Glide

Release 0.3.1

Kurt Matarese

Apr 14, 2021

CONTENTS

`Glide` aims to be a simple, reusable, extensible approach to building ETL data pipelines. It provides a suite of nodes and pipelines out of the box that cover many common use cases, such as reading and writing data to/from SQL, URLs, local/remote files, and email.

`Glide` aims to have sane defaults and rely on standard, well-known Python libraries for data processing under the hood. It strives for familiar behavior out of the provided nodes while allowing for customization by passing arguments through to the underlying libraries in many cases. It's also very easy to write completely custom nodes and pipelines.

`Glide` also tries to give you a lot for free, including simple parallel processing support, a variety of ways to manage node/pipeline context, and automatic CLI generation.

`Glide` encourages extensions that provide nodes and pipelines for interacting with various data sources and data processing modules. Check out the `glide.extensions` module to see some currently supported extensions, such as Dask for scalable analytics workflows.

CHAPTER

ONE

GLIDE: EASY ETL

1.1 Introduction

Glide is an easy-to-use data pipelining tool inspired by Consecution and Apache Storm Topologies.

Like those libraries, **Glide is:**

- A simple, reusable approach to building robust ETL pipelines
- A system for wiring together processing nodes to form a directed acyclic graph (DAG)

Glide also has:

- An expanding suite of built-in nodes and pipelines that extract, transform, and load data from/to any combination of:
  - SQL databases (SQLite, DBAPI, and SQLAlchemy support)
  - Local or remote files (CSVs, Excel, and raw file support)
  - URLs (JSON endpoints, file downloads, APIs, etc.)
  - HTML Tables
  - Emails
- Extensions for Pandas, Dask, Celery, Redis Queue and more
- A variety of node and DAG parallel/concurrent/distributed processing strategies
- A simple decorator to generate a command line interface from a pipeline in ~one line of code
- Flexible pipeline templating

Glide is not a task orchestration and/or dependency management tool like Airflow. Use Glide to define your easily developed/contained/reusable/testable data processing pipelines and then rely on a tool like Airflow to do what it's good at, namely scheduling and complex task dependency management.

1.2 Table of Contents

1.3 Installation

Warning: This project is in an alpha state and is maintained on an as-needed basis. Please test carefully for production usage and report any issues.

```
$ pip install glide
```

1.4 Primer

You are encouraged to take a deeper look at the docs, but the short of it is the following:

1. A `Node` is a part of a pipeline which has a `run` method that typically accepts data from upstream nodes, and pushes data to downstream nodes. For example:

```
class MyNode(Node):
    def run(self, data):
        # Some node-specific code here
        self.push(data)
```

2. A `Glider` is a pipeline of `Node` objects wired together in a DAG. It accepts input data in its `consume` method. For example:

```
glider = Glider(
    MyExtractNode("extract")
    | MyTransformNode("transform")
    | MyLoadNode("load")
)
glider.consume(data)
```

If a node's `run` method has additional parameters, they are populated from the node's `context`. More info on creating nodes and populating runtime context can be found here.

1.5 Basic Examples

The following examples serve to quickly illustrate some core features and built-in nodes. There is much more `Glide` can do that is not shown here. Everything below assumes you have used the following shortcut to import all necessary node and pipeline classes:

```
from glide import *
```

1.5.1 Example: CSV Extract, Transform, and Load

Apply a transformation to data from a CSV, use a function to lowercase all strings, and load into an output CSV:

```
def lower_rows(data):
    for row in data:
        for k, v in row.items():
            row[k] = v.lower() if type(v) == str else v
    return data

glider = Glider(
    CSVExtract("extract")
    | Func("transform", func=lower_rows)
    | CSVLoad("load")
)
glider.consume(
    ["/path/to/infile.csv"],
    extract=dict(chunksize=100),
    load=dict(outfile="/path/to/outfile.csv"),
)
```

1.5.2 Example: SQL Extract and Load

Read from one table, write to another:

```
conn = get_my_sqlalchemy_conn()
sql = "select * from in_table limit 10"

glider = Glider(
    SQLExtract("extract")
    | SQLLoad("load"),
    global_state=dict(conn=conn) # conn is automagically passed to any nodes that␣
↪accept a "conn" argument
)
glider.consume(
    [sql],
    load=dict(table="out_table")
)
```

1.5.3 Example: SQL Transactions

Start a transaction before writing to a database, rollback on failure:

```
glider = Glider(
    SQLExtract("extract")
    | SQLTransaction("tx")
    | SQLLoad("load", rollback=True),
    global_state=dict(conn=conn)
)
glider.consume(...)
```

1.5.4 Example: URL Extraction

Extract data from each URL in the list of requests and load to a URL endpoint:

```
glider = Glider(URLExtract("extract") | URLLoad("load"))
reqs = [
    "https://jsonplaceholder.typicode.com/todos/1",
    "https://jsonplaceholder.typicode.com/todos/2",
]
glider.consume(
    reqs,
    extract=dict(data_type="json"),
    load=dict(
        url="https://jsonplaceholder.typicode.com/todos",
        data_param="json",
        headers={"Content-type": "application/json; charset=UTF-8"},
    ),
)
```

1.6 Flow Control Examples

1.6.1 Example: Filters

Filter the propagation of data based on the result of a function:

```
def data_check(node, data):
    # do some check on data, return True/False to control filtering
    return True

glider = Glider(
    MyExtract("extract")
    | Filter("filter", func=data_check)
    | MyLoad("load")
)
```

1.6.2 Example: IterPush

Push each row of an input iterable individually:

```
glider = Glider(
    CSVExtract("extract", nrows=20)
    | IterPush("iter")
    | Print("load")
)
```

1.6.3 Example: SplitPush

Split an iterable before pushing:

```
glider = Glider(SplitPush("push", split_count=2) | Print("print"))
glider.consume([range(4)])
```

1.6.4 Example: SplitByNode

Split an iterable evenly among downstream nodes:

```
glider = Glider(SplitByNode("push") | [Print("print1"), Print("print2")])
glider.consume([range(4)])
```

1.6.5 Example: Reduce

Collect all upstream node data before pushing:

```
glider = Glider(
    CSVExtract("extract")
    | Reduce("reduce")
    | Print("load")
)
glider.consume(["/path/to/infile1.csv", "/path/to/infile2.csv"])
```

This will read both input CSVs and push them in a single iterable to the downstream nodes. You can also use the `flatten` option of `Reduce` to flatten the depth of the iterable before pushing (effectively a concat operation).

1.6.6 Example: Join

Join data on one or more columns before pushing:

```
glider = Glider(
    Reduce("reduce")
    | Join("join")
    | Print("load")
)
d1 = <list of dicts or DataFrame>
d2 = <list of dicts or DataFrame>
glider.consume([d1, d2], join=dict(on="common_key", how="inner"))
```

1.6.7 Example: Routers

Route data to a particular downstream node using a router function:

```
def parity_router(row):
    if int(row["mycolumn"]) % 2 == 0:
        return "even"
    return "odd"

glider = Glider(
```

(continues on next page)

(continued from previous page)

```
    CSVExtract("extract", nrows=20)
    | IterPush("iter")
    | [parity_zip_router, Print("even"), Print("odd")]
)
glider.consume(...)
```

This will push rows with even `mycolumn` values to the "even" `Print` node, and rows with odd `mycolumn` values to the "odd" `Print` node.

1.6.8 Example: Window Processing

Push a sliding window of the data:

```
glider = Glider(
    CSVExtract("extract", nrows=5)
    | WindowPush("window", size=3)
    | MyWindowCalcNode("calc")
)
```

1.6.9 Example: Date Windows

Generate a set of datetime windows and push them downstream:

```
import datetime

today = datetime.date.today()
glider = Glider(DateTimeWindowPush("windows") | PrettyPrint("print"))
glider.consume(
    windows=dict(
        start_date=today - datetime.timedelta(days=3),
        end_date=today,
        num_windows=2
    )
)
```

Or use `DateWindowPush` for date objects. Note that the data arg to `consume` can be ignored because the top node (`DateTimeWindowPush`) is a subclass of `NoInputNode` which takes no input data and generates data to push on its own.

1.6.10 Example: Return Values

By default `consume` does not return any values and assumes you will be outputting your results to one or more endpoints in your terminating nodes (files, databases, etc.). The `Return` node will collect the data from its parent node(s) and set it as a return value for `consume`.

```
glider = Glider(
    CSVExtract("extract")
    | MyTransformer("transform")
    | Return("return")
)
data = glider.consume(...)
```

1.7 Parallelization & Concurrency

1.7.1 Example: Parallel Transformation

Call a function in parallel processes on equal splits of data from a CSV:

```
glider = Glider(
    CSVExtract("extract")
    | ProcessPoolSubmit("transform", push_type=PushTypes.Result)
    | CSVLoad("load")
)
glider.consume(
    ["infile.csv"],
    transform=dict(func=lower_rows),
    load=dict(outfile="outfile.csv"),
)
```

We passed `push_type=PushTypes.Result` to force `ProcessPoolSubmit` to fetch and combine the asynchronous results before pushing to the downstream node. The default is to just pass the asynchronous task/futures objects forward, so the following would be equivalent:

```
glider = Glider(
    CSVExtract("extract")
    | ProcessPoolSubmit("transform")
    | FuturesReduce("reduce")
    | Flatten("flatten")
    | CSVLoad("load")
)
```

The `FuturesReduce` node waits for the results from each futures object, and then `Flatten` will combine each subresult back together into a single result to be loaded in the final `CSVLoad` node.

1.7.2 Example: Parallel Pipelines via ParaGlider

Completely parallelize a pipeline using a `ParaGlider` (who said ETL isn't fun?!?). Split processing of the inputs (two files in this case) over the pool, with each process running the entire pipeline on part of the consumed data:

```
glider = ProcessPoolParaGlider(
    CSVExtract('extract')
    | Print('load')
)
glider.consume(
    ["/path/to/infile1.csv", "/path/to/infile2.csv"],
    extract=dict(nrows=50)
)
```

1.7.3 Example: Parallel Branching

Branch into parallel execution in the middle of the DAG utilizing a parallel push node:

```
glider = Glider(
    CSVExtract("extract", nrows=60)
    | ProcessPoolPush("push", split=True)
    | [Print("load1"), Print("load2"), Print("load3")]
)
glider.consume(["/path/to/infile.csv"])
```

The above example will extract 60 rows from a CSV and then push equal slices to the logging nodes in parallel processes. Using `split=False` (default) would have passed the entire 60 rows to each logging node in parallel processes.

Once you branch off into processes with a parallel push node there is no way to reduce/join the pipeline back into the original process and resume single-process operation. The entire remainder of the pipeline is executed in each subprocess. However, that is possible with threads as shown in the next example.

1.7.4 Example: Thread Reducers

```
glider = Glider(
    CSVExtract("extract", nrows=60)
    | ThreadPoolPush("push", split=True)
    | [Print("load1"), Print("load2"), Print("load3")]
    | ThreadReduce("reduce")
    | Print("loadall")
)
glider.consume(["/path/to/infile.csv"])
```

The above code will split the data and push to the first 3 logging nodes in multiple threads. The `ThreadReduce` node won't push until all of the previous nodes have finished, and then the final logging node will print all of the results.

1.7.5 Example: Asyncio

Limited, experimental support is also available for concurrency via `asyncio` in Python >= 3.7:

```
import asyncio

async def async_sleep(data):
    # Dummy example. Await some real async work in here.
    await asyncio.sleep(0.5)
    return data

glider = Glider(
    CSVExtract("extract", nrows=5)
    | AsyncIOSubmit("transform", push_type=PushTypes.Result)
    | Print("load")
)
glider.consume(
    ["/path/to/infile.csv"],
    transform=dict(func=async_sleep)
)
```

The above example will split the input data into items to be processed on an `asyncio` event loop and synchronously wait for the results before pushing. `AsyncIOSubmit` supports specifying a `split_count` as well as a `timeout` when waiting for results. Alternatively, one can push `asyncio` futures and later reduce their results as follows:

```
glider = Glider(
      CSVExtract("extract", nrows=5)
      | AsyncIOSubmit("transform", push_type=PushTypes.Async)
      | AsyncIOFuturesReduce("reduce", flatten=True)
      | Print("load")
)
```

Note that the `asyncio` nodes will create and start an event loop for you if necessary. It's also perfectly fine to manage the event loop on your own, in which case `glide` will run tasks on the current thread's event loop.

1.8 Utility Examples

1.8.1 Example: Templated Nodes and Pipelines

Drop replacement nodes into an existing pipeline. Any node can be replaced by name:

```
glider = Glider(
    PlaceholderNode("extract")
    | CSVLoad("load")
)
glider["extract"] = CSVExtract("extract")
glider.consume(...)
```

Or reuse an existing structure of nodes with a `NodeTemplate`:

```
nodes = NodeTemplate(
    CSVExtract("extract")
    | CSVLoad("load")
)
glider = Glider(nodes()) # Copy of nodes created with each call
```

Or reuse an existing pipeline structure with `GliderTemplate`:

```
template = GliderTemplate(
    CSVExtract("extract")
    | CSVLoad("load")
)
glider = template() # Copy of pipeline created with each call
```

1.8.2 Example: Data Integrity Checks

You can use the `AssertFunc` node to assert that some condition of the data is met:

```
glider = Glider(
    CSVExtract("extract", chunksize=10, nrows=20)
    | AssertFunc("length_check", func=lambda node, data: len(data) == 10)
    | CSVLoad("load")
)
```

The `func` callable must accept two parameters, a reference to the node object and the data passed into that node. Any truthy value returned will pass the assertion test.

