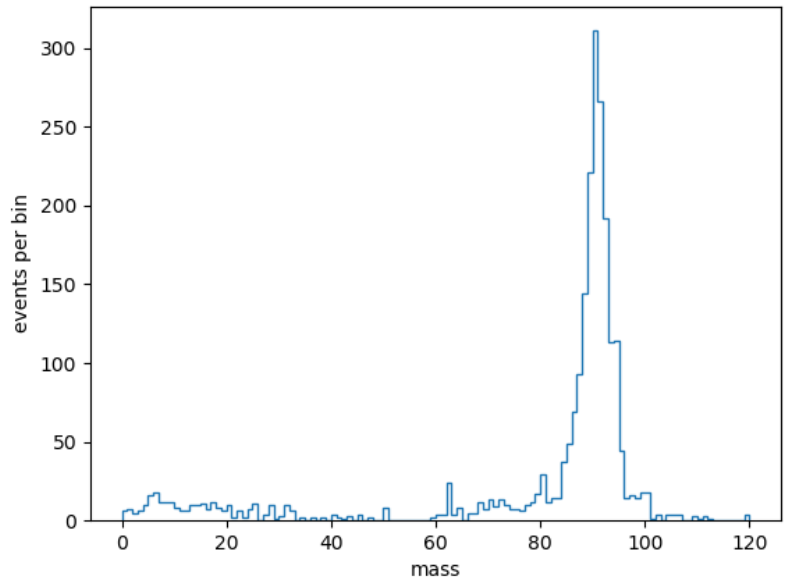
- select a TBranch and call TBranchMethods.array
- call TTreeMethods.array directly from the TTree object
- call TTreeMethods.arrays to get several arrays at a time
- call TBranch.lazyarray, TTreeMethods.lazyarray, TTreeMethods.lazyarrays, or uproot.lazyarrays to get array-like objects that read on demand
- call TTreeMethods.iterate or uproot.iterate to explicitly iterate over chunks of data (to avoid reading more than would fit into memory)
- call TTreeMethods.pandas or uproot.pandas.iterate to get Pandas DataFrames (Pandas must be installed).

When specifying multiple branches, we use a list of names, wildcard glob characters (*, ?, [...]) or full regex if surrounded by slashes:

```
>>> ttree.arrays(["px1", "py1", "pz1"])
>>> ttree.arrays("p[xyz]1")
>>> ttree.arrays("/p[xyz]1/")
```

Uproot: Basic Example

```
>>> url = "http://scikit-hep.org/uproot/examples/Zmumu.root"
>>> events = ur.open(url)["events"]
>>> data = events.arrays(["E*", "p[xyz]*"], namedecode="utf-8")
>>> mass = np.sqrt((data['E1'] + data['E2'])** 2
                   -(data['px1'] + data['px2'])** 2
                   -(data['py1'] + data['py2'])** 2
                   -(data['pz1'] + data['pz2'])** 2)
>>> plt.hist(mass, bins=120, range=(0, 120), histtype='step')
>>> plt.xlabel("mass")
>>> plt.ylabel("events per bin")
>>> plt.show()
```



Uproot: Output types

- By default when getting data arrays from uproot they are returned in a dict mapped to the branch name.
- we can however specify an alternatives such as OrderedDict, tuple or namedtuple using the *outputtype* argument.
- This can be useful for unpacking data structures directly from the uproot command, e.g.

```
>>> px, py, pz = tree.arrays("p[xyz]", outputtype=tuple)
```

- Or accessing accessing arrays as attributes without having to index

```
>>> data = tree.arrays("p[xyz]", outputtype=namedtuple)
>>> mag2 = data.px**2 + data.py**2 + data.pz**2
```

- Can put the arrays into pandas DataFrames as well

```
>>> df = tree.arrays("p[xyz]", outputtype=pd.DataFrame)
```

NOTE: Pandas DataFrames are such an important container type that there exists specialised functions (TTreeMethods.pandas.df and uproot.pandas.df) to achieve the above, e.g. `>>> df = tree.pandas.df("p[xyz]")`

Uproot: Output types

- By default when getting data arrays from uproot they are returned in a dict mapped to the branch name.
- we can however specify an alternatives such as OrderedDict, tuple or namedtuple using the *outputtype* argument.
- This can be useful for unpacking data structures directly from the uproot command, e.g.

```
>>> px, py, pz = tree.arrays("p[xyz]", outputtype=tuple)
```

- Or accessing accessing arrays as attributes without having to index

```
>>> data = tree.arrays("p[xyz]", outputtype=namedtuple)
>>> mag2 = data.px**2 + data.py**2 + data.pz**2
```

- Can put the arrays into pandas DataFrames as well

```
>>> df = tree.arrays("p[xyz]", outputtype=pd.DataFrame)
```

- Outputtype can be any function accepting the number of branches extracted from the Tree. This can be a standalone function or a constructor.

Uproot: Custom output types

- The previous example could be done in the following way:

```
>>> def InvMass(E1, E2, px1, py1, pz1, px2, py2, pz2):
        return np.sqrt((E1 + E2)**2 - (px1 + px2)**2
                       - (py1 + py2)**2 - (pz1 + pz2)**2)

>>> mass = events.arrays(["E[12]", "p[xyz]*"], outputtype=InvMass)
>>> plt.hist(mass, bins=120, range=(0, 120), histtype='step')
```

Uproot: Caching data

- Every time you ask for arrays, uproot goes to the file and re-reads them
- For especially large arrays, this can take a long time
- For quicker access, uproot's array-reading functions have a cache parameter
- The cache only needs to behave like a dict (many third-party Python caches do)

```
>>> mycache = {}
>>> # first time: reads from file
>>> events.arrays(["p[xyz]*"], cache=mycache);
>>> # any other time: reads from cache
>>> events.arrays(["p[xyz]*"], cache=mycache);
```

- Uproot has an ArrayCache class that drops elements automatically when memory hits the defined limit

```
>>> events.arrays("*", cache=uproot.ArrayCache("100 kB"));
```

- All data sizes in uproot are specified as integers in bytes or a string with the appropriate unit (interpreted as powers of 1024, not 1000).

Uproot: Lazy Arrays

- TBranchMethods.array, TTreeMethods.array, or TTreeMethods.arrays read the file or cache immediately and returns an in-memory array
- For exploratory work or to control memory usage, you might want to let the data be read on demand
- TBranch.lazyarray, TTreeMethods.lazyarray, TTreeMethods.lazyarrays, and uproot.lazyarrays functions take most of the same parameters but return lazy array objects, rather than Numpy arrays
- These arrays load data chunks when requested and not all chunks need to be read into memory at any given time.
- These arrays can be used with Numpy's universal functions (ufuncs)
- This will usually cause the entire data to be loaded.
- By default, lazy arrays hold onto all data that have been read as long as the lazy array continues to exist.
- To use a lazy array as a window into a very large dataset, you'll have to limit how much it's allowed to keep in memory at a time using the caching mechanism described previously.

