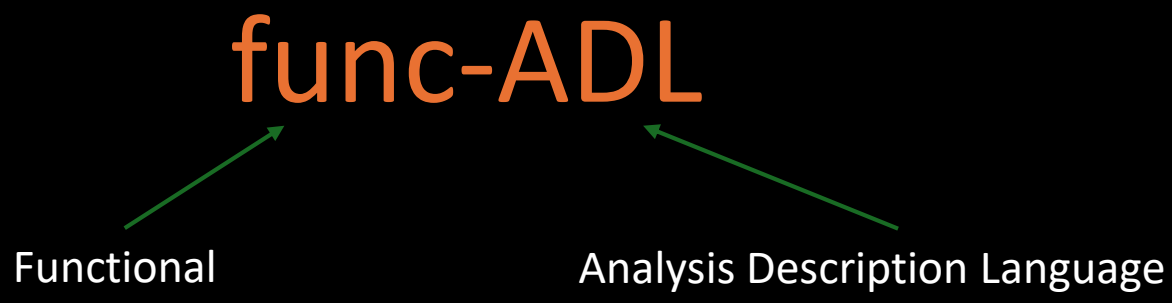
func_adl Status

G. Watts (UW/Seattle)

2024-11-05

What Does func-ADL do?



Functional: as in the programming paradigm.

- Can be expressed in multiple programming languages.
- Does not require special syntax – uses the host programming language
- Based on Microsoft Research's LINQ. Has ended up in C#, Javascripot, F#, etc.
- LINQ: **L**anguage **In**tegrated **Q**uery
- Monads!

Data processing features of a ADL:

- Group, ungroup data
- Aggregate
- Split
- Select
- Transform
- Extensible, though this isn't trivial to do.

IRIS-HEP func_ADL implementation

All python, all the time…

```
jets = (ds
        .SelectMany(lambda e: e.Jets())
        .Where(lambda j: (j.pt() / 1000) > 30)
        .Select(lambda j: j.pt() / 1000.0)
        .AsAwkwardArray('JetPt')
```

Transform & concatenate:
- Transforms the event into a list of jets
- Concatenates the jets into a single list (removes the event boundary)

Filters: Removes jets less than 30 GeV

Transforms: Converts the p_T to GeV

Result: Asks for jets as an in-memory awkward array

Using it with ServiceX 3.0 Front End

Func_adl

ServiceX Spec and Delivery

```python
from servicex import query as q, deliver, dataset


def func_adl_xaod_simple():
    query = q.FuncADL_ATLASr22()  # type: ignore
    jets_per_event = query.Select(lambda e: e.Jets('AnalysisJets'))
    jet_info_per_event = jets_per_event.Select(
        lambda jets: {
            'pt': jets.Select(lambda j: j.pt()),
            'eta': jets.Select(lambda j: j.eta())
        }
    )

    spec = {
        'Sample': [{
            'Name': "func_adl_xAOD_simple",
            'Dataset': dataset.FileList(
                [
                    "root://eospublic.cern.ch//eos/opendata/atlas/rucio/mc20_13TeV/DAOD_PHYSLIT
                ]
            ),
            'Query': jet_info_per_event
        }]
    }
    files = deliver(spec, servicex_name="servicex-uc-af")
    assert files is not None, "No files returned from deliver! Internal error"
    return files
```

The Code

Front End

- Func_adl – Code that wraps up the python code and query along with type information.
- ServiceX 3.0 frontend Library – glue between ServiceX and a func_adl query (see prev slide).
  - Takes the wrapped up query and ships it to ServiceX
- func-adl-types-atlas Loads a AnalysisBase container, uses ROOT to scan the xAOD type schema, and generates a yaml file with the type specifications
- func_adl_servicex_type_generator Generates python type-shed files given a yaml type description
- func_adl_type_generator Scripts and CI that automates building type files

Back End

- func_adl_xAOD Converts query & type information into C++ to run on EventLoop or CMS's miniAOD framework.
- func_adl_uproot Converts query into uproot code to run directly on ntuple's (anything uproot can deal with).

