

---

# **Intake Documentation**

*Release 0.2.9+12.g5a70b79.dirty*

**Anaconda**

**Nov 14, 2018**



---

## Documentation Contents

---

<b>1</b>	<b>Data User</b>	<b>3</b>
<b>2</b>	<b>Data Provider</b>	<b>5</b>
<b>3</b>	<b>IT</b>	<b>7</b>
<b>4</b>	<b>Developer</b>	<b>9</b>
<b>5</b>	<b>First steps</b>	<b>11</b>
<b>6</b>	<b>Indices and tables</b>	<b>57</b>



*Taking the pain out of data access and distribution*

Intake is a lightweight package for finding, investigating, loading and disseminating data. It will appeal to different groups for some of the reasons below, but is useful for all and acts as a common platform that everyone can use to smooth the progression of data from developers and providers to users.



# CHAPTER 1

---

## Data User

---

- Intake loads the data for a range of formats and types (see *Plugin Directory*) into containers you already use, like Pandas dataframes, Python lists, NumPy arrays, and more
- Intake loads, and gets out of your way
- GUI, search and introspect data-sets in *Catalogs*: quickly find what you need to do your work
- Install data-sets and automatically get requirements
- Leverage cloud resources and distributed computing.





## CHAPTER 2

---

### Data Provider

---

- Simple spec to define data sources
- Single point-of truth, no more copy&paste
- Distribute data using packages, shared files or a server
- Update definitions in-place
- Parametrise user options
- Make use of additional functionality like filename parsing and caching.



- Create catalogs out of established departmental practices
- Provide data access credentials via Intake parameters
- Use server-client architecture as gatekeeper:
  - add authentication methods
  - add monitoring point; track the data-sets being accessed.
- Hook Intake into proprietary data access systems.



## CHAPTER 4

---

Developer

---

- Turn boilerplate code into a reusable *Driver*
- Pluggable architecture of Intake allows for many points to add and improve
- Open, simple code-base, come and get involved on [github!](#)



For a brief demonstration and tutorial, which you can execute locally, go to [Quickstart](#). For a general description of all of the components of Intake and how they fit together, go to [Overview](#). Finally, for some notebooks using Intake and articles about Intake, go to [Examples](#). These and other documentation pages will make reference to concepts that are defined in the [Glossary](#).

## 5.1 Quickstart

This guide will show you how to get started using Intake to read data, and give you a flavour of how Intake feels to the [Data User](#). It assumes you are working in either a conda or a virtualenv/pip environment. For notebooks with executable code, see the [Examples](#). This walk-through can be run from a notebook or interactive python session.

### 5.1.1 Installation

If you are using [Anaconda](#) or [Miniconda](#), install Intake with the following commands:

```
conda install -c intake intake
```

If you are using [virtualenv/pip](#), run the following command:

```
pip install intake
```

## 5.1.2 Creating Sample Data

Let's begin by creating a sample data set and catalog. At the command line, run the `intake example` command. This will create an example data *Catalog* and two CSV data files. These files contains some basic facts about the 50 US states, and the catalog includes a specification of how to load them.

## 5.1.3 Loading a Data Source

*Data sources* can be created directly with the `open_*` () functions in the `intake` module. To read our example data:

```
>>> import intake
>>> ds = intake.open_csv('states_*.csv')
>>> print(ds)
<intake.source.csv.CSVSource object at 0x1163882e8>
```

Each open function has different arguments, specific for the data format or service being used.

## 5.1.4 Reading Data

Intake reads data into memory using *containers* you are already familiar with:

- Tables: Pandas DataFrames
- Multidimensional arrays: NumPy arrays
- Semistructred data: Python lists of objects (usually dictionaries)

To find out what kind of container a data source will produce, inspect the `container` attribute:

```
>>> ds.container
'dataframe'
```

The result will be `dataframe`, `ndarray`, or `python`. (New container types will be added in the future.)