Uproot: Lazy Arrays as lightweight skims

- These arrays can be saved to files in a way that preserves their virtualness.
- This allows you to save a "diff" with respect to the original ROOT files
- Below, we load lazy arrays from a ROOT file with persistvirtual=True and add a derived feature

```
>>> data = events.lazyarrays(["E*", "p[xyz]*"], persistvirtual=True)
>>> data["mass"] = np.sqrt((data["E1"] + data["E2"])**2 -
                           (data["px1"] + data["px2"])**2 -
                           (data["py1"] + data["py2"])**2 -
                           (data["pz1"] + data["pz2"])**2)
```

- Then save the whole thing to an awkward-array file (.awkd)

```
>>> import awkward
>>> awkward.save("derived-feature.awkd", data, mode="w")
```

Uproot: Lazy Arrays as lightweight skims

- When we read it back, the derived features come from the awkward-array
- But the original features are loaded as pointers to the original ROOT files
- They know the original ROOT filenames... So don't move them!

```
>>> data2 = awkward.load("derived-feature.awkd")
>>> # reads from derived-feature.awkd
>>> data2["mass"]
>>> # reads from the original ROOT flies
>>> data2["E1"]
```

- Similarly, a dataset with a cut applied saves the identities of the selected events but only pointers to the original ROOT data
- This acts as a lightweight skim

```
>>> selected = data[data["mass"] < 80]
>>> awkward.save("selected-events.awkd", selected, mode="w")
>>> data3 = awkward.load("selected-events.awkd")
```

Uproot: Iteration

- Lazy arrays implicitly step through chunks of data to give you the impression that you have a larger array than memory can hold all at once
- The next two methods explicitly step through chunks of data, to give you more control over the process.
  TTreeMethods.iterate() iterates over chunks of a TTree
  uproot.iterate() iterates through files

```
>>> histogram = None
>>> for data in events.iterate(["E*","p[xyz]*"], namedecode="utf-8"):
...     # operate on a batch of data in the loop
...     mass = numpy.sqrt((data["E1"] + data["E2"])**2  ...)
...     # accumulate results
...     counts, edges = numpy.histogram(mass, bins=120, range=(0,120))
...     if histogram is None:
...         histogram = counts, edges
...     else:
...         histogram = histogram[0] + counts, edges
```

Uproot: TChain Alternatives

- functions uproot.lazyarray() and uproot.lazyarrays() allow you to make a lazy array that spans many files
- Can also use a file-spanning iterator (uproot.iterate()) to erase the difference between files. Note the iteration is over chunks of many events, not single events.
- These functions may be thought of as alternatives to ROOT's TChain

```
>>> uproot.lazyarray(
...     # list of files; local files can have wildcards (*)
...     ["http://scikit-hep.org/uproot/examples/sample-%s-zlib.root" % x
...         for x in ["5.23.02", "5.24.00", "5.25.02"]],
...     # TTree name in each file
...     "sample",
...     # branch(s) in each file for lazyarray(s)
...     "f8")
```

Uproot: Limiting the number of entries to read

- **All** array-reading functions have the following parameters:

  entrystart the first entry to read, by default 0

  entrystop one after the last entry to read, by default numentries

- As with Python slices, the entrystart and entrystop can be negative to count from the end of the TTree
- Setting entrystart and/or entrystop differs from slicing the resulting array as slicing reads, then discards entries, while these arguments minimize the data to read.
- Internally, ROOT files are written in chunks and whole chunks must be read, so the best places to set entrystart and entrystop are between basket boundaries.

```
>>> len(events.array("E1", entrystart=100, entrystop=300))
200
```

Uproot: Controlling lazy chunk and iteration step sizes

- In addition to entrystart and entrystop, the lazy array and iteration functions also have:

  entrysteps the number of entries to read in each chunk or step

- Can also set it to:

  numpy.inf to make the chunks/steps as big as possible (limited by file boundaries)

  memory size string e.g. "50 kB"

  list of (entrystart, entrystop) pairs e.g. [(0,12), (34-72)]

- Note the TTree lazy array/iteration functions use basket or cluster sizes as a default entrysteps, while multi-file lazy array/iteration functions use the maximum per file: numpy.inf

```
>>> [len(chunk) for chunk in events.lazyarrays(entrysteps=500).chunks]
[500, 500, 500, 500, 304]
>>> [len(data[b"E1"]) for data in events.iterate(["E*", "p[xyz]*"],
                                                 entrysteps=500)]
[500, 500, 500, 500, 304]
```

Uproot: Writing ROOT Files and Histograms

- Create a file using uproot.create(), uproot.recreate() or uproot.update() functions (corresponding to ROOT's "CREATE", "RECREATE", and "UPDATE" file modes)
- file objects can be used as a context manager
- Just as reading behaves like getting an object from a Python dict, writing behaves like putting an object into a Python dict
- Histograms can be written directly to the file
- Uproot recognizes Numpy histograms and converts them to ROOT histograms

```
>>> with ur.create("histograms.root") as f:
...         f["numpy"] = np.histogram(np.random.normal(0, 1, 10000))
...         f["numpy2d"] = np.histogram2d(np.random.normal(0, 1, 10000),
...                                       np.random.normal(0, 1, 10000))
```

Uproot: Writing ROOT Files and Histograms

- Create a file using uproot.create(), uproot.recreate() or uproot.update() functions (corresponding to ROOT's "CREATE", "RECREATE", and "UPDATE" file modes)
- file objects can be used as a context manager
- Just as reading behaves like getting an object from a Python dict, writing behaves like putting an object into a Python dict
- Histograms can be written directly to the file
- Uproot recognizes Numpy histograms and converts them to ROOT histograms

```
>>> f = ur.open("histograms.root")
>>> f.items()
[('numpy;1', <TH1I 'numpy' 0x0000073e62a0>),
 ('numpy2d;1', <TH2D 'numpy2d' 0x0000073e64b0>)]
```

Uproot: Writing ROOT Trees

- Uproot can write TTrees whose branches are basic types (integers and floating-point numbers)
- New trees are created with the uproot.newtree() function
- New branches can be created *implicitly* in the uproot.newtree() function or *explicitly* using uproot.newbranch()
- Write new baskets using the Tree.extend() method
- Remember to add entries to all the branches and the number of entries added to the branches is the same!
- You can also use the Tree.append() function to add baskets to your file if you need to just add a single value at the end of your current basket buffer
- Make sure to add entries to every branch, similar to the extend method