Similarly, you can do a sql-based check with `AssertSQL`, in this case simply verifying the number of rows inserted:

```
glider = Glider(
    SQLExtract("extract")
    | SQLLoad("load")
    | AssertSQL("sql_check")
)

sql = "select * from in_table limit 10"
assert_sql = "select (select count(*) as x from out_table) == 10 as assert"

glider.consume(
    [sql],
    extract=dict(conn=in_conn),
    load=dict(conn=out_conn, table="out_table"),
    sql_check=dict(conn=out_conn, sql=assert_sql)
)
```

This looks for a truthy value in the `assert` column of the result to pass the assertion. You can also use the `data_check` option of `AssertSQL` to instead have it do a comparison to the result of some function of the data:

```
glider = ...

sql = "select * from in_table limit 10"
assert_sql = "select count(*) as assert from out_table"

glider.consume(
    [sql],
    extract=dict(conn=in_conn),
    load=dict(conn=out_conn, table="out_table", push_data=True),
    sql_check=dict(
        conn=out_conn,
        sql=assert_sql,
        data_check=lambda node, data: len(data)
    )
)
```

Note that we also added `push_data=True` to the `SQLLoad` node to have it push the data instead of a table name.

1.8.3 Example: Debugging

To enable debug logging for Glide change the log level of the "glide" logger:

```
import logging
logging.getLogger("glide").setLevel(logging.DEBUG)
```

Glide will then print debug information about data passed through your pipeline.

You can also pass `_log=True` to the init method of any node to enable logging of processed data:

```
glider = Glider(
    SQLExtract("extract", _log=True)
    ...
)
```

Additionaly, you can pass `_debug=True` to the init method of any node to cause the node to drop into PDB right before calling `run`, assuming you aren't executing the pipeline in a subprocess:

```
glider = Glider(
    SQLExtract("extract", _debug=True)
    ...
)
```

Finally, there are a variety of print nodes you can place in your pipeline for general logging or debugging, such as `Print`, `PrettyPrint`, `LenPrint`, `ReprPrint`, and `FormatPrint`. See the node documentation for more info.

1.8.4 Example: Profiling Pipelines

Insert a `Profile` node somewhere in your pipeline to get profiler information for all downstream nodes:

```
glider = Glider(
    Profile("profile")
    ...
)
```

1.8.5 Example: Complex Pipelines

If the hierarchy of nodes you want to form is not achievable with the `|` operator, you can use the `add_downstream Node` method to form more complex graphs. More info can be found here.

1.8.6 Example: Plotting Pipeline DAGs

If you have the Graphviz package installed, you can generate a plot of your pipelines by simply doing the following:

```
glider = Glider(...)
glider.plot("/path/to/filename.png")
```

1.9 CLI Generation

With Glide you can create parameterized command line scripts from any pipeline with a simple decorator:

```
glider = Glider(
    SQLLoad("extract")
    | SQLExtract("load")
)

@glider.cli()
def main(glide_data, node_contexts):
    glider.consume(glide_data, **node_contexts)

if __name__ == "__main__":
    main()
```

The script arguments, their types, and whether they are required or not is all inferred by inspecting the `run` arguments on the nodes of the pipeline and prefixing the node name. Please see the full documentation here for more details.

1.10 Extensions

To install all extensions and dev dependencies:

```
$ pip install glide[complete]
```

You can also just install Glide plus a specific extension:

```
$ pip install glide[dask]
$ pip install glide[celery]
$ pip install glide[rq]
$ pip install glide[swifter]
```

To access installed extensions import from the `glide.extensions` submodules as necessary. Review the documentation and tests for current extensions for help getting started.

1.10.1 Pandas

The Pandas extension is actually supported by default with all `glide` installs. Below is a simple example that extracts from a CSV, lowercases all strings, and then loads to another CSV using Pandas under the hood:

```
def lower(s):
    return s.lower() if type(s) == str else s

glider = Glider(
    DataFrameCSVExtract("extract")
    | DataFrameApplyMap("transform", func=lower)
    | DataFrameCSVLoad("load", index=False, mode="a")
)
glider.consume(...)
```

There are a variety of other helpful nodes built in, including `ToDataFrame`, `FromDataFrame`, nodes to read/write other datasources, and nodes to deal with `rolling` calculations. There is also a generic `DataFrameMethod` node that passes through to any DataFrame method.

See the extension docs here for node/pipeline reference information. See the tests here for some additional examples.

1.10.2 Dask - Experimental

See the extension docs here for node/pipeline reference information. See the tests here for some additional examples.

1.10.3 Celery - Experimental

See the extension docs here for node/pipeline reference information. See the tests here for some additional examples.

1.10.4 Redis Queue - Experimental

See the extension docs here for node/pipeline reference information. See the tests here for some additional examples.

1.10.5 Swifter - Experimental

See the extension docs here for node/pipeline reference information. See the tests here for some additional examples.

1.11 Documentation

More thorough documentation can be found here. You can supplement your knowledge by perusing the tests directory or the module reference.

1.12 How to Contribute

See the CONTRIBUTING guide.

CHAPTER

TWO

CREATING NODES

To create a custom node you simply inherit from the Glide `Node` class and define a `run` method that takes at least one positional argument for the data being pushed to it. The `run` method should call `self.push(data)` with the data it wants to push downstream.

Here is an example of a simple transformer node:

```
class ExampleTransformer(Node):
    def run(self, data):
        # Do something to the data here
        self.push(data)
```

CHAPTER

THREE

NODE CONTEXT

Each node has a `context`. This comes into play when `run` is called on the node, as the required and optional parts of the context are inferred from the positional and keyword args of `run`. Take for example:

```
class MyNode(Node):
    def run(self, data, conn, chunksize=None, **kwargs):
        # Some node-specific code here
        self.push(data)
```

By default all nodes expect their first positional arg to be the data going through the pipeline. This node also requires a `conn` argument, and has an optional `chunksize` argument. These values can be filled in from the following inputs in priority order, with earlier methods overriding those further down the list:

1. Context args passed to `consume` for the particular node:

   ```
   conn = get_my_db_conn()
   glider.consume(
       data,
       my_node=dict(conn=conn, chunksize=100)
   )
   ```

2. Default context set on the node at init time:

   ```
   conn = get_my_db_conn()
   glider = Glider(
       MyNode("my_node", conn=conn, chunksize=100)
   )
   ```

3. Global pipeline state passed via `global_state`. This only works for populating positional args currently:

   ```
   conn = get_my_db_conn()
   glider = Glider(
       MyNode("my_node"),
       global_state=dict(conn=conn)
   )
   ```

Additionally, you can update the context of nodes at runtime by using the `update_context` or `update_downstream_context` node methods.

CHAPTER

FOUR

RUNTIME CONTEXT

Sometimes it is useful or necessary to fill in node context values at runtime. A prime example is when using SQL-based nodes in a parallel processing context. Since the database connection objects can not be pickled and passed to the spawned processes you need to establish the connection within the subprocess. Glide has a special `RuntimeContext` class for this purpose. Any callable wrapped as a RuntimeContext will not be called until `consume` is called. In the example below, `get_pymysql_conn` will be executed in a subprocess to fill in the "conn" context variable for the "extract" node:

```
glider = ProcessPoolParaGlider(
    SQLExtract("extract")
    | PrettyPrint("load")
)
glider.consume(
    [sql],
    extract=dict(
        conn=RuntimeContext(get_pymysql_conn),
        cursor_type=pymysql.cursors.DictCursor,
    )
)
```

In this case it is also necessary to specify the cursor_type so `SQLExtract` can create a dict-based cursor for query execution within the subprocess as required by `SQLExtract`. Any args/kwargs passed to RuntimeContext will be passed to the function when called.

CHAPTER

FIVE

CONFIG CONTEXT

`ConfigContext` is an alternative type of `RuntimeContext` that can read a config file to fill in node context. It supports reading from JSON, INI, or YAML config files and optionally extracting specific data from the file. The following shows an example of reading a key ("nrows") from a JSON structure:

```
glider = Glider(
    CSVExtract("extract", nrows=ConfigContext("myconfig.json", key="nrows"))
    | Print("load")
)
glider.consume(...)
```

As another example, the following reads from an INI file and also passes a callable for the `key` parameter to extract a value from the config:

```
glider = Glider(
    CSVExtract("extract", nrows=ConfigContext(
        "myconfig.ini", key=lambda x: int(x["Section"]["nrows"])
    ))
    | Print("load")
)
glider.consume(...)
```

If no value is specified for `key`, the entire config file is returned. `ConfigContext` may be particularly useful when you want to load sensitive information such as API login details that you would not want to store in your code.

CHAPTER

SIX

CLEANING UP

Sometimes it is also necessary to call clean up functionality after processing is complete. Sticking with the example above that utilizes SQL-based nodes in a parallel processing context, you'll want to explicitly close your database connections in each subprocess. The `consume` method accepts a `cleanup` argument that is a dictionary mapping argument names to cleaner functions. The following example tells the `Glider` to call the function `closer` with the value from `extract_conn` once `consume` is finished. Note that `closer` is a convenience function provided by Glide that just calls `close` on the given object.:

```
glider = ProcessPoolParaGlider(
    SQLExtract("extract")
    | PrettyPrint("load")
)
glider.consume(
    [sql],
    cleanup=dict(extract_conn=closer),
    extract=dict(
        conn=RuntimeContext(get_pymysql_conn),
        cursor_type=pymysql.cursors.DictCursor,
    )
)
```

The keys of the `cleanup` dict can either be explicit (node name prefixed) or more generic arg names that will map that function to every node that has that arg in its `run` method signature (so just "conn=" would have worked too). It's often better to be explicit as shown here.

> **Note:** In single-process cases the use of `cleanup` is usually not necessary, as you often have access to the objects you need to clean up in the main process and can just do normal clean up there with context managers or explicit calls to `close` methods.

CHAPTER

SEVEN

COMMON NODES

`Glide` comes with a suite of nodes to handle common data processing tasks. The easiest way to view the options and understand their behavior is to peruse the module documentation and/or review the source code for each node.

- For extractor nodes, such as SQL/CSV/Excel/File/URL extractors, see *glide.extract*.
- For transformer nodes, see *glide.transform*.
- For filter nodes, see *glide.filter*.
- For loader nodes, such as SQL/CSV/Excel/File/URL loaders, see *glide.load*.
- For some additional flow control nodes see *glide.core*.

Keep in mind it's very easy to write your own nodes. If you don't see something you want, or you want slightly different behavior, create your own node. If you think it's something that could benefit other users please contribute!

CHAPTER

EIGHT

CREATING PIPELINES

A `Glider` is a pipeline of `Node` objects wired together in a DAG. It accepts input data in its `consume` method. For example:

```
glider = Glider(
    MyExtractNode("extract")
    | MyTransformNode("transform")
    | MyLoadNode("load")
)
glider.consume(data)
```

The `consume` method accepts node_name -> node_context keyword arguments that can update the context of the pipeline's nodes for the duration of the `consume` call. For example, if `MyLoadNode` in the example above had an argument called `foo` in its `run` method, you could set the value of `foo` for a particular pipeline run as follows:

```
glider.consume(data, load=dict(foo="bar"))
```

Pipelines can be templated as well for easy reuse. Any node can be replaced by name:

```
glider = Glider(
    PlaceholderNode("extract")
    | CSVLoad("load")
)
glider["extract"] = CSVExtract("extract")
glider.consume(...)
```

Or reuse an existing pipeline structure with `GliderTemplate`:

```
template = GliderTemplate(
    CSVExtract("extract")
    | CSVLoad("load")
)
glider = template() # Copy of pipeline created with each call
glider.consume(...)
```

8.1 Complex Pipelines

Glide's `Node` class has an `add_downstream` method that it inherits from Consecution's Node class. You can use this to form more complex topologies, such as in the following example:

```
def parity_router(num):
    if num % 2 == 0:
        return "even"
    return "odd"

def threshold_router(num):
    prepend = "odd"
    if num % 2 == 0:
        prepend = "even"
    if num >= 10:
        return "%s_large" % prepend
    return "%s_small" % prepend

glider = Glider(
    CSVExtract("extract", nrows=40)
    | IterPush("iter")
    | [
        parity_router,
        (
            Print("even")
            | [threshold_router, Print("even_large"), Print("even_small")]
        ),
        (
            Print("odd")
            | [threshold_router, Print("odd_large"), Print("odd_small")]
        ),
    ]
)

large = Print("large")
small = Print("small")
reducer = Reduce("reduce")
combined = LenPrint("combined")

large.add_downstream(reducer)
small.add_downstream(reducer)
reducer.add_downstream(combined)

glider["even_large"].add_downstream(large)
glider["odd_large"].add_downstream(large)
glider["even_small"].add_downstream(small)
glider["odd_small"].add_downstream(small)

glider.consume(range(20))
glider.plot("pipeline.png") # View hierarchy if you have GraphViz installed
```

This also takes advantage of Consecution's router functionality to use `parity_router` and `threshold_router` to steer data through the pipeline.

CHAPTER

NINE

CLI GENERATION

With Glide you can create parameterized command line scripts from any pipeline with a simple decorator:

```
glider = Glider(
    SQLLoad("extract")
    | SQLExtract("load")
)

@glider.cli()
def main(glide_data, node_contexts):
    glider.consume(glide_data, **node_contexts)

if __name__ == "__main__":
    main()
```

The script arguments, their types, and whether they are required or not is all inferred by inspecting the `run` arguments on the nodes of the pipeline and prefixing the node name. For example, `SQLLoad` requires a `conn` and a `table` argument, as well as having a few optional arguments. Since the node is named "load", the CLI will automatically generate required args called `--load_conn` and `--load_table`. Additionally, the default help strings are extracted from the run() method documentation if you use numpy docstring format.

By default, the first positional argument(s) expected on the CLI are used to populate the `glide_data` argument. If the top node of your pipeline is a subclass of `NoInputNode` then the CLI will automatically skip the `glide_data` CLI arg and not try to pass any data as the first positional argument to the wrapped function.

