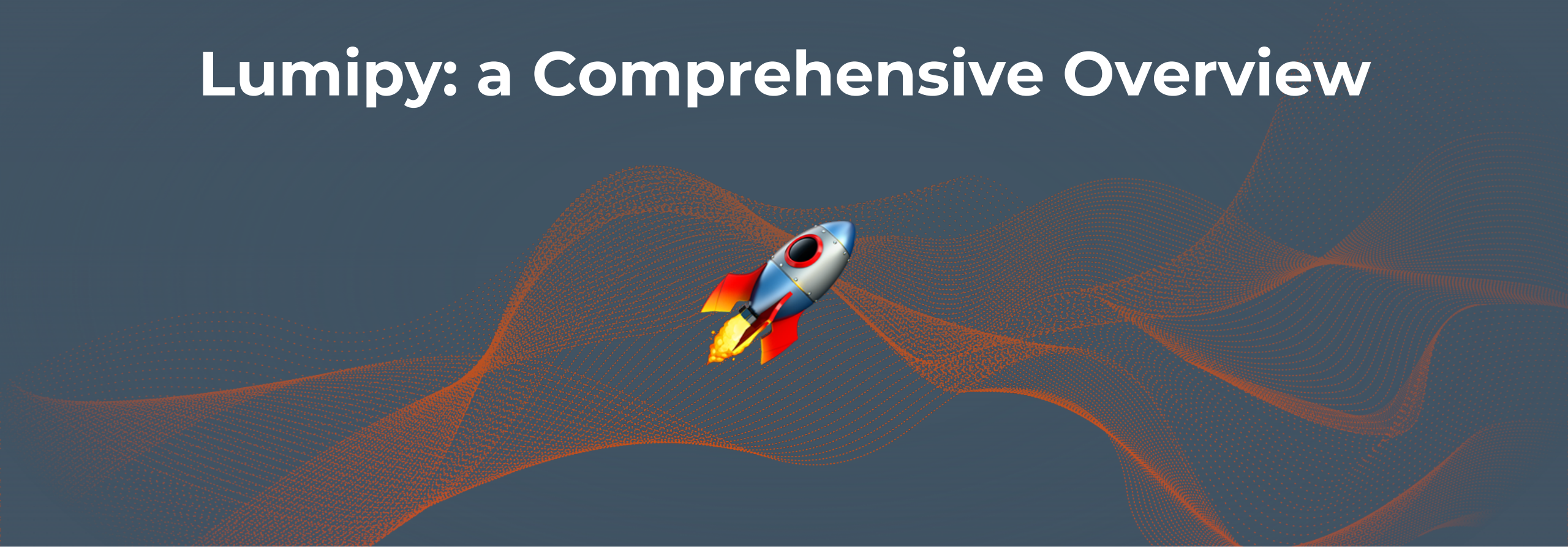
FINBOURNE
LIBERATE • SIMPLIFY • CONNECT

Lumipy: a Comprehensive Overview

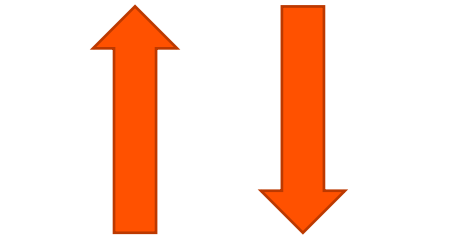


Introduction

- Lumipy is a library that makes it easy to use Luminesce as part of the python data science ecosystem

- Designed to...
  - Get your data into pandas with minimal fuss
  - Be used in Jupyter and support interactive data exploration
  - Be familiar and intuitive
  - Be tactile and discoverable

LUMINESCE
BY FINBOURNE



Getting Started

- Installation
  - Available from PyPI

```
pip install dve-lumipy-preview
```

- Authentication:
  - Uses the same infrastructure as our other python SDKs.
  - You have a choice of
    - Secrets File
    - Token and API URL
    - Proxy authentication
    - Environment Variables

```
{
    "api" : {
    "tokenUrl": "the okta token url",
    "username": "your okta username",
    "password": "your okta password",
    "clientId": "the client id",
    "clientSecret": "the client secret",
    "lumiApiUrl":
        "https://<ENV>.lusid.com/honeycomb/
    }
}
```