For data that fits in memory, you can ask Intake to load it directly:

```
>>> df = ds.read()
>>> df.head()
   state      slug code      nickname  ...
0  Alabama  alabama  AL    Yellowhammer State
1  Alaska   alaska   AK    The Last Frontier
2  Arizona  arizona  AZ    The Grand Canyon State
3  Arkansas arkansas AR    The Natural State
4  California  california  CA    Golden State
```

Many data sources will also have quick-look plotting available. The attribute `.plot` will list a number of built-in plotting methods, such as `.scatter()`, see [Plotting](#).

Intake data sources can have *partitions*. A partition refers to a contiguous chunk of data that can be loaded independent of any other partition. The partitioning scheme is entirely up to the plugin author. In the case of the CSV plugin, each `.csv` file is a partition.

To read data from a data source one chunk at a time, the `read_chunked()` method returns an iterator:



```
>>> for chunk in ds.read_chunked(): print('Chunk: %d' % len(chunk))
...
Chunk: 24
Chunk: 26
```

## 5.1.5 Working with Dask

Working with large datasets is much easier with a parallel, out-of-core computing library like [Dask](#). Intake can create Dask containers (like `dask.dataframe`) from data sources that will load their data only when required:

```
>>> ddf = ds.to_dask()
>>> ddf
Dask DataFrame Structure:
      admission_date admission_number capital_city capital_url  code_
↪ constitution_url facebook_url landscape_background_url map_image_url nickname_
↪ population population_rank skyline_background_url slug state state_flag_url_
↪ state_seal_url twitter_url website
npartitions=2
↪ object          object          int64          object          object object
↪ object          object          object          object          object object
↪ int64          object          object object          object          object
↪ object object
↪ ...          ...          ...          ...          ...          ...
↪ ...          ...          ...          ...          ...          ...
↪ ...          ...          ...          ...          ...          ...
↪ ...          ...          ...          ...          ...          ...
↪ ...          ...          ...          ...          ...          ...
↪ ...          ...          ...          ...          ...          ...
Dask Name: from-delayed, 4 tasks
```

The Dask containers will be partitioned in the same way as the Intake data source, allowing different chunks to be processed in parallel. Please read the [Dask documentation](#) to understand the differences when working with Dask collections (Bag, Array or Data-frames).

## 5.1.6 Opening a Catalog

It is often useful to move the descriptions of data sources out of your code and into a specification file that can be reused and shared with other projects and people. Intake calls this a “*Catalog*”, which contains a list of named entries describing how to load data sources. The `intake example` command, above, created a catalog file with the following *YAML*-syntax content:

```
sources:
  states
    description: US state information from [CivilServices](https://civil.services/)
    driver: csv
    args:
      urlpath: '{{ CATALOG_DIR }}/states_*.csv'
    metadata:
      origin_url: 'https://github.com/CivilServiceUSA/us-states/blob/v1.0.0/data/
↪states.csv'
```

To load a catalog from a catalog file:

```
>>> cat = intake.open_catalog('us_states.yml')
>>> list(cat)
['states']
```

This catalog contains one data source, called `states`. It can be accessed by attribute:

```
>>> cat.states.to_dask()[['state', 'slug']].head()
   state      slug
0  Alabama  alabama
1   Alaska   alaska
2  Arizona   arizona
3  Arkansas  arkansas
4 California california
```

Placing data source specifications into a catalog like this enables declaring data sets in a single canonical place, and not having to use boilerplate code in each notebook/script that makes use of the data. The catalogs can also reference one-another, be stored remotely, and include extra metadata such as a set of named quick-look plots that are appropriate for the particular data source.

Note that, if you are *creating* such catalogs, you may well start by trying the `open_csv` command, above, and then use `print(ds.yaml())`. If you do this now, you will see that the output is very similar to the catalog file we have provided.

### 5.1.7 Installing Data Source Packages with Conda

Intake makes it possible to create *conda packages* that install data sources into a global catalog. For example, we can install a data package containing the same data we have been working with:

```
conda install -c intake data-us-states
```

*Conda* installs the catalog file in this package to `$CONDA_PREFIX/share/intake/us_states.yml`. Now, when we import `intake`, we will see the data from this package appear as part of a global catalog called `intake.cat`. In this particular case we use `Dask` to do the reading (which can handle larger-than-memory data and parallel processing), but `read()` would work also:

```
>>> import intake
>>> intake.cat.states.to_dask()[['state', 'slug']].head()
   state      slug
0  Alabama  alabama
1   Alaska   alaska
2  Arizona   arizona
3  Arkansas  arkansas
4 California california
```

The global catalog is a union of all catalogs installed in the `conda/virtualenv` environment and also any catalogs installed in user-specific locations.