Type Generator

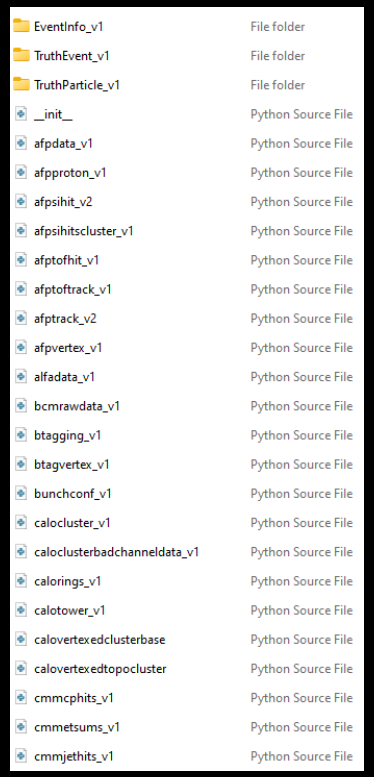
```
'p4': {
    'metadata_type': 'add_method_type_info',
    'type_string': 'xAOD::Jet_v1',
    'method_name': 'p4',
    'return_type': 'TLorentzVector',
},
'px': {
    'metadata_type': 'add_method_type_info',
    'type_string': 'xAOD::Jet_v1',
    'method_name': 'px',
    'return_type': 'float',
},
'py': {
    'metadata_type': 'add_method_type_info',
    'type_string': 'xAOD::Jet_v1',
    'method_name': 'py',
    'return_type': 'float',
},
'pz': {
    'metadata_type': 'add_method_type_info',
    'type_string': 'xAOD::Jet_v1',
    'method_name': 'pz',
    'return_type': 'float',
},
'getConstituents': {
    'metadata_type': 'add_method_type_info',
    'type_string': 'xAOD::Jet_v1',
    'method_name': 'getConstituents',
    'return_type_element': 'xAOD::JetConstituent*',
    'return_type_collection': 'xAOD::JetConstituentVector',
},
```

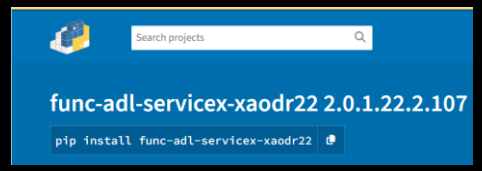
Type information

Type shed files

```
@func_adl_callback(_add_method_metadata)
class Jet_v1:
    "A class"


    def pt(self) -> float:
        "A method"
        ...

    def eta(self) -> float:
        "A method"
        ...

    def phi(self) -> float:
        "A method"
        ...

    def m(self) -> float:
        "A method"
        ...

    def e(self) -> float:
        "A method"
        ...

    def rapidity(self) -> float:
        "A method"
        ...

    def p4(self) -> func_adl_servicex_xaodr22.tlorentzvector.TLorentzVector:
        "A method"
        ...
```

Version 2.0.1 of the typeshed file





- Allows an IDE to help you with types
- Informs backends of types (or C++ errors!)
- Catches errors at the client vs on the backend

G. Watts (UW/Seattle)

Team

- Gordon Watts – lead, PI, xAOD backend, most of the front end, etc.
- Mason Proffitt – Graduate student, uproot backend, graduates in March
- Roger Janusiak – Graduate student, joining in January
- Artur Cordeiro Oudot Choi – Postdoc joining December 1st
  - Will discuss his contributions over the next month

Status

Most front-end code is now in stable production

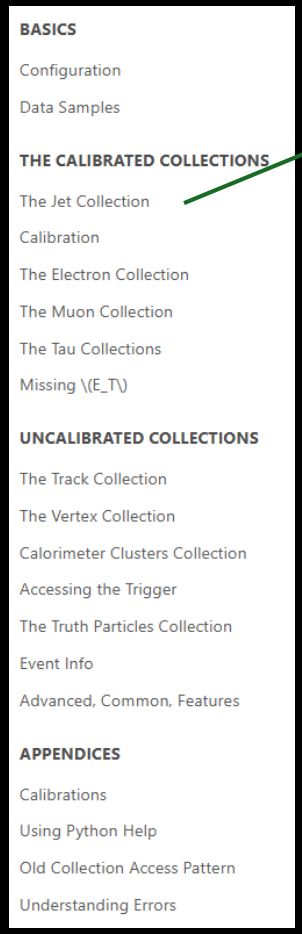
- We need to finish moving everything to 3.13
- Mason recently discovered some bugs that prevent some idioms from being processed correctly
- There are a number of new features we are looking at
  - But none of them are high priority right now
- The backend code is also in production
  - As we get new requests we add new features – but this is fairly slow right now.