Your First Query – Running a SQL String

EX NB 1

- There are three parts
  - Import lumipy
  - Create client object by giving your authentication details
  - Use the client run method with your SQL string and get the result dataframe back

```
import lumipy as lm

client = lm.get_client(
    api_secrets_filename="/path/to/secrets.json"
)

df = client.run(
    "select * from lusid.instrument limit 10"
)
```

- But there is more to lumipy than sending SQL to the client

The Atlas

EX NB 2

- An atlas works like the hub for getting at your data
- It's created in the same way as the client
- It has two functions:
  - Exploring the the providers you have available to you and their metadata
  - The starting point for building queries in the fluent syntax

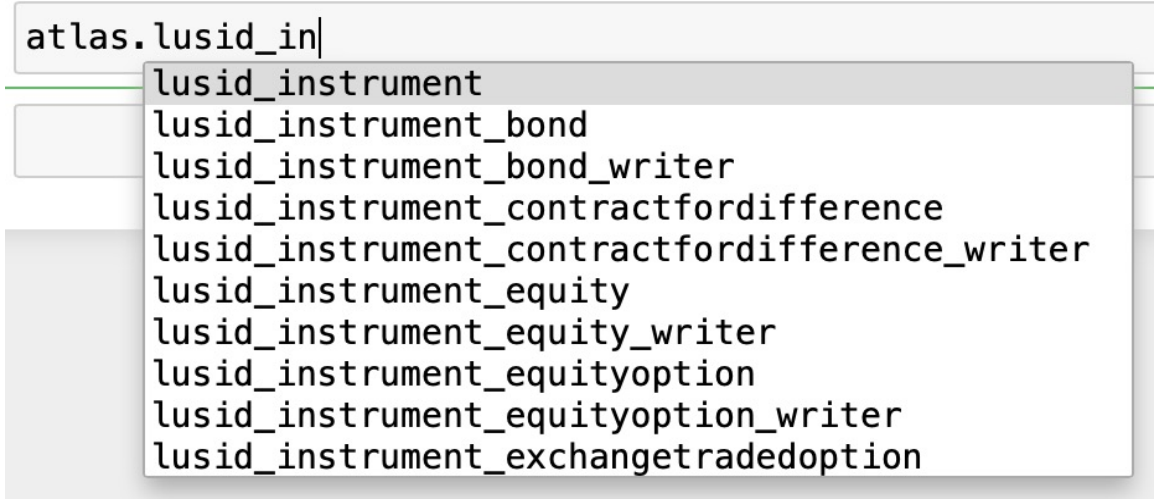
```
import lumipy as lm

atlas = lm.get_atlas(
    api_secrets_filename="/path/to/secrets.json"
)
```

```
Getting atlas🌍
  • Querying data provider metadata...
  • Querying direct provider metadata...
  • Building atlas...
Done!
Contents:
  • 343 data providers
  • 25 direct providers
```

The Atlas - Exploration

EX NB 2

- All providers are represented as attributes on the atlas
  - You can explore them via tab completion
  - You can print them to see metadata
  - You can also explore each provider's fields in the same way
- You can explore your providers programmatically
  - search_providers will return another atlas of matching providers
  - list_providers will enumerate the providers in an atlas




```
inst_atlas = atlas.search_providers('instrument')
```

```
inst_provs = inst_atlas.list_providers()
```

The Atlas – Scripting Queries

EX NB 2

- All queries start with an atlas attribute
- They are used to initialise a provider table object
- Queries are then built from provider objects via method-chaining
- These atlas attributes can be inspected using tab + shift in Jupyter

```
atlas.lusid_instrument()
```

```
Signature:
atlas.lusid_instrument(
    as_at: datetime.datetime,
    effective_at: datetime.datetime,
    use_lusid_filter: bool,
) -> lumipy.query.expression.table.base_source_table.SourceTable
Type:           LusidInstrumentFactory
String form:
└•Provider Definition: lusid_instrument
          ├•Table SQL Name: Lusid.Instrument    <...> e_at                Param
```

Query Scripting – Your First Query

EX NB 3

- The syntax is based on pyspark and pandas with minimal imports.
- Workflow
  - Start with your initialised provider object
  - Select columns from and chain other clauses
  - Start the query by calling .go() and receive a dataframe back
- By chaining methods you're building up the pieces of a SQL query
- You can inspect the underlying SQL with the print_sql method.