Uproot: Example Writing ROOT Trees

```
>>> branch = ur.newbranch(np.float64, title="This is the title")
>>> with ur.recreate("example.root") as f:
...     f["t"] = ur.newtree({"branch1": int,
...                          "branch2": np.int32,
...                          "branch3": branch})
...     f["t"].extend({"branch1": np.array([1, 2, 3, 4, 5]),
...                    "branch2": [6, 7, 8, 9, 10],
...                    "branch3": [1., 2., 3., 4., 5.]})
...     f["t"].append({"branch1": 1, "branch2": 2, "branch3": 7.0})
```

Uproot: Example Writing ROOT Trees

```
>>> with ur.open("example.root") as f:
...         f["t"].arrays("branch*")
{b'branch1': array([1, 2, 3, 4, 5, 1]),
 b'branch2': array([ 6,  7,  8,  9, 10, 2], dtype=int32),
 b'branch3': array([1., 2., 3., 4., 5., 7.])}
root [0] TFile f("example.root")
root [1] TTree* tree = (TTree*)f.GetObjectChecked("t", "TTree")
root [3] tree->Scan()
*  Row * branch1 *  branch2 *  branch3 *
*    0 *       1 *       6 *       1 *
*    1 *       2 *       7 *       2 *
*    2 *       3 *       8 *       3 *
*    3 *       4 *       9 *       4 *
*    4 *       5 *      10 *       5 *
*    5 *       1 *       2 *       7 *
```

MINUIT for Python: iminuit

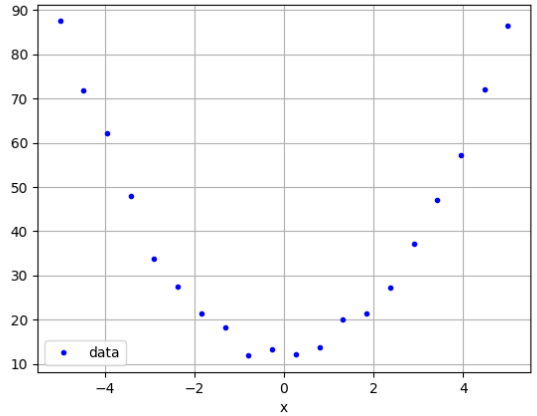
> *iminuit is a Python interface to the MINUIT C++ package. It can be used as a general robust function minimisation method, but is most commonly used for likelihood fits of models to data, and to get model parameter error estimates from likelihood profile analysis.*
>
> *iminuit.readthedocs.org*

```
>>> from iminuit import Minuit
```

IMinuit: Curve Fitting (Least-Squares)

Fit a Parabola of form $ax^2 + b + \epsilon$

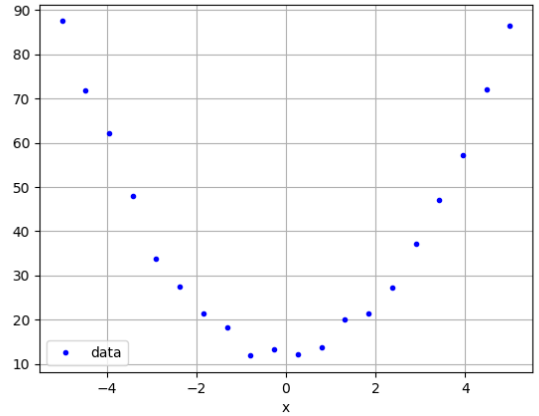
```
>>> xdata = np.linspace(-5, 5, 20)
>>> ydata = 3*xdata**2 + 12
+ np.random.normal(scale=1.5, size=xdata.size)
>>> plt.plot(xdata, ydata, 'b.')
```



IMinuit: Curve Fitting (Least-Squares)

Fit a Parabola of form $ax^2 + b + \epsilon$

```
>>> xdata = np.linspace(-5, 5, 20)
>>> ydata = 3*xdata**2 + 12
+ np.random.normal(scale=1.5, size=xdata.size)
>>> plt.plot(xdata, ydata, 'b.')
```



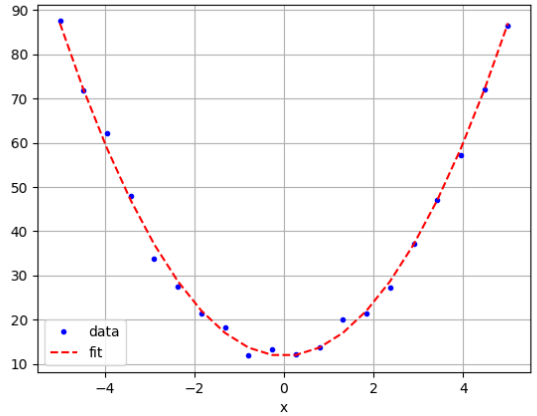

```
>>> def parabola(x, a, b):
...         return a*x**2 + b
>>> def least_squares(a, b):
...         return sum((ydata - parabola(xdata, a, b))** 2)
>>> m = Minuit(least_squares, a=5, b=5, # keywords inferred from func
                error_a=0.1, error_b=0.1, errordef=1)
```

- error_a & error_b are the initial step size. Name inferred from func
- errordef = 1 for OLS, = 0.5 for negative log likelihood functions
- errordef is needed to get correct uncertainty estimates for parameters

IMinuit: Curve Fitting (Least-Squares)

Fit a Parabola of form $ax^2 + b + \epsilon$

```
>>> xdata = np.linspace(-5, 5, 20)
>>> ydata = 3*xdata**2 + 12
+ np.random.normal(scale=1.5, size=xdata.size)
>>> plt.plot(xdata, ydata, 'b.')
```




```
>>> def parabola(x, a, b):
...     return a*x**2 + b
>>> def least_squares(a, b):
...     return sum((ydata - parabola(xdata, a, b))** 2)
>>> m = Minuit(least_squares, a=5, b=5, # keywords inferred from func
                error_a=0.1, error_b=0.1, errordef=1)
>>> m.migrad()
>>> plt.plot(xdata, parabola(xdata, *m.np_values()), 'r--')
```

IMinuit: Limiting and Fixing Parameters

- Can specify limits for parameters using the **limit_<name>** keyword arg where <name> is inferred from the function args.

  lower limit set to (<value>, None) or (<value>, float("infinity"))

  upper limit set to (None, <value>) or (-float("infinity"), <value>)

  two-sided limit set to (<min_value>, <max_value>)

- Can fix a parameter by setting the **fix_<name>** keyword arg to True. Again the <name> is inferred from the function

Example

```
>>> m = Minuit(least_squares,
               a=5, b=12,
               error_a=0.1, error_b=0.1,
               fix_b=True,
               limit_a=(0, 10),
               errordef=1)
```

IMinuit: A Note on Vectorised functions

You can also use iminuit with functions that accept numpy arrays (i.e. a single argument containing each param in an array rather than separate args per param). This has pros and cons.