Let's ignore the fact that you can't pass a real database connection object on the command line for a second and see how you would run this script:

```
$ python my_script.py "select * from input_table limit 10" \
--extract_conn foo \
--load_conn bar \
--load_table output_table
```

To pass multiple inputs to `glide_data` you would simply use space-separated positional arguments:

```
$ python my_script.py "sql query 1" "sql query 2" \
--extract_conn foo \
--load_conn bar \
--load_table output_table
```

One way to populate the `conn` arguments of pipeline nodes is to define it in the `global_state` or in the node initialization calls. In either case it is no longer considered a *required* command line argument. So the following would work:

```
glider = Glider(
    SQLExtract("extract")
    | SQLLoad("load"),
    global_state=dict(conn=get_my_db_conn())
)
```

```
$ python my_script.py "select * from input_table limit 10" \
--load_table output_table
```

9.1 Blacklisting Args

In the previous example it is no longer necessary to even have the node-specific connection arguments show up on the command line (such as in –help output). You can blacklist the arg from ever getting put into the CLI as follows:

```
@glider.cli(blacklist=["conn"])
def main(glide_data, node_contexts):
    glider.consume(glide_data, **node_contexts)
```

Or, if you just wanted to blacklist an argument that appears in multiple nodes from a single node (such as the `conn` argument required in both the extract and load nodes in this example), you could be more explicit and prefix the node name:

```
@glider.cli(blacklist=["load_conn"])
def main(glide_data, node_contexts):
    glider.consume(glide_data, **node_contexts)
```

That would remove `load_conn` from the CLI but not `extract_conn`.

9.2 Custom Arguments

You can also override or add any argument you want using the `Arg` class which takes the standard `argparse` arguments:

```
from glide.core import Glider, Arg

glider = ...

@glider.cli(Arg("--load_table", required=False, default="output_table"))
def main(glide_data, node_contexts):
    glider.consume(glide_data, **node_contexts)
```

And now, assuming you had used the `Glider` with `conn` passed in the `global_state`, you could simply do:

```
$ python my_script.py "select * from input_table limit 10"
```

You can override the `glide_data` positional argument in this way too if you want to change the type/requirements:

```
@glider.cli(Arg("glide_data", type=str, default="some default sql query"))
def main(glide_data, node_contexts):
    glider.consume(glide_data, **node_contexts)
```

You can also override some of the naming of specific node arguments to potentially simplify your CLI. You can use the argparse `dest` param to have an arg point at a specific context element. Here we name the custom arg `table` and have it fill in the value of `load_table` which ends up being a `run` argument of the "load" node:

```
@glider.cli(Arg("--table", dest="load_table", default="output_table"))
def main(glide_data, node_contexts):
    glider.consume(glide_data, **node_contexts)
```

If your custom args match a node's `run` args exactly, they can be used to fill in that value in the node context, potentially across multiple nodes if many have the same arg name. We saw similar behavior with the `conn` argument on the `global_state` above, but here is a more specific example sticking with the `table` custom arg:

```
@glider.cli(Arg("--table", default="output_table"))
def main(glide_data, node_contexts, table=None):
    glider.consume(glide_data, **node_contexts)
```

Notice that `load_table` is not targeted specifically, but its context will be filled in by the value of `table` on the CLI because the name of the CLI arg exactly matches the name of an arg in that node's `run` method. Also notice that `table` is now passed as a keyword argument to `main`. Any custom or injected args that do not exactly match a node name qualified CLI arg (such as "load_table") will be included as keyword arguments to `main`.

Note: Due to a known issue in argparse, even if you define an arg as required it will still show up in the optional arguments section of the help output if it has a dash or double-dash at the start of the arg name.

9.3 Argument Injection and Clean Up

The script decorator also has the ability to inject values into arguments based on the result of a function, and call clean up functions for the various injected arguments. The following example shows two useful cases:

```
def get_data():
    # do something to populate data iterable
    return data

@glider.cli(
    Arg("--load_table", required=False, default="output_table")
    inject=dict(glide_data=get_data, conn=get_my_db_conn),
    cleanup=dict(conn=lambda x: x.close()),
)
def main(glide_data, node_contexts, **kwargs):
    glider.consume(glide_data, **node_contexts)
```

Here we use the `inject` decorator argument and pass a dictionary that maps injected argument names to functions that return the values. We inject a `glide_data` arg and a `conn` arg and neither are necessary for the command line. This automatically blacklists those args from the command line as well. Since we added the `load_table` arg and gave it a default as well, we can now simply run:

```
$ python my_script.py
```

Note: Injected args are also passed to the wrapped function as keyword args if they do not exactly match a node name qualified CLI arg.

Note: If an injected argument name is mapped to a non-function via `inject` the value will be used as is. The main difference is those values are interpreted as soon as the module is loaded (when the decorator is init'd). If that is not desirable, pass a function as shown above which will only be executed once the decorated function is actually called. Injected RuntimeContexts and other objects that are not a `types.FunctionType` or `functools.partial` are passed through as-is.

The `cleanup` decorator argument takes a dictionary that maps argument names to callables that accept the argument value to perform some clean up. In this case, it closes the database connection after the wrapped method is complete.

9.4 Boolean Args

Node `run` args whose default is a boolean value will be converted to boolean flags on the CLI. If the default is `True`, the flag will invert the logic of the flag and prepend `no_` to the beginning of the arg name for clarity.

For example, the `SQLLoad` node has a `run` keyword arg with a default of `commit=True`. Assuming this node was named `load`, this will produce a CLI flag `--load_no_commit` which, when passed in a terminal, will set `commit=False` in the node. If the default had been `False` the CLI arg name would have simply been `--load_commit` and it would set the value to `True` when passed in a terminal.

This leads to more clear CLI behavior as opposed to having a flag with a truth-like name getting a false-like result when passed in a terminal. Of course another option would have been to define the node keyword arg as `no_commit=False` instead of `commit=True`. This would also lead to understandable CLI behavior but, in my opinion, would lead to more confusing variable naming in your code.

9.5 Parent CLIs

If you want to inherit or share arguments you can accomplish that using the `Parent` and `Arg` decorators together. These are using climax. under the hood, which is utilizing `argparse`. For example, the following script inherits a `--dry_run` boolean CLI flag:

```
from glide.core import Parent, Arg

@Parent()
@Arg("--dry_run", action="store_true")
def parent_cli():
    pass

@glider.cli(parents=[parent_cli])
def main(glide_data, dry_run=False, node_contexts):
    if dry_run:
        something_else()
    else:
        glider.consume(glide_data, **node_contexts)
```

CHAPTER

TEN

PARALLEL PROCESSING

There are three main ways you can attempt parallel processing using Glide:

- Method 1: Parallelization *within* nodes such as `ProcessPoolSubmit` or a distributed processing extension such as Dask/Celery/Redis Queue
- Method 2: Completely parallel pipelines via `ParaGliders` (each process executes the entire pipeline)
- Method 3: Branched parallelism using parallel push nodes such as `ProcessPoolPush` or `ThreadPoolPush`

Each has its own use cases. Method 1 is perhaps the most straightforward since you can return to single process operation after the node is done doing whatever it needed to do in parallel. Method 2 may be useful and easy to understand in certain cases as well. Method 3 can lead to more complex/confusing flows and should probably only be used towards the end of pipelines to branch the output in parallel, such as if writing to several databases in parallel as a final step.

Please see the quickstart or tests for examples of each method.

Note: Combining the approaches may not work and has not been tested. Standard limitations apply regarding what types of data can be serialized and passed to a parallel process.

CHAPTER

ELEVEN

COMMON PIPELINES

`Glide` comes with some common, templated ETL pipelines that connect combinations of common nodes. The names are generally of the format "Source2Destination". The names of the available pipelines are listed in the *glide.pipelines* module documentation.

To use these pipelines, simply call the template to get an instance of a Glider, such as:

```
glider = File2Email()
glider.consume([file1, file2], load=dict(client=my_smtp_cient))
```

By default these templated pipelines have a `PlaceholderNode` named "transform" that you can easily replace once the glider is created:

```
glider["transform"] = MyTransformerNode("transform")
glider.consume(...)
```

You can also override the `Glider` class used to create the pipeline:

```
glider = File2Email(glider=ProcessPoolParaGlider)
```

All of these templated pipelines are simply a convenience and are meant to cover very simple cases. More often than not it's likely best to create your own explicit pipelines. Glide package

CHAPTER

TWELVE

SUBPACKAGES

12.1 glide.extensions package

12.1.1 Submodules

glide.extensions.celery module

http://www.celeryproject.org/

This extension assumes you have setup your own celery app and imported/added provided tasks to your app as necessary. Any code used by your Celery workers must be importable by those workers, and you may need to make sure your app allows pickle for serialization

class glide.extensions.celery.**CeleryApplyAsync**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

A Node that calls apply_async on a given Celery Task

run(*data*, *task*, *timeout=None*, *push_type='async'*, ***kwargs*)

Call task.apply_async with the given data as the first task argument

Parameters

- **data** – Data to process
- **task** – A Celery Task registered with your app.
- **timeout** (*int, optional*) – A timeout to use if waiting for results via AsyncResult.get()
- **push_type** (*str, optional*) – If "async", push the AsyncResult immediately. If "input", push the input data immediately after task submission. If "result", collect the task result synchronously and push it.
- ****kwargs** – Keyword arguments pass to task.apply_async

class glide.extensions.celery.**CeleryParaGlider**(*consume_task*, **args*, ***kwargs*)

Bases: *glide.core.ParaGlider*

A ParaGlider that uses Celery to execute parallel calls to consume()

Parameters

- **consume_task** – A Celery Task that will behave like consume()
- ***args** – Arguments passed through to ParaGlider init
- ****kwargs** – Keyword arguments passed through to ParaGlider init

consume_task
A Celery Task that behaves like consume(), such as CeleryConsumeTask.

See ParaGlider for additional attributes.

consume(*data=None*, *cleanup=None*, *split_count=None*, *synchronous=False*, *timeout=None*, ***node_contexts*)
Setup node contexts and consume data with the pipeline

Parameters

- **data** (*iterable, optional*) – Iterable of data to consume
- **cleanup** (*dict, optional*) – A mapping of arg names to clean up functions to be run after data processing is complete.
- **split_count** (*int, optional*) – How many slices to split the data into for parallel processing. Default is to inspect the celery app and set split_count = worker count.
- **synchronous** (*bool, optional*) – If False, return AsyncResults. If True, wait for tasks to complete and return their results, if any.
- **timeout** (*int or float, optional*) – If waiting for results, pass this as timeout to AsyncResult.get().
- ****node_contexts*** – Keyword arguments that are node_name->param_dict

class glide.extensions.celery.**CeleryReduce**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.flow.Reduce*

Collect the asynchronous results before pushing

end()
Do the push once all results are in

class glide.extensions.celery.**CelerySendTask**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

A Node that calls app.send_task

run(*data*, *app*, *task_name*, *timeout=None*, *push_type='async'*, ***kwargs*)
Call app.send_task with the given data as the first task argument

Parameters

- **data** – Data to process
- **app** – Celery app
- **task_name** – A name of a Celery Task registered with your app.
- **timeout** (*int, optional*) – A timeout to use if waiting for results via AsyncResult.get()
- **push_type** (*str, optional*) – If "async", push the AsyncResult immediately. If "input", push the input data immediately after task submission. If "result", collect the task result synchronously and push it.
- ****kwargs*** – Keyword arguments pass to task.send_task

glide.extensions.dask module

https://docs.dask.org/en/latest/

class glide.extensions.dask.**DaskClientMap**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.PoolSubmit*

Apply a transform to a Pandas DataFrame using dask Client

check_data(*data*)
Optional input data check

get_executor(***executor_kwargs*)
Override this to return the parallel executor

get_results(*futures*, *timeout=None*)
Override this to fetch results from an asynchronous task

get_worker_count(*executor*)
Override this to return a count of workers active in the executor

shutdown_executor(*executor*)
Override this to shutdown the executor

submit(*executor*, *func*, *splits*, ***kwargs*)
Override this to submit work to the executor

class glide.extensions.dask.**DaskClientPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.flow.FuturesPush*

Use a dask Client to do a parallel push

as_completed_func = None

executor_class = None

run(**args*, ***kwargs*)
Subclasses will override this method to implement core node logic

class glide.extensions.dask.**DaskDataFrameApply**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

Apply a transform to a Pandas DataFrame using dask dataframe

run(*df*, *func*, *from_pandas_kwargs=None*, ***kwargs*)
Convert to dask dataframe and use apply()

NOTE: it may be more efficient to not convert to/from Dask Dataframe in this manner depending on the pipeline

Parameters

- **df** (*pandas.DataFrame*) – The pandas DataFrame to apply func to
- **func** (*callable*) – A callable that will be passed to Dask DataFrame.apply
- **from_pandas_kwargs** (*optional*) – Keyword arguments to pass to dask.dataframe.from_pandas
- ****kwargs** – Keyword arguments passed to Dask DataFrame.apply

class glide.extensions.dask.**DaskDelayedPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.PushNode*

Use dask delayed to do a parallel push

class glide.extensions.dask.**DaskFuturesReduce**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.flow.Reduce*

Collect the asynchronous results before pushing

end()

Do the push once all Futures results are in.

> **Warning:** Dask futures will not work if you have closed your client connection!

class glide.extensions.dask.**DaskParaGlider**(**args*, *executor_kwargs=None*, ***kwargs*)

Bases: *glide.core.ParaGlider*

A ParaGlider that uses a dask Client to execute parallel calls to consume()

get_executor()

Override this method to create the parallel executor

get_results(*futures*, *timeout=None*)

Override this method to get the asynchronous results

get_worker_count(*executor*)

Override this method to get the active worker count from the executor

glide.extensions.pandas module

https://pandas.pydata.org/

class glide.extensions.pandas.**DataFrameApplyMap**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

Apply a transform to a Pandas DataFrame

run(*df*, *func*, ***kwargs*)

Use applymap() on a DataFrame

Parameters

- **df** (*pandas.DataFrame*) – The pandas DataFrame to apply func to
- **func** (*callable*) – A callable that will be passed to df.applymap
- ****kwargs** – Keyword arguments passed to applymap

class glide.extensions.pandas.**DataFrameBollingerBands**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.extensions.pandas.DataFrameRollingNode*

Compute bollinger bands for the specified columns in a DataFrame

compute_stats(*df*, *rolling*, *column_name*)

Override this to implement logic to manipulate the DataFrame

class glide.extensions.pandas.**DataFrameCSVExtract**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.extensions.pandas.DataFramePush*

Extract data from a CSV using Pandas

run(*f*, ***kwargs*)

Extract data for input file and push as a DataFrame

Parameters

- **f** – file or buffer to be passed to pandas.read_csv
- ****kwargs** – kwargs to be passed to pandas.read_csv

class glide.extensions.pandas.**DataFrameCSVLoad**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

Load data into a CSV from a Pandas DataFrame

begin()

Initialize state for CSV writing

end()

Reset state in case the node gets reused

run(*df*, *f*, *push_file=False*, *dry_run=False*, ***kwargs*)

Use Pandas to_csv to output a DataFrame

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to load to a CSV
- **f** (*file or buffer*) – File to write the DataFrame to
- **push_file** (*bool, optional*) – If true, push the file forward instead of the data
- **dry_run** (*bool, optional*) – If true, skip actually loading the data
- ****kwargs** – Keyword arguments passed to DataFrame.to_csv

class glide.extensions.pandas.**DataFrameExcelExtract**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.extensions.pandas.DataFramePush*

Extract data from an Excel file using Pandas

run(*f*, ***kwargs*)

Extract data for input file and push as a DataFrame. This will push a DataFrame or dict of DataFrames in the case of reading multiple sheets from an Excel file.