#### Adding Data Source Packages using the Intake path

Intake checks the Intake config file for `catalog_path` or the environment variable `"INTAKE_PATH"` for a colon separated list of paths (semicolon on windows) to search for catalog files. When you import `intake` we will see all entries from all of the catalogues referenced as part of a global catalog called `intake.cat`.

## 5.1.8 Using the GUI

A graphical data browser is available in the Jupyter notebook environment. It will show the contents of any installed catalogs, plus allows for selecting local and remote catalogs, to browse and select entries from these. See [GUI](#).

## 5.2 Overview

### 5.2.1 Introduction

This page describes the technical design of Intake, with brief details of the aims of the project and components of the library

### 5.2.2 Why Intake?

Intake solves a related set of problems:

- Python API standards for loading data (such as DB-API 2.0) are optimized for transactional databases and query results that are processed one row at a time.
- Libraries that do load data in bulk tend to each have their own API for doing so, which adds friction when switching data formats.
- Loading data into a distributed data structure (like those found in Dask and Spark) often require writing a separate loader.
- Abstractions often focus on just one data model (tabular, n-dimensional array, or semi-structured), when many projects need to work with multiple kinds of data.

Intake has the explicit goal of **not** defining a computational expression system. Intake plugins load the data into containers (e.g., arrays or data-frames) that provide their data processing features. As a result, it is very easy to make a new Intake plugin with a relatively small amount of Python.

### 5.2.3 Structure

Intake is a Python library for accessing data in a simple and uniform way. It consists of three parts:

1. A lightweight plugin system for adding data loader *drivers* for new file formats and servers (like databases, REST endpoints or other cataloging services)
2. A cataloging system for specifying these sources in simple *YAML* syntax, or with plugins that read source specs from some external data service
3. A server-client architecture that can share data catalog metadata over the network, or even stream the data directly to clients if needed

Intake supports loading data into standard Python containers. The list can be easily extended, but the currently supported list is:

- Pandas Dataframes - tabular data
- NumPy Arrays - tensor data
- Python lists of dictionaries - semi-structured data

Additionally, Intake can load data into distributed data structures. Currently it supports Dask, a flexible parallel computing library with distributed containers like `dask.dataframe`, `dask.array`, and `dask.bag`. In the future, other distributed computing systems could use Intake to create similar data structures.

## 5.2.4 Concepts

Intake is built out of four core concepts:

- Data Source classes: the “driver” plugins that each implement loading of some specific type of data into python, with plugin-specific arguments.
- Data Source: An object that represents a reference to a data source. Data source objects have methods for loading the data into standard containers, like Pandas DataFrames, but do not load any data until specifically requested.
- Catalog: A collection of catalog entries, each of which defined a Data Source. Catalog objects can be created from local YAML definitions, by connecting to remote servers, or by some driver that knows how to query an external data service.
- Catalog Entry: A named data source. The catalog entry includes metadata about the source, as well as the name of the driver and arguments. Arguments can be parameterized, allowing one entry to return different subsets of data depending on the user request.

The business of a plugin is to go from some data format (bunch of files or some remote service) to a “*Container*” of the data (e.g., data-frame), a thing on which you can perform further analysis. Drivers can be used directly by the user, or indirectly through data catalogs. Data sources can be pickled, sent over the network to other hosts, and reopened (assuming the remote system has access to the required files or servers).

See also the *Glossary*.

## 5.2.5 Future Directions

Ongoing work for enhancements, as well as requests for plugins, etc., can be found at the [issue tracker](#), but see also *Roadmap* for general mid- and long-term goals.

## 5.3 Examples

Here we list links to notebooks and other code demonstrating the use of Intake in various scenarios. The first section is of general interest to various users, and the sections that follow tend to be more specific about particular features and workflows.