Pros:

- Easy to change number of fitted parameters
- Sometimes simpler function body that's easier to read
- Technically more efficient, although this is probably not noticable unless you have >100 parameters

Cons:

- IMinuit cannot figure out names for each parameter

IMinuit: A Note on Vectorised functions

- Use **Minuit.from_array_func** when creating the iminuit instance.
- Initial values are given as a tuple after the function with length N_{params}
- Keyword args **error_<name>** and **fix_<name>** drop the _<name> and take tuples of length N_{params}
- Limits keyword **limit_<name>** also drops _<name> but is a list of size N_{params} containing tuples of (<min_value>, <max_value>)
- By default the parameters are named $x1, ..., xN$. Can specify names using the **name** keyword which is also a tuple of length N_{params}

Vectorised version of previous example

```
>>> def least_squares_np(par):   # par is a numpy array here
...         a, b = par
...         return np.sum( (ydata - np.polyval([a, 0, b], xdata))** 2)
>>> m = Minuit.from_array_func(least_squares_np, (5, 12),
                               error =(0.1, 0.1), fix =(False, True),
                               limit =[(0, 10), (None, None)],
                               name=("a", "b"), errordef =1)
```

IMinuit: Minimising the function

- To run the actual minimization, you call the **Minuit.migrad()** method
- Migrad performs Variable-Metric Minimization
- It combines a steepest-descends algorithm along with line search strategy
- Popular because of its robustness
- **Minuit.migrad()** returns **MigradResult** consisting of two objects, one contains metadata about the function minimum (**FMin**). The second is the parameter list. This list contains **Param** objects, one for each parameter.
- The FMin object can be got at any time thereafter using the **Minuit.get_fmin()** method

```
>>> m.get_fmin()
```

- The most important attribute of the FMin object is **is_valid**. If this is false, the fit did not converge and the result is useless. There can be many reasons, including:
  - The fit function is not analytical everywhere in the parameter space or does not have a local minimum
  - Migrad reached the call limit before the convergence

IMinuit: Minimising the function

- If you are interested in parameter uncertainty, you should also make sure that:
  - **has_covariance**, **has_accurate_covar**, and **has_posdef_covar** are True
  - **has_made_posdef_covar** and **hesse_failed** are False
- The Param objects contain metadata about the parameters. The most important attributes are:

index	The parameter's index
name	The paramete's name
value	The value of the parameter at minimum
error	Uncertainty estimate for the parameter value

- The Params list can be got at any time using the **Minuit.get_param_states()** method like so:

```
>>> m.get_param_states()
```

IMinuit: Parameter Uncertainties

- In the previous slide we saw how to get individual parameters uncertainties.
- These don't contain any covariance information.
- Minuit offers two ways to compute the parameter uncertainties, **Hesse** (symmetric) and **Minos** (asymmetric)
- The Migrad algorithm computes an approximation of the Hesse matrix automatically during minimization
- When the default strategy is used, Minuit does a check whether this approximation is sufficiently accurate and if not, it computes the Hesse matrix automatically
- All this happens inside C++ Minuit and is a bit opaque, it is recommended to call **Minuit.hesse()** explicitly after the minimization if exact errors are important, like so:

```
>>> m.hesse()
```

- This updates and returns the parameter list

IMinuit: Parameter Uncertainties

- To see the covariance matrix, use the **Minuit.matrix()** method.
- Use the same method for the correlation matrix with the addition of the boolean **correlation** keyword

```
>>> m.matrix()   # covariance matrix
>>> m.matrix( correlation =True)   # correlation matrix
```

- Note Minuit cannot accurately minimise the function if two parameters are (almost) perfectly (anti-)correlated
- You can get these matricies as NumPy arrays if you prefer, using the **Minuit.np_matrix()** method in an analogous way
- Minos is not automatically called during minimization, it needs to be called explicitly afterwards, like so

```
>>> m.minos()
```

- After calling Minos, the parameter list is updated to show the Minos errors

IMinuit: Minimisation Results

```
>> fmin, (a, b) = m.migrad()
>> print(a.name, a.value, a.error)
a 3.04 0.03
>> print(b.name, b.value, b.error)
b 11.3 0.3

>>> (a, b) = m.hesse()
>>> print(a.name, a.value, a.error)
a 3.04 0.03
>>> print(b.name, b.value, b.error)
b 11.3 0.3

>>> for name, param in m.minos().items():
...     print(name, param.min, param.lower, param.upper)
a 3.04 -0.03 0.03
b 11.3 -0.3 0.3
```

IMinuit: Minimisation Results

```
>>> print(m.get_param_states())
```

	Name	Value	Hesse Err	Minos Err-	Minos Err+	Limit-	Limit+	Fixed
0	a	3.04	0.03	-0.03	0.03	0	10	
1	b	11.3	0.3	-0.3	0.3			yes

```
# If we extract the Param objects from the Param list
# we can't get at Minos errors
>>> a, b = m.get_param_states()
>>> print(a.name, '=', a.value, '+-', a.error)
a = 3.04 +- 0.03

>>> merrors = m.get_param_states().merrors
>>> b = merrors['b']
>>> print(b.name, '=', b.min, b.lower, b.upper)
b = 11.3 -0.3 0.3

>>> print(m.matrix()[0][0])
0.0009
```

IMinuit: Quick Access to Minimisation Results

- We have seen that we can access the parameters through **Minuit.get_param_states()**
- We have seen that we can access the Hesse covariance matrix via **Minuit.matrix()**
- We can also access the parameters through the return values of the **Minuit.migrad()**, **Minuit.hesse()** and **Minuit.minos()** methods
- In addition **Minuit** objects provide the following attributes that return dict-like objects:

values	dict-like view of the parameter values
errors	dict-like view of symmetric uncertainties
merrors	asymmetric uncertainties
covariance	the covariance matrix computed by Hesse

- There also exists corresponding *methods* **Minuit.np_values()**, **Minuit.np_errors()**, **Minuit.np_merrors** & **Minuit.np_covariance()** which return the above as NumPy arrays.
- These NumPy methods return copies of the parameter metadata rather than views as returned by the attributes

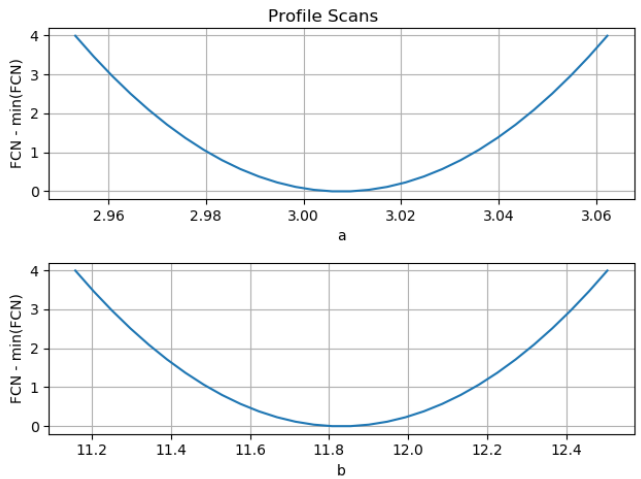
IMinuit: Quick Access to Minimisation Results