Parameters

- **f** – file or buffer to be passed to pandas.read_excel
- ****kwargs** – kwargs to be passed to pandas.read_excel

class glide.extensions.pandas.**DataFrameExcelLoad**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

Load data into an Excel file from a Pandas DataFrame

run(*df_or_dict*, *f*, *push_file=False*, *dry_run=False*, ***kwargs*)

Use Pandas to_excel to output a DataFrame

Parameters

- **df_or_dict** – DataFrame or dict of DataFrames to load to an Excel file. In the case of a dict the keys will be the sheet names.
- **f** (*file or buffer*) – File to write the DataFrame to
- **push_file** (*bool, optional*) – If true, push the file forward instead of the data
- **dry_run** (*bool, optional*) – If true, skip actually loading the data
- ****kwargs** – Keyword arguments passed to DataFrame.to_excel

class glide.extensions.pandas.**DataFrameHTMLExtract**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

Extract data from HTML tables using Pandas

run(*f*, ***kwargs*)

Extract data for input file and push as a DataFrame

Parameters

- **f** – file or buffer to be passed to pandas.read_html
- ****kwargs** – kwargs to be passed to pandas.read_html

class glide.extensions.pandas.**DataFrameHTMLLoad**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

run(*df*, *f*, *push_file=False*, *dry_run=False*, ***kwargs*)

Use Pandas to_html to output a DataFrame

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to load to an HTML file
- **f** (*file or buffer*) – File to write the DataFrame to
- **push_file** (*bool, optional*) – If true, push the file forward instead of the data
- **dry_run** (*bool, optional*) – If true, skip actually loading the data
- ****kwargs** – Keyword arguments passed to DataFrame.to_html

class glide.extensions.pandas.**DataFrameMethod**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

Helper to execute any pandas DataFrame method

run(*df*, *method*, ***kwargs*)

Helper to execute any pandas DataFrame method

Parameters

- **df** (*pandas.DataFrame*) – DataFrame object used to run the method
- **method** (*str*) – A name of a valid DataFrame method
- ****kwargs** – Arguments to pass to the DataFrame method

class glide.extensions.pandas.**DataFrameMovingAverage**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.extensions.pandas.DataFrameRollingNode*

Compute a moving average on a DataFrame

compute_stats(*df*, *rolling*, *column_name*)
Override this to implement logic to manipulate the DataFrame

class glide.extensions.pandas.**DataFramePush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*, *glide.extensions.pandas.DataFramePushMixin*

Base class for DataFrame-based nodes

class glide.extensions.pandas.**DataFramePushMixin**
Bases: object

Shared logic for DataFrame-based nodes

do_push(*df*, *chunksize=None*)
Push the DataFrame to the next node, obeying chunksize if passed

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to push, or chunks of a DataFrame if the chunksize argument is passed and truthy.
- **chunksize** (*int, optional*) – If truthy the df argument is expected to be chunks of a DataFrame that will be pushed individually.

class glide.extensions.pandas.**DataFrameRollingNode**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Apply df.rolling to a DataFrame

compute_stats(*df*, *rolling*, *column_name*)
Override this to implement logic to manipulate the DataFrame

run(*df*, *windows*, *columns=None*, *suffix=None*, ***kwargs*)
Use df.rolling to apply a rolling window calculation on a dataframe

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rolling.html

Parameters

- **df** (*pandas.DataFrame*) – The pandas DataFrame to process
- **windows** (*int or list of ints*) – Size(s) of the moving window(s). If a list, all windows will be calculated and the window size will be appended as a suffix.
- **columns** (*list, optional*) – A list of columns to calculate values for
- **suffix** (*str, optional*) – A suffix to add to the column names of calculated values
- ****kwargs** – Keyword arguments passed to df.rolling

class glide.extensions.pandas.**DataFrameRollingStd**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.extensions.pandas.DataFrameRollingNode*

Compute a rolling standard deviation on a DataFrame

compute_stats(*df*, *rolling*, *column_name*)
Override this to implement logic to manipulate the DataFrame

class glide.extensions.pandas.**DataFrameRollingSum**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.extensions.pandas.DataFrameRollingNode*

Compute a rolling window sum on a DataFrame

compute_stats(*df*, *rolling*, *column_name*)
Override this to implement logic to manipulate the DataFrame

class glide.extensions.pandas.**DataFrameSQLExtract**(**args*, ***kwargs*)
Bases: *glide.extensions.pandas.PandasSQLNode*

Extract data from a SQL db using Pandas

run(*sql*, *conn*, ***kwargs*)
Extract data for input query and push as a DataFrame

Parameters

- **sql** – SQL query to pass to pandas.read_sql
- **conn** – A SQL database connection
- ****kwargs** – kwargs to be passed to pandas.read_sql

class glide.extensions.pandas.**DataFrameSQLLoad**(**args*, ***kwargs*)
Bases: *glide.extensions.pandas.PandasSQLNode*

Load data into a SQL db from a Pandas DataFrame

run(*df*, *conn*, *table*, *push_table=False*, *dry_run=False*, ***kwargs*)
Use Pandas to_sql to output a DataFrame

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to load to a SQL table
- **conn** – Database connection
- **table** (*str*) – Name of a table to write the data to
- **push_table** (*bool, optional*) – If true, push the table forward instead of the data
- **dry_run** (*bool, optional*) – If true, skip actually loading the data
- ****kwargs** – Keyword arguments passed to DataFrame.to_sql

class glide.extensions.pandas.**DataFrameSQLTableExtract**(**args*, ***kwargs*)
Bases: *glide.extensions.pandas.PandasSQLNode*

Extract data from a SQL table using Pandas

run(*table*, *conn*, *where=None*, *limit=None*, ***kwargs*)
Extract data for input table and push as a DataFrame

Parameters

- **table** (*str*) – SQL table to query
- **conn** – A SQL database connection
- **where** (*str, optional*) – A SQL where clause
- **limit** (*int, optional*) – Limit to put in SQL limit clause
- ****kwargs** – kwargs to be passed to pandas.read_sql

class glide.extensions.pandas.**DataFrameSQLTempLoad**(**args*, ***kwargs*)
Bases: *glide.extensions.pandas.PandasSQLNode*

Load data into a SQL temp table from a Pandas DataFrame

run(*df*, *conn*, *schema=None*, *dry_run=False*, ***kwargs*)
Use Pandas to_sql to output a DataFrame to a temporary table. Push a reference to the temp table forward.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to load to a SQL table
- **conn** – Database connection
- **schema** (*str, optional*) – schema to create the temp table in
- **dry_run** (*bool, optional*) – If true, skip actually loading the data
- ****kwargs** – Keyword arguments passed to DataFrame.to_sql

class glide.extensions.pandas.**FromDataFrame**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

run(*df*, *orient='records'*, ***kwargs*)

Push the DataFrame to the next node, obeying chunksize if passed

Parameters

- **df** – A DataFrame to convert to an iterable of records
- **orient** – The orient arg passed to df.to_dict()
- ****kwargs** – Keyword arguments passed to df.to_dict()

class glide.extensions.pandas.**PandasSQLNode**(**args*, ***kwargs*)

Bases: *glide.sql.BaseSQLNode*, *glide.extensions.pandas.DataFramePushMixin*

Captures the connection types allowed to work with Pandas to_sql/from_sql

allowed_conn_types = [<class 'sqlalchemy.engine.base.Connection'>, <class 'sqlalchemy.

class glide.extensions.pandas.**ToDataFrame**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

run(*rows*, ***kwargs*)

Convert the rows to a DataFrame

Parameters

- **rows** – An iterable of rows to convert to a DataFrame
- ****kwargs** – Keyword arguments passed to from_records()

glide.extensions.rq module

http://python-rq.org/docs/

class glide.extensions.rq.**RQJob**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

A Node that queues a function using Redis Queue

> **Warning:** Python RQ seems to not update the job status if your function does not return a non-None value. Your code may hang if you poll waiting for a result in this scenario.

run(*data*, *func*, *queue=None*, *queue_name='default'*, *redis_conn=None*, *push_type='async'*, *poll_sleep=1*, *timeout=None*, ***kwargs*)

Execute func on data using Redis Queue

Parameters

- **data** – Data to process
- **func** (*callable*) – Function to execute using Redis Queue
- **queue** (*Queue, optional*) – An rq Queue object
- **queue_name** (*str, optional*) – When creating a queue, the name of the queue to use
- **redis_conn** (*type, optional*) – When creating a queue, the redis connection to use
- **push_type** (*type, optional*) – If "async", push the Job immediately. If "input", push the input data immediately after task submission. If "result", collect the task result synchronously and push it.
- **poll_sleep** (*int or float, optional*) – If waiting for the result, sleep this many seconds between polls
- **timeout** (*int or float, optional*) – If waiting for result, raise an exception if polling for all results takes longer than timeout seconds.
- ****kwargs** – Keyword arguments to pass to enqueue()

class glide.extensions.rq.**RQParaGlider**(*queue*, **args*, ***kwargs*)

Bases: *glide.core.ParaGlider*

A ParaGlider that uses Redis Queue to execute parallel calls to consume()

Parameters

- **queue** – An rq Queue object
- ***args** – Arguments passed through to ParaGlider init
- ****kwargs** – Keyword arguments passed through to ParaGlider init

queue

An rq Queue object

See ParaGlider for additional attributes.

consume(*data=None*, *cleanup=None*, *split_count=None*, *synchronous=False*, *timeout=None*, ***node_contexts*)

Setup node contexts and consume data with the pipeline

Parameters

- **data** (*iterable, optional*) – Iterable of data to consume
- **cleanup** (*dict, optional*) – A mapping of arg names to clean up functions to be run after data processing is complete.
- **split_count** (*int, optional*) – How many slices to split the data into for parallel processing. Default is the number of workers in the provided queue.
- **synchronous** (*bool, optional*) – If False, return Jobs. If True, wait for jobs to complete and return their results, if any.
- **timeout** (*int or float, optional*) – If waiting for results, raise an exception if polling for all results takes longer than timeout seconds.
- ****node_contexts** – Keyword arguments that are node_name->param_dict

class glide.extensions.rq.**RQReduce**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.Reduce*

Collect asynchronous results before pushing

end()
Do the push once all results are in

exception glide.extensions.rq.**RQTimeoutException**
Bases: Exception

Exception for timeouts polling for results

glide.extensions.rq.**complete_count**(*async_results*)
TODO: Would it be better to rely on the job registry instead of job.result?

— Example: from rq import job from rq.registry import FinishedJobRegistry registry = FinishedJobRegistry('default', connection=redis_conn) job_ids = registry.get_job_ids() job_obj = job.Job.fetch("job-id-here", connection=redis_conn)

glide.extensions.rq.**get_async_result**(*async_result*, *timeout=None*)
Poll for a result

glide.extensions.rq.**get_async_results**(*async_results*, *timeout=None*)
Poll for results

glide.extensions.rq.**rq_consume**(**args*, ***kwargs*)
Hack: RQ only seems to update the job status if your function returns a non-None value. To force that, we use this simple wrapper around consume().

glide.extensions.swifter module

https://github.com/jmcarpenter2/swifter

class glide.extensions.swifter.**SwifterApply**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Apply a Swifter transform to a Pandas DataFrame

run(*df*, *func*, *threads=False*, ***kwargs*)
Use Swifter apply() on a DataFrame

Parameters

- **df** (*pandas.DataFrame*) – The pandas DataFrame to apply func to
- **func** (*callable*) – A callable that will be passed to df.swifter.apply
- **threads** (*bool*) – If true use the "threads" scheduler, else "processes"
- ****kwargs** – Keyword arguments passed to Dask df.swifter.apply

CHAPTER

THIRTEEN

SUBMODULES

13.1 glide.core module

Core classes used to power pipelines

class glide.core.**AssertFunc**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

run(*data*, *func*)
Assert that a function returns a truthy value

Parameters

- **data** – Data to push if func(self, data) is truthy
- **func** (*callable*) – A callable that accepts (node, data) args and returns a truthy value if the assertion should pass.

class glide.core.**AsyncIOSubmit**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

A node that splits input data over an async function

get_results(*futures*, *timeout=None*)

run(*data*, *func*, *split_count=None*, *timeout=None*, *push_type='async'*)
Use a asyncio to apply func to data

Parameters

- **data** – An iterable to process
- **func** (*callable*) – A async callable that will be passed data to operate on using asyncio.
- **split_count** (*int, optional*) – How many slices to split the data into for concurrent processing. Default is to set split_count = len(data).
- **timeout** (*int or float, optional*) – Time to wait for jobs to complete before raising an error. Ignored unless using a push_type that waits for results.
- **push_type** (*str, optional*) – If "async", push the Futures immediately. If "input", push the input data immediately after task submission. If "result", collect the result synchronously and push it.

class glide.core.**ConfigContext**(*filename=None*, *var=None*, *key=None*)
Bases: *glide.core.RuntimeContext*

class glide.core.**ContextPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Pass context to downstream nodes

run(*data*, *func*, *propagate=False*)
Pass dynamically generated context to downstream nodes

Parameters

- **data** – Data being processed by the node.
- **func** (*callable*) – A function that takes the node object and data as input and returns a context dict to be used to update/add downstream node context.
- **propagate** (*bool, optional*) – Passed through to node.update_downstream_context()

class glide.core.**GlidePipeline**(*node*, *global_state=None*)
Bases: consecution.pipeline.Pipeline

end()
Override this method to execute any logic you want to perform after all nodes are done processing data. The .end() method of all nodes will be called.

class glide.core.**Glider**(**args*, ***kwargs*)
Bases: object

Main class for forming and executing pipelines. It thinly wraps Consecution's Pipeline, but does not subclass it due to a bug in pickle that hits an infinite recursion when using multiprocessing with a super().func reference.