Provider parameters go here as keyword arguments

```
inst = atlas.lusid_instrument()
```

```
query = inst.select('*').limit(10)
```

```
df = query.go()
```

```
query.print_sql()
```

Query Scripting – More Basics

EX NB 3

- Parameters are given to the atlas attribute
- New columns are given in the select or aggregate methods as keyword args

- Methods corresponding to SQL clauses must be chained in the same order as a normal SQL query
- You don't have to use all the clauses each time.

```
import datetime as dt
inst = atlas.lusid_instrument(
    as_at=dt.datetime(2022, 9, 1)
)
```

```
query = inst.select(
    '^',
    inst.scope,
    NewColumn=3.1415
).where(

).group_by(

).aggregate(

).having(

).order_by(

).limit(

)
```

Query Scripting – Case Statements

EX NB 3

- Case statements are a series of conditions and values that build up a column
- They are built by starting at the table object and calling the when method
- After that you call the then method
- This is repeated until you optionally call otherwise. That's the end of the sequence
- If no otherwise is specified then the default value will be NULL

```
label = table.when(
    table.cost >= 1000000
).then(
    “millions”
).when(
    (table.cost >= 1000) & (table.cost < 1000000)
).then(
    “thousands”
).when(
    (table.cost >= 100) & (table.cost < 1000)
).then(
    “hundreds”
).otherwise(
    “less than a hundred”
)

query = table.select(‘*’, Label=label)
```

Query Scripting – Functions

EX NB 4

- We wish to avoid all the (many) imports and doc-searching you have to do in Pyspark. You only need the one in Lumipy.
- Functions are used via accessor attributes just like pandas Series' .str and .dt
- The full set of accessors is larger:
  - str – string functions
  - dt – datetime functions
  - stats – stats functions
  - metric – distance metrics
  - financial – finance-specific functions
  - linreg – linear regression functions
  - cume – cumulative functions

```
# pandas
pf[pf.PortfolioCode.str.contains('swap')]
```

```
# lumipy
pf.select(
    '*'
).where(
    pf.portfolio_code.str.contains('swap')
)
```

```
median = ar.duration.stats.quantile(0.5)
```

```
ar.duration.stats.quantile()
```

```
Signature: ar.duration.stats.quantile(q: float) -> lumipy.query.expression.column
_op.aggregation_op.Quantile
Docstring:
Apply a quantile function calculation to this expression.
This is an aggregation that will map to a single value.

Notes:
    The quantile function of a given random variable and q value finds the value
x where the probability of
    observing a value less than or equal to x is equal to q. See
        https://en.wikipedia.org/wiki/Quantile function
```

Query Scripting – Table Variables

EX NB 4

- Any product of select etc can be turned into a table variable with to_table_var
- You don't have to specify a name. Lumipy will automatically generate them.
- Once you have a table variable it can be treated like any other table: you start by chaining select.
- You can also make scalar variables in a similar way.

```
pf = atlas.lusid_portfolio()
```

```
tv = pf.select(
    '*'
).where(
    pf.portfolio_scope =='Finbourne-Examples'
).to_table_var()
```

```
query = tv.select(
    tv.portfolio_scope
).group_by(
    tv.portfolio_scope
).aggregate(
    NumPortfoliosInScope=tv.portfolio_code.count()
)
```

Query Scripting – Windows

EX NB 4

- Windows are specified in lumipy using a top-level function in the module
- These correspond to FILTER OVER in SQL. You can specify partitions, ordering and sliding/expanding windows.
- Parameters:
  - lower – lower limit of the window given as n rows before current. None = no limit.
  - upper – upper limit of the window given as n rows after current limit. None = no limit. Defaults to 0.
  - groups – partitions split the window by
  - orders – ordering to sort the data by before applying the window
- Filter is specified by chaining the filter method on the window object

```
import lumipy as lm

# default = all rows up to current
window = lm.window()
```

```
# 90 rows before current
lm.window(lower=90)
```

```
# sliding window 10 rows either side
lm.window(lower=10, upper=10)
```

```
# all rows, partition by column
lm.window(groups=table1.col, upper=None)
```

```
fltr_window = window.filter(table1.col > 2)
```