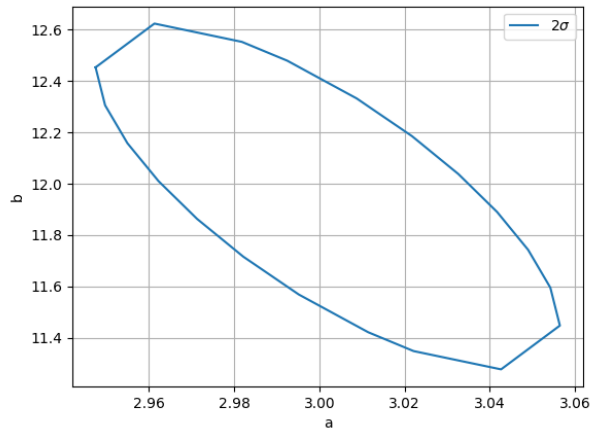
```
>>> print(m.values[0], "+-", m.errors[0]) # access by index
3.04 +- 0.03
>>> print(m.values['a'], "+-", m.errors['a']) # access by name
3.04 +- 0.03
>>> print(m.values['b'], m.merrors[('b', -1)], m.merrors[('b', 1)])
11.3 -0.3 0.3
>>> # you can also iterate over the view like you would over a dict
>>> for key, value in m.values.items():
...     print(key, value)
a 3.04
b 11.3
>>> print(m.covariance[('a', 'a')], m.covariance[('b', 'b')])
0.0009 0.09
# With the NumPy Minos errors, Note the column-wise format
>>> print(m.np_merrors())  # This is used by matplotlib errorbar
array([[0.03, 0.3],         # array([abs(a_down), abs(b_down)],
       [0.03, 0.3]])        #       [   a_up   ,    b_up    ])
```

IMinuit: Plotting

- Can get 1D parameter profile information and contour information using *mnprofile* and *mncontour* methods e.g.

```
>>> f1, (ax1, ax2, ax3) = plt.subplots(3)
>>> ax1.plot(*m.mnprofile('a',
                          subtract_min=True)[:-1])
>>> ax2.plot(*m.mnprofile('b',
                          subtract_min=True)[:-1])
>>> _, _, cont = m.mncontour('a','b',sigma=2)
>>> ax3.plot(*np.hsplit(np.array(cont), 2))
```

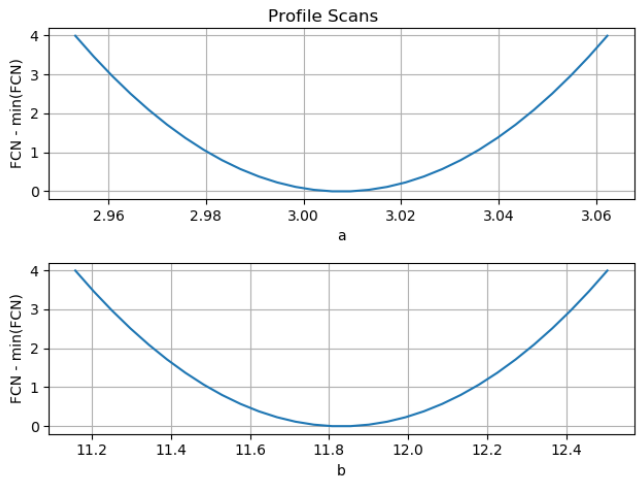




IMinuit: Plotting

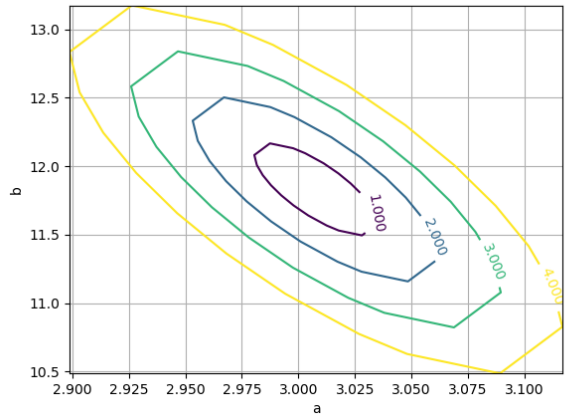
- Can get 1D parameter profile information and contour information using *mnprofile* and *mncontour* methods e.g.

```
>>> f1, (ax1, ax2, ax3) = plt.subplots(3)
>>> ax1.plot(*m.mnprofile('a',
                          subtract_min=True)[:-1])
>>> ax2.plot(*m.mnprofile('b',
                          subtract_min=True)[:-1])
>>> _, _, cont = m.mncontour('a','b',sigma=2)
>>> ax3.plot(*np.hsplit(np.array(cont), 2))
```



- Convenience wrappers *draw_mnprofile* and *draw_mncontour* do the plotting for you

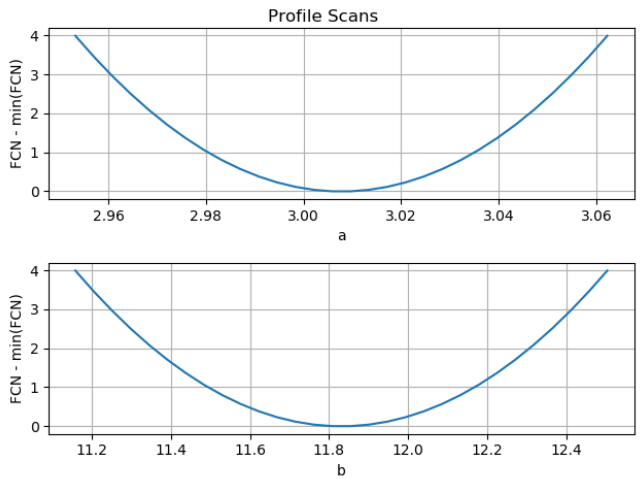
```
>>> m.draw_mncontour('a', 'b', nsigma=4)
```



IMinuit: Plotting

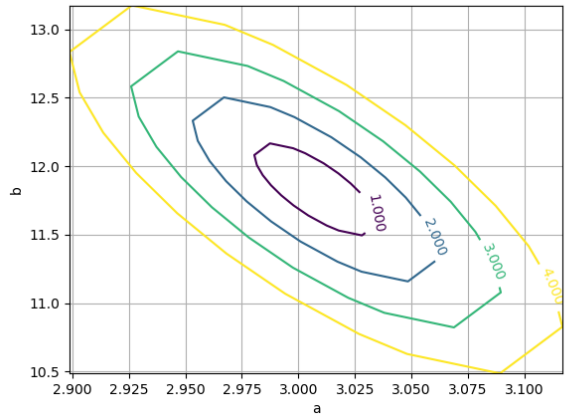
- Can get 1D parameter profile information and contour information using *mnprofile* and *mncontour* methods e.g.

```
>>> f1, (ax1, ax2, ax3) = plt.subplots(3)
>>> ax1.plot(*m.mnprofile('a',
                          subtract_min=True)[:-1])
>>> ax2.plot(*m.mnprofile('b',
                          subtract_min=True)[:-1])
>>> _, _, cont = m.mncontour('a','b',sigma=2)
>>> ax3.plot(*np.hsplit(np.array(cont), 2))
```