Parameters

- ***args** – Arguments passed through to Consecution's Pipeline class.
- ****kwargs** – Keyword arguments passed through to Consecution's Pipeline class.

pipeline
A Consecution Pipeline

cli(**script_args*, *blacklist=None*, *parents=None*, *inject=None*, *cleanup=None*)
Generate a decorator for this Glider that can be used to expose a CLI

Parameters

- ***script_args** – Arguments to be added to the script CLI
- **blacklist** (*list, optional*) – List of arguments to filter from appearing in the CLI
- **parents** (*list, optional*) – List of parent CLIs to inherit from
- **inject** (*dict, optional*) – A dictionary of arg names to functions/values that inject a value for that arg. Those args will be passed as context to nodes that can accept them in their run() method.
- **cleanup** (*dict, optional*) – A dictionary of arg names to callables that will be used to perform clean up when the CLI script is complete.

Returns **decorator** – A decorator that can be used to turn a function into a CLI "main" function.

Return type *GliderScript*

consume(*data=None*, *cleanup=None*, ***node_contexts*)
Setup node contexts and consume data with the pipeline

Parameters

- **data** (*iterable, optional*) – Iterable of data to consume
- **cleanup** (*dict, optional*) – A mapping of arg names to clean up functions to be run after data processing is complete.
- ****node_contexts** – Keyword arguments that are node_name->param_dict

get_node_lookup()
Passthrough to Consecution Pipeline._node_lookup

property global_state
Get the pipeline global_state attribute

plot(**args*, ***kwargs*)
Passthrough to Consecution Pipeline.plot

property top_node
Get the pipeline top_node attribute

class glide.core.**GliderScript**(*glider*, **script_args*, *blacklist=None*, *parents=None*, *inject=None*, *cleanup=None*)
Bases: tlbx.cli_utils.Script

A decorator that can be used to create a CLI from a Glider pipeline

Parameters

- **glider** (Glider) – A Glider pipeline to be used to auto-generate the CLI
- ***script_args** – Arguments to be added to the script CLI
- **blacklist** (*list, optional*) – List of arguments to filter from appearing in the CLI
- **parents** (*list, optional*) – List of parent CLIs to inherit from
- **inject** (*dict, optional*) – A dictionary of arg names to functions/values that inject a value for that arg. Those args will be passed as context to nodes that can accept them in their run() method.
- **cleanup** (*dict, optional*) – A dictionary of arg names to callables that will be used to perform clean up when the CLI script is complete.

blacklisted(*node_name*, *arg_name*)
Determine if an argument has been blacklisted from the CLI

clean_up(***kwargs*)
Override Script method to do any required clean up

get_injected_kwargs()
Override Script method to return populated kwargs from inject arg

class glide.core.**GlobalState**(***kwargs*)
Bases: tlbx.object_utils.MappingMixin, consecution.pipeline.GlobalState

Consecution GlobalState with more dict-like behavior

class glide.core.**GroupByNode**(**args*, ***kwargs*)
Bases: *glide.core.Node*

This approach was copied from Consecution. It batches data by key and then pushes once the key changes. For that reason it requires sorting ahead of time to function properly. It may make sense to provide different behavior.

key(*data*)

process(*data*)
Required method used by Consecution to process nodes

class glide.core.**NoInputNode**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

A node that does not take a data positional arg in run() and is expected to generate data to be pushed

run_requires_data = False

class glide.core.**Node**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: consecution.nodes.Node

Override Consecution's Node class to add necessary functionality

Parameters

- **name** (*str*) – The name of the Node.
- **_log** (*bool, optional*) – If true, log data processed by the node.
- **_debug** (*bool, optional*) – If true, drop into PDB right before calling run() for this node.
- ****default_context** – Keyword args that establish the default_context of the Node. Note that this context is copy.deepcopy'd on init, so any value in default_context must be usable by deepcopy.

name
The name of the Node.

Type str

_log
If true, log data processed by the node. Note that this overrides Consecution's log() functionality.

Type bool

_debug
If true, drop into PDB right before calling run() for this node.

Type bool

default_context
A dictionary to establish default context for the node that can be used to populate run() arguments.

Type dict

context
The current context of the Node

Type dict

run_args
An OrderedDict of positional args to run()

Type dict

run_kwargs
An OrderedDict of keyword args and defaults to run()

Type dict

run_requires_data
If true, the first positional arg to run is expected to be the data to process

Type bool

process(*data*)
Required method used by Consecution to process nodes

reset_context()
Reset context dict for this Node to the default

run(*data*, ***kwargs*)
Subclasses will override this method to implement core node logic

run_requires_data = True

set_global_results(*results*)

update_context(*context*)
Update the context dict for this Node

update_downstream_context(*context*, *propagate=False*)
Update the run context of downstream nodes

Parameters

- **context** (*dict*) – A dict used to update the context of downstream nodes
- **propagate** (*bool, optional*) – If true, propagate the update to all child nodes in the DAG. The default behavior is to only push updates to the immediate downstream nodes.

class glide.core.**ParaGlider**(**args*, *executor_kwargs=None*, ***kwargs*)
Bases: *glide.core.Glider*

Parameters

- ***args** – Arguments passed through to Glider
- **executor_kwargs** (*dict, optional*) – A dict of keyword arguments to pass to the process or thread executor
- ****kwargs** – Keyword arguments passed through to Glider

pipeline
A Consecution Pipeline

executor_kwargs
A dict of keyword arguments to pass to the process or thread executor

consume(*data=None*, *cleanup=None*, *split_count=None*, *synchronous=False*, *timeout=None*, ***node_contexts*)
Setup node contexts and consume data with the pipeline

Parameters

- **data** (*iterable, optional*) – Iterable of data to consume
- **cleanup** (*dict, optional*) – A mapping of arg names to clean up functions to be run after data processing is complete.
- **split_count** (*int, optional*) – How many slices to split the data into for parallel processing. Default is to use executor._max_workers.
- **synchronous** (*bool, optional*) – If False, return Futures. If True, wait for futures to complete and return their results, if any.
- **timeout** (*int or float, optional*) – Raises a concurrent.futures.TimeoutError if __next__() is called and the result isn't available after timeout seconds from the original call to as_completed(). Ignored if synchronous=False.

- ****node_contexts** – Keyword arguments that are node_name->param_dict

get_executor()
Override this method to create the parallel executor

get_results(*futures*, *timeout=None*)
Override this method to get the asynchronous results

get_worker_count(*executor*)
Override this method to get the active worker count from the executor

class glide.core.**PlaceholderNode**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.PushNode*

Used as a placeholder in pipelines. Will pass values through by default

class glide.core.**PoolSubmit**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Apply a function to the data in parallel

check_data(*data*)
Optional input data check

get_executor(***executor_kwargs*)
Override this to return the parallel executor

get_results(*futures*, *timeout=None*)
Override this to fetch results from an asynchronous task

get_worker_count(*executor*)
Override this to return a count of workers active in the executor

run(*data*, *func*, *executor=None*, *executor_kwargs=None*, *split_count=None*, *timeout=None*, *push_type='async'*, ***kwargs*)
Use a parallel executor to apply func to data

Parameters

- **data** – An iterable to process
- **func** (*callable*) – A callable that will be passed data to operate on in parallel
- **executor** (*Executor, optional*) – If passed use this executor instead of creating one.
- **executor_kwargs** (*dict, optional*) – Keyword arguments to pass when initalizing an executor.
- **split_count** (*int, optional*) – How many slices to split the data into for parallel processing. Default is to set split_count = number of workers
- **timeout** (*int or float, optional*) – Time to wait for jobs to complete before raising an error. Ignored unless using a push_type that waits for results.
- **push_type** (*str, optional*) – If "async", push the Futures immediately. If "input", push the input data immediately after task submission. If "result", collect the result synchronously and push it.
- ****kwargs** – Keyword arguments passed to the executor when submitting work

shutdown_executor(*executor*)
Override this to shutdown the executor

submit(*executor*, *func*, *splits*, ***kwargs*)
Override this to submit work to the executor

class glide.core.**ProcessPoolParaGlider**(**args*, *executor_kwargs=None*, ***kwargs*)
Bases: *glide.core.ParaGlider*

A parallel Glider that uses a ProcessPoolExecutor to execute parallel calls to consume()

get_executor()
Override this method to create the parallel executor

get_results(*futures*, *timeout=None*)
Override this method to get the asynchronous results

get_worker_count(*executor*)
Override this method to get the active worker count from the executor

class glide.core.**ProcessPoolSubmit**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.PoolSubmit*

A PoolExecutor that uses ProcessPoolExecutor

get_executor(***executor_kwargs*)
Override this to return the parallel executor

get_results(*futures*, *timeout=None*)
Override this to fetch results from an asynchronous task

get_worker_count(*executor*)
Override this to return a count of workers active in the executor

shutdown_executor(*executor*, ***kwargs*)
Override this to shutdown the executor

submit(*executor*, *func*, *splits*, ***kwargs*)
Override this to submit work to the executor

class glide.core.**Profile**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

A node that profiles the call to push(), thus profiling all downstream nodes

run(*data*, *filename=None*)
Profiles calls to push(), thus profiling all downstream nodes

Parameters

- **data** – Data to push
- **filename** (*str, optional*) – Filename to pass to runctx() to save stats

class glide.core.**PushNode**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

A node that just passes all data through in run()

run(*data*, ***kwargs*)
Subclasses will override this method to implement core node logic

class glide.core.**PushTypes**
Bases: object

The names of push strategies for nodes that support asynchronous execution

Async = 'async'

Input = 'input'

Result = 'result'

class glide.core.**RuntimeContext**(*func*, **args*, ***kwargs*)
Bases: object

A function to be executed at runtime to populate context values

Parameters

- **func** (*callable*) – The function to execute
- **args** – Positional arguments to pass to func when called
- **kwargs** – Keyword arguments to pass to func when called

copy()
Create a copy of this RuntimeContext referencing the same objects

class glide.core.**Shell**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.NoInputNode*

Run a local shell command using subprocess.run

run(*cmd*, *shell=False*, *capture_output=False*, ***kwargs*)
Run a local shell command using subprocess.run and push the return value

Parameters

- **cmd** (*list or str*) – Shell command to run. If passing a single string, either shell must be True or else the string must simply name the program to be executed without specifying any arguments.
- **shell** (*bool, optional*) – Arg passed through to subprocess.run
- **capture_output** (*bool, optional*) – Arg passed through to subprocess.run
- ****kwargs** – kwargs passed to subprocess.run

class glide.core.**ThreadPoolParaGlider**(**args*, *executor_kwargs=None*, ***kwargs*)
Bases: *glide.core.ProcessPoolParaGlider*

A parallel Glider that uses a ThreadPoolExecutor to execute parallel calls to consume()

get_executor()
Override this method to create the parallel executor

class glide.core.**ThreadPoolSubmit**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.ProcessPoolSubmit*

A PoolExecutor that uses ThreadPoolExecutor

get_executor(***executor_kwargs*)
Override this to return the parallel executor

glide.core.**clean_up_nodes**(*cleanup*, *contexts*)
Call clean up functions for node context objects

glide.core.**consume**(*pipeline*, *data*, *cleanup=None*, ***node_contexts*)
Handles node contexts before/after calling pipeline.consume()

Note: It would have been better to subclass Pipeline and implement this logic right before/after the core consume() call, but there is a bug in pickle that prevents that from working with multiprocessing.

---

glide.core.**consume_none**(*pipeline*)
This mimics the behavior of Consecution's consume() but allows for running a pipeline with no input data.

glide.core.**get_node_contexts**(*pipeline*)
Get a dict of node_name->node_context from pipeline

glide.core.**reset_node_contexts**(*pipeline*, *node_contexts*)
Helper function for resetting node contexts in a pipeline

glide.core.**update_node_contexts**(*pipeline*, *node_contexts*)
Helper function for updating node contexts in a pipeline

13.2 glide.extract module

A home for common data extraction nodes

Nodes:

- CSVExtract
- ExcelExtract
- SQLExtract
- SQLParamExtract
- SQLTableExtract
- FileExtract
- URLExtract
- EmailExtract

class glide.extract.**CSVExtract**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Extract data from a CSV

run(*f*, *compression=None*, *open_flags='r'*, *chunksize=None*, *nrows=None*, *reader=<class 'csv.DictReader'>*, ***kwargs*)
Extract data for input file and push dict rows

Parameters

- **f** (*file path or buffer*) – file path or buffer to read CSV
- **compression** (*str, optional*) – param passed to pandas get_filepath_or_buffer
- **open_flags** (*str, optional*) – Flags to pass to open() if f is not already an opened buffer
- **chunksize** (*int, optional*) – Read data in chunks of this size
- **nrows** (*int, optional*) – Limit to reading this number of rows
- **reader** (*csv Reader, optional*) – The CSV reader class to use. Defaults to csv.DictReader
- ****kwargs** – keyword arguments passed to the reader

class glide.extract.**EmailExtract**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Extract data from an email inbox using IMAPClient: https://imapclient.readthedocs.io

run(*criteria*, *sort=None*, *folder='INBOX'*, *client=None*, *host=None*, *username=None*, *password=None*, *push_all=False*, *push_type='message'*, *limit=None*, ***kwargs*)
Extract data from an email inbox and push the data forward.

Note: Instances of IMAPClient are NOT thread safe. They should not be shared and accessed concurrently from multiple threads.