Query Scripting – Window Functions

EX NB 4

- Window functions are used by chaining methods on the window objects – just like the table columns
- The windows also have accessor objects that organise the available functions
- All of our statistical functions are window-able, so you can do some sophisticated analyses in sliding/expanding windows that are even partitioned by value

```
import lumipy as lm

w90days = lm.window(lower=90)
```

```
# max drawdown in a 90 day window
max_dd_90d = w90days.finance.max_drawdown(
    table.prices
)
```

Query Scripting – Drive and View Creation EX NB 4

- All queries can be written to drive by using to_drive with a given filepath
- This will write the results to the location in the format specified in the filepath
  - CSV if it ends in .csv
  - Excel if it ends in .xlsx
  - Sqlite if it ends in .sqlite
- You can also read files back out of drive using the drive direct providers
  - drive_csv
  - drive_excel
  - drive_rawtext
  - drive_sqlite
  - drive_xml
- Views can be created with create_view on queries and they can be deleted via delete_view on client objects

```
inst = atlas.lusid_instrument()
```

```
query = inst.select(
    '^'
).limit(
    100
).to_drive('/testing/path/inst.csv')
```

```
csv = atlas.drive_csv(
    file='/testing/path/inst.csv'
)

query = csv.select('*').where(csv.col1 > 2)
```

```
query.create_view('test.less.than.two')
```

```
client.delete_view('test.less.than.two')
```

Query Scripting – Joins

EX NB 5

- Joins are built using methods on the table objects where you specify the other table and an ‘on’ condition
- You may also supply aliases for either side of the join. When chaining many joins together you are forced to name them.
- The result will behave like another table which you then call the select method on
- Lumipy will automatically alias any columns with clashing names.
- If you want to join on something you’ve already selected, filtered, etc. you should convert to a table variable

```
join_table = table1.inner_join(
    table2,
    on=table1.id == table2.id
)
```

```
query = join_table.select('*').limit(10)
```

```
tv = table1.select('*').where(table1.col >
2).to_table_var()

join_table = tv.inner_join(
    table2,
    on=table1.id == table2.id
)
```

Query Scripting – Concat (Unions)

- There is a top-level concat function in lumipy that's analogous to pandas.concat
- This will union together a collection of subqueries
- Particularly useful when scripting:
  - You can create a parameterised set of subqueries with a function and then combine them with concat

```
import pandas as pd

df = pd.concat(list_of_dataframes)
```

```
import lumipy as lm

query = lm.concat(list_of_subqueries)
```

```
def make_subqueries(t_ranges):
    for t1, t2 in t_ranges:
        # do stuff
        yield subquery


query = lm.concat(make_subqueries())
```

Python Providers - Introduction

- There are many great python apps for data science that are an indispensable part of our workflow
- Connecting these to luminesce is the other half of what lumipy does
- We have (experimental) support for writing your own providers in python
- These just require you to inherit from a base class and implement two methods
- There are also pre-built python providers. For example a pandas one that turns dataframes into luminesce providers

Python Providers - Workflow

- The workflow for building and running python providers consists of a few steps
  - Import lumipy.provider
  - Initialise provider classes
  - Build a provider manager
  - Start it all up by calling run on the provider manager
- This will handle spinning things up behind the scenes. All you have to do is make the provider objects and configure the manager
- This will be made available in future as a docker image

```
import lumipy.provider as lp
```