- Convenience wrappers *draw_mnprofile* and *draw_mncontour* do the plotting for you

```
>>> m.draw_mncontour('a', 'b', nsigma=4)
```



- Corresponding functions without the **mn** prefix exist, note these are not sigmas, just contours of the fit func

Machine Learning in Python: scikit-learn

> ***scikit-learn*** *- Machine Learning in Python*
>
> - *Simple and efficient tools for data mining and data analysis*
> - *Accessible to everybody, and reusable in various contexts*
> - *Built on NumPy, SciPy, and matplotlib*
> - *Open source, commercially usable - BSD license*
>
> *scikit-learn.org*

```
>>> import sklearn as skl
```

Note

What follows is a brief introduction to using machine learning in Python and should in no way be considered a substitute for a definitive course such as that provided by Yandex.

scikit-learn: Terminology

- Machine learning is about learning some properties of a data set and then testing those properties against another data set
- A common practice in machine learning is to evaluate an algorithm by splitting a data set into two

  training set on which we learn some properties

  testing set on which we test the learned properties
- If each entry (**sample**) is more than just a single number, i.e. a multi-dimensional entry (multivariate data), it is said to have several attributes or **features**
- learning problems fall into two main categories:

  supervised learning in which the data comes with additional attributes (**target**) that we want to predict.

  unsupervised learning in which the training data consists of a set of input vectors x without any corresponding target values

scikit-learn: Terminology

We can further divide these categories as follows:

- **Supervised Learning**

  classification samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data.

  regression if the desired output consists of one or more continuous variables, then the task is called regression.

- **Unsupervised Learning**

  clustering the goal in such problems may be to discover groups of similar examples within the data

  density estimation to determine the distribution of data within the input space

scikit-learn: Example datasets

Classification
load_iris()
load_digits()
load_wine()
load_breast_cancer()
Regression
load_boston()
load_diabetes()

- scikit-learn comes with a few standard datasets, loaded through functions in the **sklearn.datasets** module
- A dataset is a dictionary-like object that holds all the data and some metadata about the data.
- This data is stored in the **.data** member, which is a n_samples, n_features array
- In the case of supervised problem, one or more response variables are stored in the **.target** member
- A detailed description of the dataset and it's features can be found by printing the **DESCR** member

scikit-learn: Simple Classification Example - digits

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.svm import SVC
>>> digits = load_digits()
>>> clf = SVC(gamma=0.001, C=100.)
```

- First we load the digits dataset and create an estimator, in this case SVC which implements support vector classification
- The estimator's constructor takes as arguments the model's parameters which for now we manually guess

scikit-learn: Simple Classification Example - digits

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.svm import SVC
>>> digits = load_digits()
>>> clf = SVC(gamma=0.001, C=100.)
>>> clf.fit(digits.data[:-1], digits.target[:-1])
```

- First we load the digits dataset and create an estimator, in this case SVC which implements support vector classification
- The estimator's constructor takes as arguments the model's parameters which for now we manually guess
- The clf (for classifier) estimator instance is first **fitted** to the model; that is, it must *learn* from the model
- For our **training** set we use all but one of our samples

All estimators expose a **fit** method that takes a dataset (usually a 2-d array)

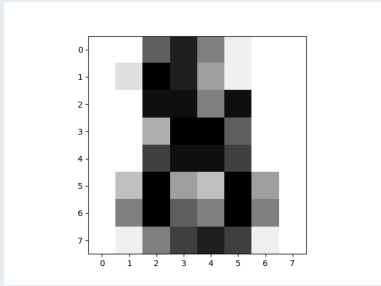
scikit-learn: Simple Classification Example - digits

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.svm import SVC
>>> digits = load_digits()
>>> clf = SVC(gamma=0.001, C=100.)
>>> clf.fit(digits.data[:-1], digits.target[:-1])
>>> clf.predict(digits.data[-1:])
array([8])
```

- First we load the digits dataset and create an estimator, in this case SVC which implements support vector classification
- The estimator's constructor takes as arguments the model's parameters which for now we manually guess
- The clf (for classifier) estimator instance is first **fitted** to the model; that is, it must *learn* from the model
- For our **training** set we use all but one of our samples
- We can now **predict** new values by determining the image from the training set that best matches the last image

scikit-learn: Simple Classification Example - digits

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.svm import SVC
>>> digits = load_digits()
>>> clf = SVC(gamma=0.001, C=100.)
>>> clf.fit(digits.data[:-1], digits.target[:-1])
>>> clf.predict(digits.data[-1:])
array([8])
```



- First we load the digits dataset and create an estimator, in this case SVC which implements support vector classification
- The estimator's constructor takes as arguments the model's parameters which for now we manually guess
- The clf (for classifier) estimator instance is first **fitted** to the model; that is, it must *learn* from the model
- For our **training** set we use all but one of our samples
- We can now **predict** new values by determining the image from the training set that best matches the last image

scikit-learn: Model persistence

- It is possible to save a model in scikit-learn by using Python's built-in persistence model, pickle

```
>>> import pickle
>>> clf2 = pickle.loads(pickle.dumps(clf))
>>> clf2.predict(digits.data[-1:])
array([8])
```

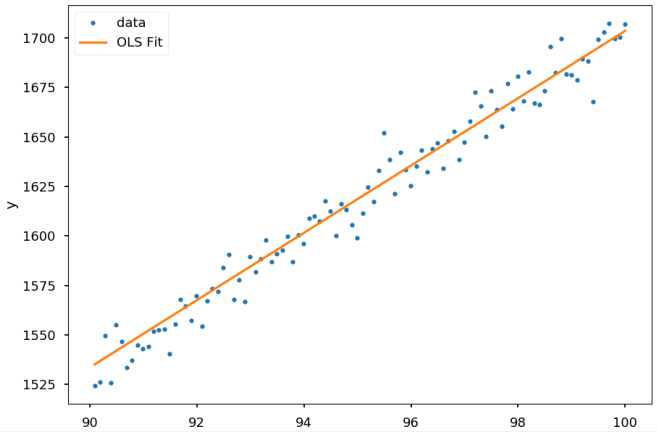
- In the specific case of scikit-learn, it may be more interesting to use joblib's replacement for pickle, which is more efficient on big data but it can only pickle to the disk

```
>>> from joblib import dump, load
>>> dump(clf, 'filename.joblib')
```