Parameters

- **criteria** (*str or list*) – Criteria argument passed to IMAPClient.search. See https://tools.ietf.org/html/rfc3501.html#section-6.4.4.
- **sort** (*str or list, optional*) – Sort criteria passed to IMAPClient.sort. Note that SORT is an extension to the IMAP4 standard so it may not be supported by all IMAP servers. See https://tools.ietf.org/html/rfc5256.
- **folder** (*str, optional*) – Folder to read emails from
- **client** (*optional*) – An established IMAPClient connection. If not present, the host/login information is required.
- **host** (*str, optional*) – The IMAP host to connect to
- **username** (*str, optional*) – The IMAP username for login
- **password** (*str, optional*) – The IMAP password for login
- **push_all** (*bool, optional*) – When true push all retrievd data/emails at once
- **push_type** (*str, optional*) – What type of data to extract and push from the emails. Options include:
  - **message**: push email.message.EmailMessage objects
  - **message_id**: push a list of message IDs that can be fetched
  - **all**: push a list of dict(message=<email.message.EmailMessages>, payload=<extracted payload>)
  - **body**: push a list of email bodies
  - **attachment**: push a list of attachments (an email with multiple attachments will be grouped in a sublist)
- **limit** (*int, optional*) – Limit to N rows
- ****kwargs** – Keyword arguments to pass IMAPClient if not client is passed

class glide.extract.**ExcelExtract**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Extract data from an Excel file

run(*f*, *dict_rows=False*, ***kwargs*)
Use pyexcel to read data from a file

Parameters

- **f** (*str or buffer*) – The Excel file to read. Multiple excel formats supported.
- **dict_rows** (*bool, optional*) – If true the rows of each sheet will be converted to dicts with column names as keys.
- ****kwargs** – Keyword arguments passed to pyexcel

class glide.extract.**FileExtract**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Extract raw data from a file

run(*f*, *compression=None*, *open_flags='r'*, *chunksize=None*, *push_lines=False*, *limit=None*)
Extract raw data from a file or buffer and push contents

Parameters

- **f** (*file path or buffer*) – File path or buffer to read
- **compression** (*str, optional*) – param passed to pandas get_filepath_or_buffer
- **open_flags** (*str, optional*) – Flags to pass to open() if f is not already an opened buffer
- **chunksize** (*int, optional*) – Push lines in chunks of this size
- **push_lines** (*bool, optional*) – Push each line as it's read instead of reading entire file and pushing
- **limit** (*int, optional*) – Limit to first N lines

class glide.extract.**SQLExtract**(**args*, ***kwargs*)
Bases: *glide.sql.SQLNode*

Generic SQL extract Node

run(*sql*, *conn*, *cursor=None*, *cursor_type=None*, *params=None*, *chunksize=None*, ***kwargs*)
Extract data for input query and push fetched rows.

Parameters

- **sql** (*str*) – SQL query to run
- **conn** – SQL connection object
- **cursor** (*optional*) – SQL connection cursor object
- **cursor_type** (*optional*) – SQL connection cursor type when creating a cursor is necessary
- **params** (*tuple or dict, optional*) – A tuple or dict of params to pass to the execute method
- **chunksize** (*int, optional*) – Fetch and push data in chunks of this size
- ****kwargs** – Keyword arguments pushed to the execute method

class glide.extract.**SQLParamExtract**(**args*, ***kwargs*)
Bases: *glide.extract.SQLExtract*

Generic SQL extract node that expects SQL params as data instead of a query

run(*params*, *sql*, *conn*, *cursor=None*, *cursor_type=None*, *chunksize=None*, ***kwargs*)
Extract data for input params and push fetched rows.

Parameters

- **params** (*tuple or dict*) – A tuple or dict of params to pass to the execute method
- **sql** (*str*) – SQL query to run
- **conn** – SQL connection object
- **cursor** (*optional*) – SQL connection cursor object

- **cursor_type** (*optional*) – SQL connection cursor type when creating a cursor is necessary
- **chunksize** (*int, optional*) – Fetch and push data in chunks of this size
- ****kwargs** – Keyword arguments pushed to the execute method

class glide.extract.**SQLTableExtract**(**args*, ***kwargs*)
Bases: *glide.sql.SQLNode*

Generic SQL table extract node

run(*table*, *conn*, *cursor=None*, *cursor_type=None*, *where=None*, *limit=None*, *params=None*, *chunksize=None*, ***kwargs*)
Extract data for input table and push fetched rows

Parameters

- **table** (*str*) – SQL table name
- **conn** – SQL connection object
- **cursor** (*optional*) – SQL connection cursor object
- **cursor_type** (*optional*) – SQL connection cursor type when creating a cursor is necessary
- **where** (*str, optional*) – SQL where clause
- **limit** (*int, optional*) – Limit to put in SQL limit clause
- **params** (*tuple or dict, optional*) – A tuple or dict of params to pass to the execute method
- **chunksize** (*int, optional*) – Fetch and push data in chunks of this size
- ****kwargs** – Keyword arguments passed to cursor.execute

class glide.extract.**URLExtract**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Extract data from a URL with requests

run(*request*, *data_type='content'*, *session=None*, *skip_raise=False*, *handle_paging=None*, *page_limit=None*, *push_pages=False*, ***kwargs*)
Extract data from a URL using requests and push response.content. Input request may be a string (GET that url) or a dictionary of args to requests.request:

http://2.python-requests.org/en/master/api/?highlight=get#requests.request

See the requests docs for information on authentication options:

https://requests.kennethreitz.org/en/master/user/authentication/

Parameters

- **request** (*str or dict*) – If str, a URL to GET. If a dict, args to requests.request
- **data_type** (*str, optional*) – One of "content", "text", or "json" to control extraction of data from requests response.
- **session** (*optional*) – A requests Session to use to make the request
- **skip_raise** (*bool, optional*) – If False, raise exceptions for bad response status
- **handle_paging** (*callable, optional*) – A callable that accepts the following params and updates the args that will be passed to requests.request in place. The callable

should return two values, the page data extracted from the API response and a flag denoting whether the last page has been reached. Arguments:

- **result**: the API result of the most recent request
- **request**: a request args dict to update

- **page_limit** (*int, optional*) – If passed, use as a cap of the number of pages pulled
- **push_pages** (*bool, optional*) – If true, push each page individually.
- ****kwargs** – Keyword arguments to pass to the request method. If a dict is passed for the request parameter it overrides values of this.

13.3 glide.flow module

class glide.flow.**ArraySplitByNode**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.SplitByNode*

A node that splits the data before pushing

get_splits(*data*, *split_count*)
Split the data into split_count slices

class glide.flow.**ArraySplitPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.SplitPush*

A node that splits the data before pushing

get_splits(*data*, *split_count*)
Split the data into split_count slices

class glide.flow.**AsyncIOFuturesReduce**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.Reduce*

Collect results from asyncio futures before pushing

The following are parameters that get pulled from the node context and used in end().

Parameters

- **flatten** (*bool, optional*) – Flatten the results into a single list before pushing
- **timeout** (*int or float, optional*) – Timeout to pass to asyncio.wait
- **close** (*bool, optional*) – Whether to call loop.close() after processing is done

end()
Do the push once all Futures results are in

class glide.flow.**DateTimeWindowPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.NoInputNode*

run(*start_date*, *end_date*, *window_size_hours=None*, *num_windows=None*, *reverse=False*, *add_second=True*)
Subclasses will override this method to implement core node logic

class glide.flow.**DateWindowPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.NoInputNode*

run(*start_date*, *end_date*, *reverse=False*)
Subclasses will override this method to implement core node logic

class glide.flow.**FileConcat**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Concat a set of input files into one output file

run(*files*, *f_out*, *in_flags='rb'*, *out_flags='wb'*, *push_input=False*)
Concat a set of input files into one output file

Parameters

- **f_in** (*file path or buffer*) – File path or buffer to read
- **f_out** (*file path or buffer*) – File path or buffer to write
- **in_flags** (*str, optional*) – Flags to use when opening the input file
- **out_flags** (*str, optional*) – Flags to use when opening the output file
- **push_input** (*bool, optional*) – If true, push f_in instead of f_out

class glide.flow.**FileCopy**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Copy one file to another

run(*f_in*, *f_out*, *in_flags='rb'*, *out_flags='wb'*, *push_input=False*)
Copy f_in to f_out and push file reference

Parameters

- **f_in** (*file path or buffer*) – File path or buffer to read
- **f_out** (*file path or buffer*) – File path or buffer to write
- **in_flags** (*str, optional*) – Flags to use when opening the input file
- **out_flags** (*str, optional*) – Flags to use when opening the output file
- **push_input** (*bool, optional*) – If true, push f_in instead of f_out

class glide.flow.**Flatten**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Flatten the input before pushing

run(*data*)
Flatten the input before pushing. Assumes data is in ~list of ~lists format

class glide.flow.**FuturesPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.PushNode*

A node that either splits or duplicates its input to pass to multiple downstream nodes in parallel according to the executor_class that supports the futures interface. If an executor_kwargs dict is in the context of this node it will be passed to the parallel executor.

Parameters **Node documentation for parameters** (*See*) –

executor_class
An Executor that will be used to parallelize the push

as_completed_func
A callable used to get the Futures results as completed

See Node documentation for additional attributes

as_completed_func(*timeout=None*)
An iterator over the given futures that yields each as it completes.

Parameters

- **fs** – The sequence of Futures (possibly created by different Executors) to iterate over.
- **timeout** – The maximum number of seconds to wait. If None, then there is no limit on the wait time.

Returns An iterator that yields the given Futures as they complete (finished or cancelled). If any given Futures are duplicated, they will be returned once.

Raises **TimeoutError** – If the entire result iterator could not be generated before the given timeout.

executor_class
alias of concurrent.futures.process.ProcessPoolExecutor

class glide.flow.**FuturesReduce**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.Reduce*

Collect results from futures before pushing

The following are parameters that get pulled from the node context and used in end().

Parameters

- **flatten** (*bool, optional*) – Flatten the results into a single list before pushing
- **timeout** (*int or float, optional*) – Timeout to pass to futures.as_completed()

end()
Do the push once all Futures results are in

class glide.flow.**IterPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Push each item of an iterable individually

run(*data*, ***kwargs*)
Subclasses will override this method to implement core node logic

class glide.flow.**Join**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Join iterables before pushing

run(*data*, *on=None*, *how='left'*, *rsuffixes=None*)
Join items before pushing. This converts each dataset to a DataFrame and reuses pandas join method under the hood.

Parameters

- **data** – The datasets to join (i.e. a list of datasets or DataFrames)
- **on** (*optional*) – Passed to the underlying pandas join method
- **how** (*str, optional*) – Passed to the underlying pandas join method
- **rsuffixes** (*list, optional*) – A list of suffixes to append to duplicate column names in the right datasets. The length of this should be len(data) - 1.

class glide.flow.**PollFunc**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

run(*data*, *func*, *result_param='status'*, *result_value='success'*, *sleep_time=2*, *max_iter=10*, *data_param=None*, ***kwargs*)
Poll a function for a result

Parameters

- **data** – Data to pass to func. Typically a request or URL that needs to be polled for a result.
- **func** (*callable*) – The function that will be called on each iteration to get a result. It is expected to return a dict with a key/value representing completion (see result_param/result_value).
- **result_param** (*str*) – The key to extract from the func result to look for success.
- **result_value** – The value representing success. Keep polling until this value is found.
- **sleep_time** (*float*) – The amount of time to sleep between iterations
- **max_iter** (*int*) – The maximum number of iterations before giving up
- **data_param** (*str, optional*) – If given, pull this param out of the func result on success and push. Otherwise push the full response from func.
- **kwargs** – Keyword arguments passed to func

class glide.flow.**ProcessPoolPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.FuturesPush*

A multi-process FuturesPushNode

class glide.flow.**Reduce**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Waits until end() to call push(), effectively waiting for all nodes before it to finish before continuing the pipeline.

The following are parameters that get pulled from the node context and used in end().

Parameters **flatten** (*bool, optional*) – Flatten the results into a single list before pushing

begin()
Setup a place for results to be collected

end()
Do the push once all results are in

run(*data*, ***kwargs*)
Collect results from previous nodes

class glide.flow.**Return**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.Reduce*

Collects upstream data and sets the result in the global state

Notes

Because this relies on the pipeline's global_state under the hood it will not work with pipelines that do process branching mid-pipeline such as ProcessPoolPush.

Parameters **flatten** (*bool, optional*) – Flatten the results into a single list before returning

end()
Collects upstream data and sets the result in the global state

class glide.flow.**SkipFalseNode**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

This overrides the behavior of calling run() such that if a "false" object is pushed it will never call run, just push to next node instead

class glide.flow.**SplitByNode**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.PushNode*

A node that splits the data based on the number of immediate downstream nodes.

If the data is a Pandas object it will use np.array_split, otherwise it will split the iterator into chunks of roughly equal size.

get_splits(*data*, *split_count*)
Split the data into split_count slices

class glide.flow.**SplitPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

A node that splits the data before pushing.

If the data is a Pandas object it will use np.array_split, otherwise it will split the iterator into chunks of roughly equal size.

get_splits(*data*, *split_count*)
Split the data into split_count slices

run(*data*, *split_count*, ***kwargs*)
Split the data and push each slice

class glide.flow.**ThreadPoolPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.FuturesPush*

A multi-threaded FuturesPushNode

executor_class
alias of concurrent.futures.thread.ThreadPoolExecutor

class glide.flow.**ThreadReduce**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.Reduce*

A plain-old Reducer with a name that makes it clear it works with threads

class glide.flow.**WindowPush**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

run(*data*, *size*, ***kwargs*)
Push windows of the specified size

Parameters

- **data** – The data to slice into windows
- **size** (*int*) – The window size

class glide.flow.**WindowReduce**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

begin()
Initialize a place for a window to be collected

run(*data*, *size*, ***kwargs*)
Collect results to fill and push windows

Parameters

- **data** – Data to collect into window
- **size** (*int*) – Size of window to collect

13.4 glide.filter module

A home for common filter nodes

Nodes:

- Filter
- DictKeyFilter
- AttributeFilter

class glide.filter.**AttributeFilter**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

A node that pushes a specific attribute of an object

run(*data*, *attribute*)
Given an object, extract and push an attribute

Parameters

- **data** – The object to pull the attribute from
- **attribute** – The attribute to read from the object

class glide.filter.**DictKeyFilter**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

A node that pushes a specific value from a dict-like object

run(*data*, *key*)
Given a dict-like object, extract and push a key

Parameters

- **data** (*dict-like*) – The dict-like object to extract the value from
- **key** (*hashable*) – The key to extract from data

class glide.filter.**Filter**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

A node that only pushes if some condition is met

run(*data*, *func*, ***kwargs*)
Subclasses will override this method to implement core node logic

13.5 glide.load module

A home for common data load nodes

Nodes:

- CSVLoad
- ExcelLoad
- SQLLoad
- SQLTempLoad
- FileLoad
- URLLoad

- EmailLoad
- Print
- PrettyPrint
- LenPrint
- ReprPrint
- FormatPrint

class glide.load.**CSVLoad**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.flow.SkipFalseNode*

Load data into a CSV using DictWriter

begin()
Initialize state for CSV writing

end()
Reset state in case the node gets reused

run(*rows*, *f*, *push_file=False*, *dry_run=False*, ***kwargs*)
Use DictWriter to output dict rows to a CSV.