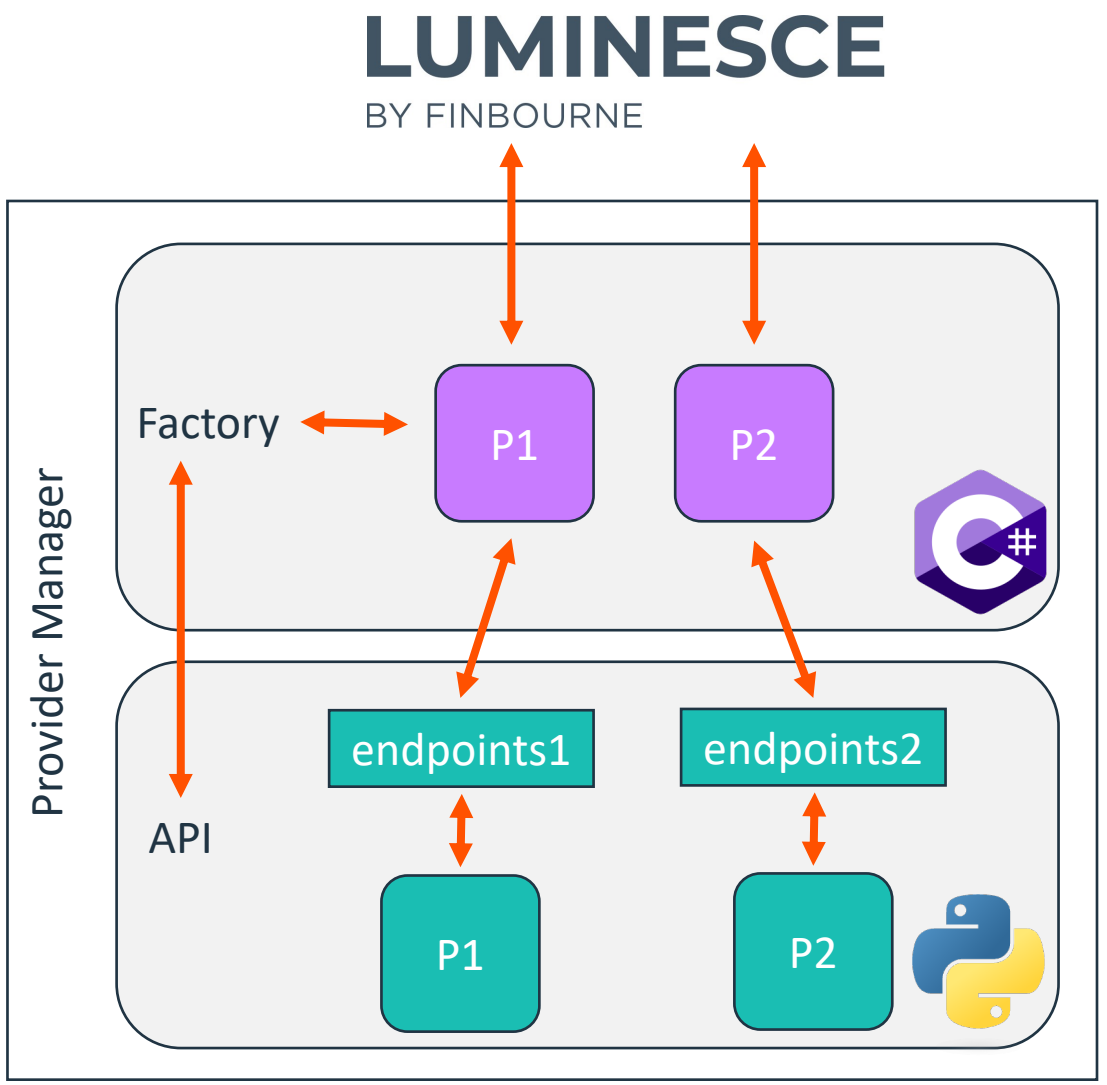
```
providers = [
    lp.PandasProvider(df, 'stock.returns’),
    lp.QuadraticProgram()
]
```

```
manager = lp.ProviderManager(
    *providers,
    user=‘<my user id>',
    domain=‘<my domain>’
)
```

```
manager.run()
```

Python Providers – How they Work

- The python provider infrastructure consists of two pieces: the Provider API and the Provider Factory
- The Provider API
  - is a local webserver written in python (FastAPI)
  - It wraps a collection of python provider objects and creates a group of API endpoints for each one
- The Provider Factory:
  - Is a dotnet application that runs at the same time as the API
  - It will start up a luminesce provider for each group of endpoints
  - Once running these providers call out to the underlying python API



Python Providers - Examples

- Three examples:
  - A straightforward pandas dataframe running as a provider
  - An index builder that wraps a quadratic optimisation for allocating weights given some returns data
  - Something completely useless…

Summary

- Lumipy provides a suite of tools for using luminesce and python together
- For getting data from luminesce into the python data science ecosystem you have the atlas and fluent query syntax
- For getting data and result back into luminesce we have the python provider infrastructure

Thank You For Listening!

- Any questions?