- Later, you can reload the pickled model (possibly in another Python process) with:

```
>>> clf = load('filename.joblib')
```

scikit-learn: Regression Example - Linear Regression



- LinearRegression, in its simplest form, fits a linear model to the data set by adjusting a set of parameters in order to make the sum of the squared residuals of the model as small as possible.
- Linear Model: $y = X\beta + \epsilon$

```
>>> x = np.linspace(0,100,1000).reshape(-1,1)
>>> y = 17*x + 3 + 10*np.random.randn(1000).reshape(-1,1)
>>> from sklearn.linear_model import LinearRegression
>>> regr = LinearRegression()
>>> regr.fit(x[:900], y[:900])
>>> coef, intercept = regr.coef_[0,0], regr.intercept_[0]
model: y = 17.007463184253808 x + 2.8429478076877785
>>> np.mean((regr.predict(x[900:]) - y[900:])*2)
mean square error = 2.2645812970629233
>>> regr.score(x[900:], y[900:])
score = 0.9701062883580784 #1=perfect pred. 0=no linear relationship
```

scikit-learn: Multivariate linear regression

The diabetes dataset

- The diabetes dataset consists of 10 physiological variables (age, sex, weight, blood pressure, etc.) measured on 442 patients, and an indication of disease progression after one year
- The task at hand is to predict disease progression from physiological variables.

```
>>> diabetes = load_diabetes()
>>> xtest, ytest = diabetes.data[-20:], diabetes.target[-20:]
>>> xtrain, ytrain = diabetes.data[:-20], diabetes.target[:-20]
>>> regr = LinearRegression()
>>> regr.fit(xtrain, ytrain)
>>> print(regr.coef_) # Note 10 co- efficients as 10 features
[ 3.03499549e-01 -2.37639315e+02 5.10530605e+02  3.27736980e+02
 -8.14131709e+02  4.92814588e+02  1.02848452e+02  1.84606489e+02
  7.43519617e+02  7.60951722e+01]
```

scikit-learn: Score and cross-validation

- As we have seen, every estimator exposes a **score** method that can judge the quality of the fit (or the prediction) on new data.

```
>>> digits = load_digits()
>>> svc = SVC(C=1)
>>> print(svc.fit(digits.data[:-100], digits.target[:-100])
                .score(digits.data[-100:], digits.target[-100:]))
0.98
```

- To get a better measure of prediction accuracy we can successively split the data in folds that we use for training and testing

```
>>> x = np.array(np.array_split(digits.data, 3))
>>> y = np.array(np.array_split(digits.target, 3))
>>> mask = np.arange(3)
>>> for i in range(3):
...     xtrain, ytrain = x[mask!=i][0], y[mask!=i][0]
...     xtest, ytest = x[mask==i][0], y[mask==i][0]
...     print(svc.fit(xtrain, ytrain).score(xtest, ytest), end=', ')
0.9215358931552587, 0.9549248747913188, 0.9348914858096828,
```

scikit-learn: Score and cross-validation

- As we have seen, every estimator exposes a **score** method that can judge the quality of the fit (or the prediction) on new data.

```
>>> digits = load_digits()
>>> svc = SVC(C=1)
>>> print(svc.fit(digits.data[:-100], digits.target[:-100])
                .score(digits.data[-100:], digits.target[-100:]))
0.98
```

- To get a better measure of prediction accuracy we can successively split the data in folds that we use for training and testing
- This is called a KFold cross-validation

scikit-learn: Cross-validation generators

- Scikit-learn has a collection of classes which can be used to generate lists of train/test indices for popular cross-validation strategies
- They expose a **split** method which accepts the input dataset to be split and yields the train/test set indices for each iteration of the chosen cross-validation strategy
- The cross-validation can then be performed easily

```
>>> from sklearn.model_selection import KFold
>>> k_fold = KFold(n_splits=3)
>>> print([svc.fit(digits.data[train], digits.target[train])
                .score(digits.data[test], digits.target[test])
            for train, test in k_fold.split(digits.data)])
[0.9348914858096828, 0.9565943238731218, 0.939899833055091 8]
```

scikit-learn: Cross-validation generators

- Scikit-learn has a collection of classes which can be used to generate lists of train/test indices for popular cross-validation strategies
- They expose a **split** method which accepts the input dataset to be split and yields the train/test set indices for each iteration of the chosen cross-validation strategy
- The cross-validation can then be performed easily
- The cross-validation score can be directly calculated using the **cross_val_score** helper. Given an estimator, the cross-validation object and the input dataset, the cross_val_score splits the data repeatedly into a training and a testing set, trains the estimator using the training set and computes the scores based on the testing set for each iteration of cross-validation.

```
>>> from sklearn.model_selection import cross_val_score
>>> print(cross_val_score(svc, digits.data, digits.target,
                          cv=k_fold, n_jobs=-1))
[0.93489149 0.95659432 0.93989983]
```

scikit-learn: Grid-search and model parameter selection.

- scikit-learn provides an object (**GridSearchCV**) that, given data, computes the score during the fit of an estimator on a parameter grid and chooses the parameters to maximize the cross-validation score
- This object takes an estimator during the construction and exposes an estimator API

```
>>> from sklearn.model_selection import GridSearchCV, train_test_split
>>> xtrain, xtest, ytrain, ytest = train_test_split(digits.data,
                                                    digits.target,
                                                    test_size=0.33,
                                                    random_state=42)
>>> param_grid = {'C': np.logspace(0, 5, 10),
                  'gamma': np.logspace(-4, -1, 10)}
>>> clf = GridSearchCV(svm.SVC(kernel='linear'), param_grid, cv=5,
                       iid=False, n_jobs=-1)
>>> clf.fit(xtrain, ytrain)
>>> print("Best train score = ", clf.best_score_) # 0.9683221116260136
>>> print("Test score = ", clf.score(xtest, ytest)) # 0.979797979797798
```