Parameters

- **rows** – Iterable of rows to load to a CSV
- **f** (*file or buffer*) – File to write rows to
- **push_file** (*bool, optional*) – If true, push the file forward instead of the data
- **dry_run** (*bool, optional*) – If true, skip actually loading the data
- ****kwargs** – Keyword arguments passed to csv.DictWriter

class glide.load.**EmailLoad**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Load data to email via SMTP

run(*data*, *frm=None*, *to=None*, *subject=None*, *body=None*, *html=None*, *attach_as='attachment'*, *attachment_name=None*, *formatter=None*, *client=None*, *host=None*, *port=None*, *username=None*, *password=None*, *dry_run=False*)
Load data to email via SMTP.

Parameters

- **data** – EmailMessage or data to send. If the latter, the message will be created from the other node arguments.
- **frm** (*str, optional*) – The from email address
- **to** (*str or list, optional*) – A str or list of destination email addresses
- **subject** (*str, optional*) – The email subject
- **body** (*str, optional*) – The email text body
- **html** (*str, optional*) – The email html body
- **attach_as** (*str*) – Where to put the data in the email message if building the message from node arguments. Options: attachment, body, html.

- **attachment_name** (*str, optional*) – The file name to write the data to when attaching data to the email. The file extension will be used to infer the mimetype of the attachment. This should not be a full path as a temp directory will be created for this.
- **formatter** (*callable*) – A function to format and return a string from the input data if attach_as is set to "body" or "html".
- **client** (*optional*) – A connected smtplib.SMTP client
- **host** (*str, optional*) – The SMTP host to connect to if no client is provided
- **port** (*int, optional*) – The SMTP port to connect to if no client is provided
- **username** (*str, optional*) – The SMTP username for login if no client is provided
- **password** (*str, optional*) – The SMTP password for login if no client is provided
- **dry_run** (*bool, optional*) – If true, skip actually loading the data

class glide.load.**ExcelLoad**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.flow.SkipFalseNode*

Load data into an Excel file using pyexcel

run(*rows*, *f*, *dict_rows=False*, *sheet_name='Sheet1'*, *push_file=False*, *dry_run=False*, ***kwargs*)

Use DictWriter to output dict rows to a CSV.

Parameters

- **rows** – Iterable of rows to load to an Excel file, or a dict of sheet_name->iterable for multi-sheet loads.
- **f** (*file or buffer*) – File to write rows to
- **dict_rows** (*bool, optional*) – If true the rows of each sheet will be converted from dicts to lists
- **sheet_name** (*str, optional*) – Sheet name to use if input is an iterable of rows. Unused otherwise.
- **push_file** (*bool, optional*) – If true, push the file forward instead of the data
- **dry_run** (*bool, optional*) – If true, skip actually loading the data
- ****kwargs** – Keyword arguments passed to pyexcel

class glide.load.**FileLoad**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

Load raw content to a file

run(*data*, *f*, *open_flags='w'*, *push_file=False*, *dry_run=False*)

Load raw data to a file or buffer

Parameters

- **data** – Data to write to file
- **f** (*file path or buffer*) – File path or buffer to write
- **open_flags** (*str, optional*) – Flags to pass to open() if f is not already an opened buffer
- **push_file** (*bool*) – If true, push the file forward instead of the data
- **dry_run** (*bool, optional*) – If true, skip actually loading the data

class glide.load.**FormatPrint**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Format and print the data

run(*data*, *label=None*, *indent=None*, *color=None*, *autocolor=False*, *format_func='pf'*)
Format using tlx.format_msg, then print

Parameters

- **data** – The data to print
- ****kwargs** – Keyword arguments passed to tlx.format_msg

class glide.load.**LenPrint**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.load.Print*

Prints the length of the data

get_label()
Get a label for the print statement

print(*data*)
Print the data

class glide.load.**PrettyPrint**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.load.Print*

Pretty-prints the data

print(*data*)
Print the data

class glide.load.**Print**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Print the data

get_label()
Get a label for the print statement

print(*data*)
Print the data

run(*data*, *label=True*)
Print the data with the printer function and push

class glide.load.**ReprPrint**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.load.Print*

Prints the reprlib.repr of the data

print(*data*)
Print the data

class glide.load.**SQLLoad**(**args*, ***kwargs*)
Bases: *glide.sql.SQLNode*

Generic SQL loader

run(*rows*, *conn*, *table*, *cursor=None*, *commit=True*, *rollback=False*, *stmt_type='REPLACE'*, *odku=False*, *swap=False*, *keep_old=False*, *push_data=False*, *dry_run=False*)
Form SQL statement and use bulk execute to write rows to table

Parameters

- **rows** – Iterable of rows to load to the table

- **conn** – Database connection
- **table** (*str*) – Name of a table to write the data to
- **cursor** (*optional*) – Database connection cursor
- **commit** (*bool, optional*) – If true try to commit the transaction. If your connection autocommits this will have no effect. If this is a SQLAlchemy connection and you are in a transaction, it will try to get a reference to the current transaction and call commit on that.
- **rollback** (*bool, optional*) – If true try to rollback the transaction on exceptions. Behavior may vary by backend DB library if you are not currently in a transaction.
- **stmt_type** (*str, optional*) – Type of SQL statement to use (REPLACE, INSERT, etc.). **Note:** Backend support for this varies.
- **odku** (*bool or list, optional*) – If true, add ON DUPLICATE KEY UPDATE clause for all columns. If a list then only add it for the specified columns. **Note:** Backend support for this varies.
- **swap** (*bool, optional*) – If true, load a table and then swap it into the target table via rename. Not supported with all database back ends.
- **keep_old** (*bool, optional*) – If true and swapping tables, keep the original table with a __old suffix added to the name
- **push_data** (*bool, optional*) – If true, push the data forward instead of the table name
- **dry_run** (*bool, optional*) – If true, skip actually loading the data

class glide.load.**SQLTempLoad**(**args*, ***kwargs*)
Bases: *glide.sql.SQLNode*

Generic SQL temp table loader

run(*rows*, *conn*, *cursor=None*, *schema=None*, *commit=True*, *rollback=False*, *dry_run=False*)
Create and bulk load a temp table

Parameters

- **rows** – Iterable of rows to load to the table
- **conn** – Database connection
- **cursor** (*optional*) – Database connection cursor
- **schema** (*str, optional*) – Schema to create temp table in
- **commit** (*bool, optional*) – If true try to commit the transaction. If your connection autocommits this will have no effect. If this is a SQLAlchemy connection and you are in a transaction, it will try to get a reference to the current transaction and call commit on that.
- **rollback** (*bool, optional*) – If true try to rollback the transaction on exceptions. Behavior may vary by backend DB library if you are not currently in a transaction.
- **dry_run** (*bool, optional*) – If true, skip actually loading the data

class glide.load.**URLLoad**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Load data to URL with requests

run(*data*, *url*, *data_param='data'*, *session=None*, *skip_raise=False*, *dry_run=False*, ***kwargs*)
Load data to URL using requests and push response.content. The url maybe be a string (POST that url) or a dictionary of args to requests.request:

http://2.python-requests.org/en/master/api/?highlight=get#requests.request

Parameters

- **data** – Data to load to the URL
- **url** (*str or dict*) – If str, a URL to POST to. If a dict, args to requets.request
- **data_param** (*str, optional*) – parameter to stuff data in when calling requests methods
- **session** (*optional*) – A requests Session to use to make the request
- **skip_raise** (*bool, optional*) – if False, raise exceptions for bad response status
- **dry_run** (*bool, optional*) – If true, skip actually loading the data
- ****kwargs** – Keyword arguments to pass to the request method. If a dict is passed for the url parameter it overrides values here.

13.6 glide.math module

class glide.math.**Average**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Push the average of the input

run(*data*)
Take the average of data and push it

class glide.math.**Sum**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Push the sum of the input

run(*data*)
Take the sum of data and push it

13.7 glide.pipelines module

Basic ETL pipeline templates for common nodes

Pipelines:

- SQL2SQL
- SQL2CSV
- SQLParam2SQL
- SQLParam2CSV
- CSV2SQL
- CSV2CSV
- File2File
- File2Email
- File2URL
- Email2Email

- Email2File
- URL2Email
- URL2File
- URL2URL

class glide.pipelines.**GliderTemplate**(*nodes*)
Bases: object

A template for a Glider. It will create a new pipeline with a copy of its templated nodes when __call__'d.

Parameters **nodes** – A top node potentially tied to other downstream nodes

nodes
A top node potentially tied to other downstream nodes

class glide.pipelines.**NodeTemplate**(*nodes*)
Bases: object

A set of nodes that can be used as a template

glide.pipelines.**basic_glider**(*extract=<class 'glide.core.PlaceholderNode'>*, *transform=<class 'glide.core.PlaceholderNode'>*, *load=<class 'glide.core.PlaceholderNode'>*)
Convenience function to produce a basic ETL template

Parameters

- **extract** (*type, optional*) – A Node class to use as the extractor
- **transform** (*type, optional*) – A Node class to use as the transformer
- **load** (*type, optional*) – A Node class to use as the loader

Returns

Return type A GliderTemplate that can be called to produce Gliders from the template.

13.8 glide.sql module

class glide.sql.**AssertSQL**(**args*, ***kwargs*)
Bases: *glide.sql.SQLNode*

run(*data*, *sql*, *conn*, *cursor=None*, *cursor_type=None*, *params=None*, *data_check=None*, ***kwargs*)
Run a SQL query to check data.

Parameters

- **data** – Data to pass through on success
- **sql** (*str*) – SQL query to run. Should return a single row with a "assert" column to indicate success. Truthy values for "assert" will be considered successful, unless data_check is passed in which case it will be compared for equality to the result of that callable.
- **conn** – SQL connection object
- **cursor** (*optional*) – SQL connection cursor object
- **cursor_type** (*optional*) – SQL connection cursor type when creating a cursor is necessary

- **params** (*tuple or dict, optional*) – A tuple or dict of params to pass to the execute method
- **data_check** (*callable, optional*) – A callable that will be passed the node and data as arguments and is expected to return a value to be compared to the SQL result.
- ****kwargs** – Keyword arguments pushed to the execute method

class glide.sql.**BaseSQLNode**(**args*, ***kwargs*)

Bases: *glide.flow.SkipFalseNode*

Base class for SQL-based nodes, checks for valid connection types on init

allowed_conn_types

A list or tuple of connection types that are allowed

Type list or tuple

allowed_conn_types = None

begin()

check_conn(*conn*)

Check the database connection

commit(*obj*)

Commit any currently active transactions

create_like(*conn*, *cursor*, *table*, *like_table*, *drop=False*)

Create a table like another table, optionally trying to drop *table* first

drop_table(*conn*, *cursor*, *table*)

Drop tables all day long

execute(*conn*, *cursor*, *sql*, *params=None*, ***kwargs*)

Executes the sql statement and returns an object that can fetch results

Parameters

- **conn** – A SQL database connection object
- **cursor** – A SQL database cursor
- **sql** (*str*) – A sql query to execute
- **params** (*tuple, optional*) – A tuple of params to pass to the execute method of the conn or cursor
- ****kwargs** – kwargs passed through to execute()

Returns cursor object that has executed but not fetched a query.

Return type cursor

executemany(*conn*, *cursor*, *sql*, *rows*)

Bulk executes the sql statement and returns an object that can fetch results

Parameters

- **conn** – A SQL database connection object
- **cursor** – A SQL database cursor
- **sql** (*str*) – A sql query to execute
- **rows** – Rows of data to bulk execute

Returns cursor object that has executed but not fetched a query.

Return type cursor

get_bulk_statement(*conn*, *stmt_type*, *table*, *rows*, *odku=False*)
Get a bulk execution SQL statement

Parameters

- **conn** – A SQL database connection object
- **stmt_type** (*str*) – Type of SQL statement to use (REPLACE, INSERT, etc.)
- **table** (*str*) – name of a SQL table
- **rows** – An iterable of dict rows. The first row is used to determine column names.
- **odku** (*bool or list, optional*) – If true, add ON DUPLICATE KEY UPDATE clause for all columns. If a list then only add it for the specified columns. **Note:** Backend support for this varies.

Returns

Return type A SQL bulk load query of the given stmt_type

get_sql_executor(*conn*, *cursor_type=None*)
Get the object that can execute queries

rename_tables(*conn*, *cursor*, *renames*)
Execute one or more table renames

rollback(*obj*)
Rollback any currently active transactions

transaction(*conn*, *cursor=None*)
Start a transaction. If conn is a SQLAlchemy conn return a reference to the transaction object, otherwise just return the conn which should have commit/rollback methods.

class glide.sql.**SQLCursorPushMixin**
Bases: object

Shared logic for SQL cursor-based nodes

do_push(*cursor*, *chunksize=None*)
Fetch data and push to the next node, obeying chunksize if passed

Parameters

- **cursor** – A cursor-like object with fetchmany and fetchall methods
- **chunksize** (*int, optional*) – If truthy the data will be fetched and pushed in chunks

class glide.sql.**SQLExecute**(**args*, ***kwargs*)
Bases: *glide.sql.SQLNode*

run(*sql*, *conn*, *cursor=None*, *cursor_type=None*, *params=None*, *commit=True*, *rollback=False*, *dry_run=False*, ***kwargs*)
Perform a generic SQL query execution and push the cursor/execute response.