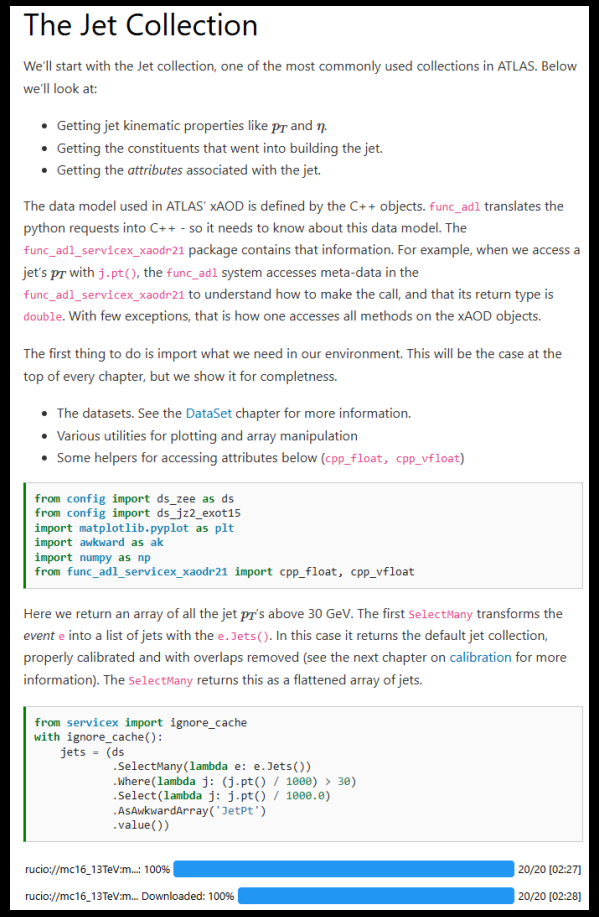
Biggest Friction:

- Documentation!!!
- Real use cases to drive further development

Documentation

The most extensive documentation is for xaod usages (github)





The Jet Collection

We'll start with the Jet collection, one of the most commonly used collections in ATLAS. Below we'll look at:

- Getting jet kinematic properties like p_T and η.
- Getting the constituents that went into building the jet.
- Getting the *attributes* associated with the jet.

The data model used in ATLAS' xAOD is defined by the C++ objects. `func_adl` translates the python requests into C++ - so it needs to know about this data model. The `func_adl_servicex_xaodr21` package contains that information. For example, when we access a jet's p_T with `j.pt()`, the `func_adl` system accesses meta-data in the `func_adl_servicex_xaodr21` to understand how to make the call, and that its return type is `double`. With few exceptions, that is how one accesses all methods on the xAOD objects.

The first thing to do is import what we need in our environment. This will be the case at the top of every chapter, but we show it for completness.

- The datasets. See the DataSet chapter for more information.
- Various utilities for plotting and array manipulation
- Some helpers for accessing attributes below (`cpp_float, cpp_vfloat`)

```
from config import ds_zee as ds
from config import ds_jz2_exot15
import matplotlib.pyplot as plt
import awkward as ak
import numpy as np
from func_adl_servicex_xaodr21 import cpp_float, cpp_vfloat
```

Here we return an array of all the jet p_T's above 30 GeV. The first `SelectMany` transforms the *event* `e` into a list of jets with the `e.Jets()`. In this case it returns the default jet collection, properly calibrated and with overlaps removed (see the next chapter on calibration for more information). The `SelectMany` returns this as a flattened array of jets.

```
from servicex import ignore_cache
with ignore_cache():
    jets = (ds
            .SelectMany(lambda e: e.Jets())
            .Where(lambda j: (j.pt() / 1000) > 30)
            .Select(lambda j: j.pt() / 1000.0)
            .AsAwkwardArray('JetPt')
            .value())
```

rucio://mc16_13TeV:m...: 100% 20/20 [02:27]

rucio://mc16_13TeV:m... Downloaded: 100% 20/20 [02:28]

A JupyterBook of active documentation

Good:

- Quite extensive
- Looks at each use-case in isolation
- Describes how to calibrate an xAOD

Bad:

- All documentation is based around R21
- PHYSLITE is much easier to use (so well behind the current state of the code)
- Does not show how to use complex cases
- Does not introduce func-adl

Hopeful:

- Roger Janusiak (grad student) joins IRIS-HEP in January and has already started exploring this repo
  - R22 and PHYSLITE is his first task

G. Watts (UW/Seattle)

More Complex Use Cases

Building a script that will extract a Long-Lived Particle DNN training dataset from a xAOD derivation
- Not PHYSLITE
- Combine later with PHYSLITE (result per-jet)

Request fairly complex data:
- Calorimeter clusters
- Muon segments
- Tracks

Currently driving the development of the backend

Development Issues

SerivceX 2.0 front-end allowed for a local light-weight version of the backend

- Could develop the backend code locally
- Run it in a container
- Full access to logs, errors.
- Edit the backend code and it would be picked up in the next run

3.0 loses that ability

- Need development work to fix that

Conclusion

- Good
  - Code is in production
  - Has been used to help other analyses
  - Have enough people to do this work…
- Challenges
  - Big shift in team personal!
  - xAOD expertise is limited to one person
  - Low-level work is limited by shift in tools