scikit-learn: Unsupervised learning - clustering iris dataset

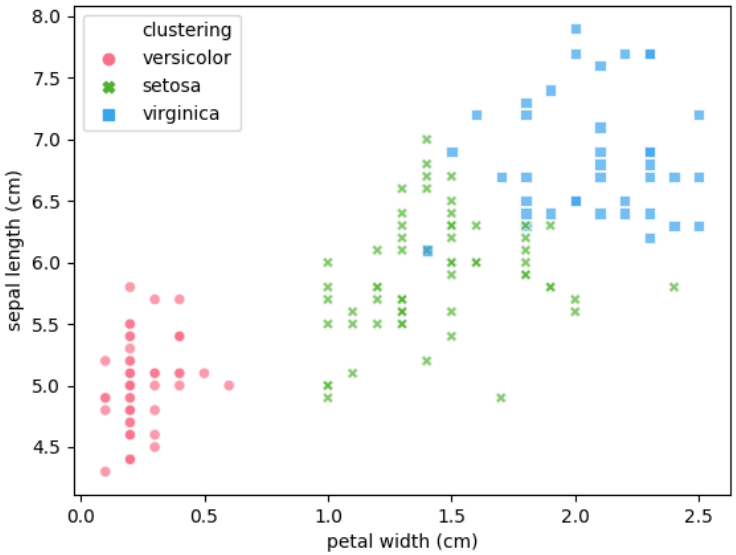
The Problem

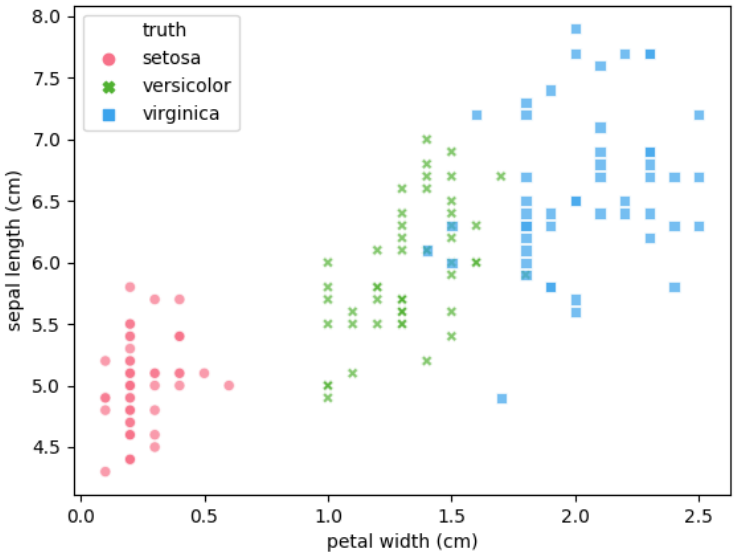
Given the iris dataset, if we knew that there were 3 types of iris, but did not have access to the **target** labels: we could try a clustering task: split the observations into well-separated group called **clusters**.

- There exist a lot of different clustering criteria and associated algorithms
- The simplest clustering algorithm is **K-means**

```
>>> from sklearn import cluster
>>> iris = load_iris()
>>> k_means = cluster.KMeans(n_clusters=3)
>>> k_means.fit(iris.data)
>>> print(k_means.labels_[::10])
[1 1 1 1 1 0 0 0 0 0 2 2 2 2 2]
>>> print(iris.target[::10])
[0 0 0 0 0 1 1 1 1 1 2 2 2 2 2]
```

scikit-learn: Unsupervised learning - clustering iris dataset





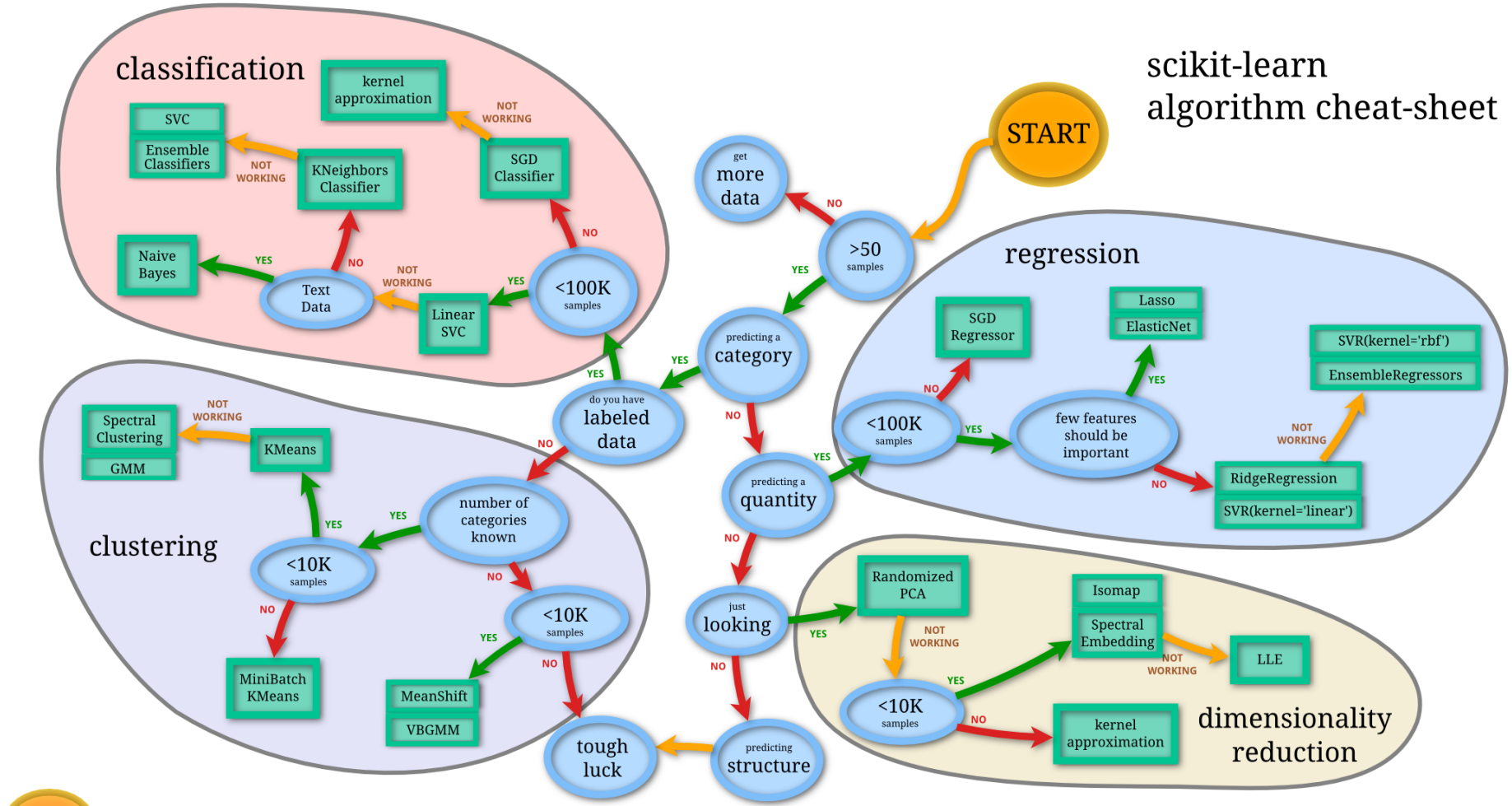
Warning

There is absolutely no guarantee of recovering a ground truth:

1. Choosing the right number of clusters is hard.
2. The algorithm is sensitive to initialization, and can fall into local minima, although scikit-learn employs several tricks to mitigate this issue.

Don't over-interpret clustering results

scikit-learn: Choosing the right estimator



The End....Or just the beginning

- Now do Worksheet2:
  http://www.hep.ph.ic.ac.uk/ arichard/pgtasks/Worksheet2.ipynb