Parameters

- **sql** (*str*) – SQL query to run
- **conn** – SQL connection object
- **cursor** (*optional*) – SQL connection cursor object

- **cursor_type** (*optional*) – SQL connection cursor type when creating a cursor is necessary
- **params** (*tuple or dict, optional*) – A tuple or dict of params to pass to the execute method
- **commit** (*bool, optional*) – If true try to commit the transaction. If your connection autocommits this will have no effect. If this is a SQLAlchemy connection and you are in a transaction, it will try to get a reference to the current transaction and call commit on that.
- **rollback** (*bool, optional*) – If true try to rollback the transaction on exceptions. Behavior may vary by backend DB library if you are not currently in a transaction.
- ****kwargs** – Keyword arguments pushed to the execute method

class glide.sql.**SQLFetch**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*, *glide.sql.SQLCursorPushMixin*

run(*cursor*, *chunksize=None*)
Fetch data from the cursor and push the result.

Parameters

- **cursor** – A cursor-like object that can fetch results
- **chunksize** (*int, optional*) – Fetch and push data in chunks of this size

class glide.sql.**SQLNode**(**args*, ***kwargs*)
Bases: *glide.sql.BaseSQLNode*, *glide.sql.SQLCursorPushMixin*

A generic SQL node that will behave differently based on the connection type

allowed_conn_types = [<class 'object'>]

check_conn(*conn*)
Make sure the object is a valid SQL connection

class glide.sql.**SQLTransaction**(**args*, ***kwargs*)
Bases: *glide.sql.SQLNode*

run(*data*, *conn*, *cursor=None*)
Begin a SQL transaction on the connection

Parameters

- **data** – Data being passed through the pipeline
- **conn** – Database connection to start the transaction on
- **cursor** (*optional*) – SQL connection cursor object

13.9 glide.sql_utils module

SQL utilities

class glide.sql_utils.**SQLiteTemporaryTable**(**args*, ***kwargs*)
Bases: pandas.io.sql.SQLiteTable

Override the default Pandas SQLite table creation to make it a temp table

class glide.sql_utils.**TemporaryTable**(*name*, *pandas_sql_engine*, *frame=None*, *index=True*, *if_exists='fail'*, *prefix='pandas'*, *index_label=None*, *schema=None*, *keys=None*, *dtype=None*)

Bases: pandas.io.sql.SQLTable

Override the default Pandas table creation to make it a temp table

glide.sql_utils.**add_table_suffix**(*table*, *suffix*)

Helper to deal with backticks when adding table suffix

glide.sql_utils.**build_table_select**(*table*, *where=None*, *limit=None*)

Simple helper to build a SQL query to select from a table

glide.sql_utils.**get_bulk_insert**(*table_name*, *column_names*, ***kwargs*)

Get a bulk insert statement

glide.sql_utils.**get_bulk_insert_ignore**(*table_name*, *column_names*, ***kwargs*)

Get a bulk insert ignore statement

glide.sql_utils.**get_bulk_replace**(*table_name*, *column_names*, ***kwargs*)

Get a bulk replace statement

glide.sql_utils.**get_bulk_statement**(*stmt_type*, *table_name*, *column_names*, *dicts=True*, *value_string='%s'*, *odku=False*)

Get a SQL statement suitable for use with bulk execute functions

Parameters

- **stmt_type** (*str*) – One of REPLACE, INSERT, or INSERT IGNORE. **Note:** Backend support for this varies.
- **table_name** (*str*) – Name of SQL table to use in statement
- **column_names** (*list*) – A list of column names to load
- **dicts** (*bool, optional*) – If true, assume the data will be a list of dict rows
- **value_string** (*str, optional*) – The parameter replacement string used by the underyling DB API
- **odku** (*bool or list, optional*) – If true, add ON DUPLICATE KEY UPDATE clause for all columns. If a list then only add it for the specified columns. **Note:** Backend support for this varies.

Returns **sql** – The sql query string to use with bulk execute functions

Return type str

glide.sql_utils.**get_temp_table**(*conn*, *data*, *create=False*, ***kwargs*)

Reuse Pandas logic for creating a temp table. The definition will be formed based on the first row of data passed

glide.sql_utils.**get_temp_table_name**()

Create a unique temp table name

glide.sql_utils.**is_sqlalchemy_conn**(*conn*)

Check if conn is a sqlalchemy connection

glide.sql_utils.**is_sqlalchemy_transaction**(*o*)

Check if an object is a sqlalchemy transaction

13.10 glide.transform module

A home for common transform nodes

Nodes:

- Func
- Map
- Sort
- Transpose
- DictKeyTransform
- HashKey
- JSONDumps
- JSONLoads
- EmailMessageTransform

class glide.transform.**DictKeyTransform**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

run(*data*, *drop=None*, ***transforms*)

Rename/replace keys in an iterable of dicts

Parameters

- **data** – Data to process. Expected to be a list/iterable of dict rows.
- **drop** (*list, optional*) – A list of keys to drop after transformations are complete.
- ****transforms** – key->value pairs used to populate columns of each dict row. If the value is a callable it is expected to take the row as input and return the value to fill in for the key.

class glide.transform.**EmailMessageTransform**(*name*, *_log=False*, *_debug=False*, ***default_context*)

Bases: *glide.core.Node*

Update EmailMessage objects

run(*msg*, *frm=None*, *to=None*, *subject=None*, *body=None*, *html=None*, *attachments=None*)

Update the EmailMessage with the given arguments

Parameters

- **msg** (*EmailMessage*) – EmailMessage object to update
- **frm** (*str, optional*) – Update from address
- **to** (*str, optional*) – Update to address(es)
- **subject** (*str, optional*) – Update email subject
- **body** (*str, optional*) – Update email body
- **html** (*str, optional*) – Update email html
- **attachments** (*list, optional*) – Replace the email attachments with these

class glide.transform.**Func**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Call func with data and push the result

run(*data*, *func*)
Call func with data and push the result

Parameters

- **data** – Data to process
- **func** (*callable*) – Function to pass data to

class glide.transform.**HashKey**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

run(*data*, *columns=None*, *hash_func=<built-in function openssl_md5>*, *hash_dest='id'*, *encoding='utf8'*)
Create a unique hash key from the specified columns and place it in each row.

Parameters

- **data** – An iterable of dict-like rows
- **columns** (*list, optional*) – A list of columns to incorporate into the key. If None, the keys of the first row will be used. If the first row is not an OrderedDict, the keys will be sorted before use.
- **hash_func** (*callable, optional*) – A callable from the hashlib module
- **hash_dest** (*str, optional*) – Column name to put the calculated key
- **encoding** (*str, optional*) – How to encode the values before hashing

class glide.transform.**JSONDumps**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Call json.dumps on the data

run(*data*)
Call json.dumps on the data and push

class glide.transform.**JSONLoads**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Call json.loads on the data

run(*data*)
Call json.loads on the data and push

class glide.transform.**Map**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Call the built-in map() function with func and data

run(*data*, *func*, *as_list=False*)
Call the built-in map() function with func and data

Parameters

- **data** – Data to process
- **func** (*callable*) – Function to pass to map()
- **as_list** (*bool, optional*) – If True, read the map() result into a list before pushing

class glide.transform.**Sort**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Sort data before pushing

run(*data*, *key=None*, *reverse=False*, *inplace=False*)
Sort data before pushing

Parameters

- **data** – The data to sort
- **key** (*callable, optional*) – Passed to the underlying sort methods
- **reverse** (*bool, optional*) – Passed to the underlying sort methods
- **inplace** (*bool, optional*) – If True, try to use list.sort(), otherwise use sorted()

class glide.transform.**Transpose**(*name*, *_log=False*, *_debug=False*, ***default_context*)
Bases: *glide.core.Node*

Transpose tabular data using zip

run(*data*)
Transpose tabular data using zip

13.11 glide.utils module

Common utilities

class glide.utils.**DateTimeWindowAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)
Bases: argparse.Action

An argparse Action for handling datetime window CLI args

class glide.utils.**DateWindowAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)
Bases: argparse.Action

An argparse Action for handling date window CLI args

glide.utils.**cancel_asyncio_tasks**(*tasks*, *loop*, *cancel_timeout=None*)
Cancel a set of asyncio tasks

Parameters

- **tasks** (*iterable*) – An iterable of asyncio tasks to cancel
- **loop** – asyncio Event Loop
- **cancel_timeout** (*int or float, optional*) – A timeout to use when waiting for tasks to finish cancelling

glide.utils.**closer**(*x*)
Helper to call close on x

glide.utils.**date_from_str**(*s*)

glide.utils.**date_window_cli**()
An argparse parent CLI that adds date window support

glide.utils.**datetime_cmp**(*d1*, *d2*)

glide.utils.**datetime_window_cli**()
An argparse parent CLI that adds datetime window support

glide.utils.**dbg**(*msg*, ***kwargs*)
Call tlbx dbg with glide logger

glide.utils.**dbgsql**(*msg*, ***kwargs*)
Call tlbx dbgsql with glide logger

glide.utils.**divide_data**(*data*, *n*)
Divide data into n chunks, with special handling for pandas objects

glide.utils.**error**(*msg*, ***kwargs*)
Call tlbx error with glide logger

glide.utils.**excel_file_type**(*f*)
Best guess at Excel file type from name

glide.utils.**find_class_in_dict**(*cls*, *d*, *include=None*, *exclude=None*)
Get a list of keys that are an instance of a class in a dict

glide.utils.**flatten**(*l*)
Flatten a list of iterables

glide.utils.**get_class_list_docstring**(*heading*, *classes*)
Helper to generate a part of a module docstring from a list of classes

glide.utils.**get_date_windows**(*start_date*, *end_date*, *reverse=False*)

glide.utils.**get_datetime_windows**(*start_date*, *end_date*, *window_size_hours=None*, *num_windows=None*, *reverse=False*, *add_second=True*)
Produce a list of start/end date tuples

Parameters

- **start_date** (*date, datetime, or str*) – The absolute start date of the range
- **end_date** (*date, datetime, or str*) – The absolute end date of the range
- **window_size_hours** (*float, optional*) – The size of the windows in hours. May be a float to represent partial hours.
- **num_windows** (*int, optional*) – The number of windows to split the date range into. One of num_windows or window_size_hours must be specified.
- **reverse** (*bool, optional*) – If true return the windows in reverse order
- **add_second** (*bool, optional*) – If true, offset the start of each window to be one second past the end date of the previous window.

Returns **dt_windows** – A list of tuples of start / end datetime pairs

Return type list

glide.utils.**get_file_handle**(**args*, ***kwargs*)
Context manager pass through to open_filepath_or_buffer. This will automatically close the file if and only if it was opened here. If file handles are passed in it is assumed the caller will manage them.

glide.utils.**get_or_create_event_loop**()
Get an existing asyncio Event Loop or create one if necessary

glide.utils.**info**(*msg*, ***kwargs*)
Call tlbx info with glide logger

glide.utils.**is_file_obj**(*o*)
Test if an object is a file object

glide.utils.**is_function**(*f*)
Test if f is a function

glide.utils.**is_pandas**(*o*)
Test if an object is a Pandas object

glide.utils.**iterize**(*o*)
Automatically wrap certain objects that you would not normally process item by item.

TODO: this function should probaly be improved/generalized.

glide.utils.**join**(*tables*, *on=None*, *how='left'*, *rsuffixes=None*)
Join a list of iterables or DataFrames

glide.utils.**listify**(*o*)
Ensure an object is a list by wrapping if necessary

glide.utils.**load_ini_config**(*filename*, *key=None*)
Load a config from an ini file, optionally extracting a key

glide.utils.**load_json_config**(*filename*, *key=None*)
Load a config from a json file, optionally extracting a key

glide.utils.**load_yaml_config**(*filename*, *key=None*)
Load a config from a yaml file, optionally extracting a key

glide.utils.**nchunks**(*a*, *n*)
Divide iterable a into n chunks

glide.utils.**not_none**(**args*)

glide.utils.**open_filepath_or_buffer**(*f*, *open_flags='r'*, *compression=None*, *is_text=True*)
Use pandas IO functions to return a handle from a filepath or buffer.

Parameters

- **f** (*str or buffer*) – filepath or buffer to open
- **open_flags** (*str, optional*) – mode to open file
- **compression** (*str, optional*) – compression arg passed to pandas functions
- **is_text** (*bool*) – Whether file/buffer is in text format, Passed through to pandas helpers.

Returns

- **f** (*file-like*) – A file-like object
- **handles** (*list of file-like*) – A list of file-like objects opened. Seems mostly relevant for zipped archives.
- **close** (*bool*) – A flag indicating whether the caller should close the file object when done

glide.utils.**read_excel**(*f*, ***kwargs*)
Read data from an Excel file using pyexcel

Parameters

- **f** (*str or buffer*) – Excel file to read from
- ****kwargs** – Keyword arguments passed to pyexcel

glide.utils.**save_excel**(*f*, *data*, ***kwargs*)
Write data to an Excel file using pyexcel

Note: If f is a file that ends in .xls, pyexcel_xls will be used, otherwise it defaults to pyexcel_xlsx.

Parameters

- **f** (*str or buffer*) – Excel file to write to
- **data** (*dict*) – Data to write to the file. This is expected to be a dict of {sheet_name: sheet_data} format.
- ****kwargs** – Keyword arguments passed to pyexcel's save_data

glide.utils.**size**(*o*, *default=None*)
Helper to return the len() of an object if it is available

glide.utils.**split_count_helper**(*data*, *split_count*)
Helper to override the split count if data len is shorter

glide.utils.**to_date**(*d*)

glide.utils.**to_datetime**(*d*)

glide.utils.**warn**(*msg*, ***kwargs*)
Call tlbx warn with glide logger

glide.utils.**window**(*seq*, *size=2*)
Returns a sliding window over data from the iterable

13.12 glide.version module

Package version

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W