Many of the entries here include a link to Binder, which a service that lets you execute code live in a notebook environment. This is a great way to experience using Intake. It can take a while, sometimes, for Binder to come up, please have patience.

See also the [example data](#) page, containing data-sets which can be built and installed as conda packages.

### 5.3.1 General

- Basic Data scientist workflow: using Intake [\[Static\]](#) [\[Executable\]](#).
- Workflow for creating catalogs: a Data Engineer’s approach to Intake [\[Static\]](#) [\[Executable\]](#)

### 5.3.2 Developer

Tutorials delving deeper into the Internals of Intake, for those who wish to contribute

- How you would go about writing a new plugin [\[Static\]](#) [\[Executable\]](#)

### 5.3.3 Features

More specific examples of Intake functionality

- Caching:
  - Using automatically cached of data-files [Static] [Executable]
  - Earth science demonstration of cached dataset [Static] [Executable]
- File-name pattern parsing:
  - Satellite imagery, science workflow [Static] [Executable]
  - How to set up pattern parsing [Static] [Executable]

### 5.3.4 Blogs

These are Intake-related articles that may be of interest.

- [Taking the Pain out of Data Access](#)
- [Caching Data on First Read Makes Future Analysis Faster](#)
- [Parsing Data from Filenames and Paths](#)

## 5.4 GUI

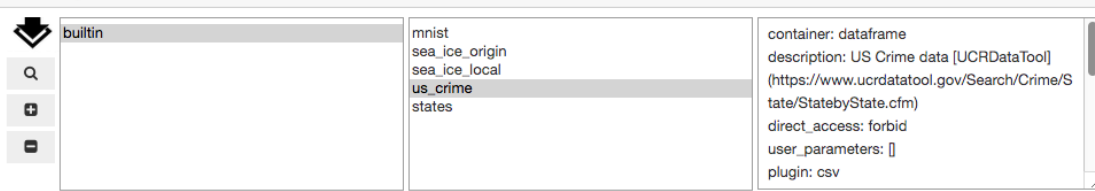
### 5.4.1 Using the Data Browser

**Note:** the data browser requires `ipywidgets` to be available in the current environment

The Intake top-level singleton `intake.gui` gives access to a graphical data browser within the Jupyter notebook. To expose it, simply enter it into a code cell (Jupyter automatically display the last object in a code cell).

```
In [1]: import intake
```

```
In [2]: intake.gui
```



```
In [3]: intake.gui.item
```

```
Out[3]: <Catalog Entry: us_crime>
```

```
In [4]: intake.gui.item.read()
```

```
Out[4]:
```

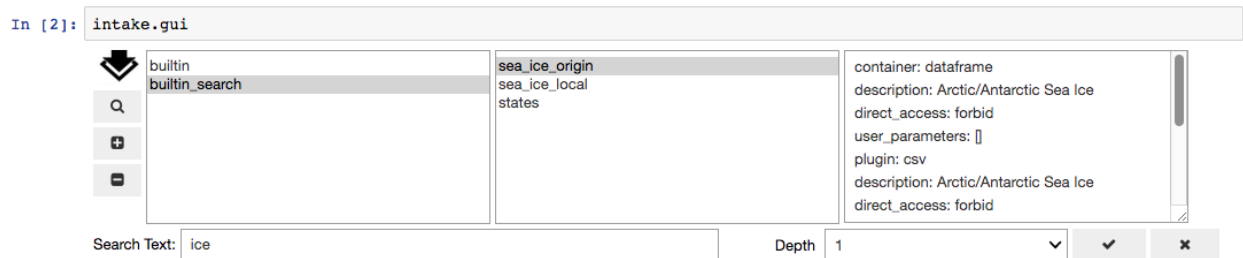
	Year	Population	Violent crime total	Murder and nonnegligent Manslaughter	Legacy rape /1	Revised rape /2	Robbery	Aggravated assault	Property crime total	Burglary	...	Violent Crime rate	Murder and nonnegligent manslaughter rate	Legacy rape rate /1	Revised rape rate /2	Robber
0	1960	179323175	288460	9110	17190	NaN	107840	154320	3095700	912100	...	160.9	5.1	9.6	NaN	6i
1	1961	182992000	289290	8740	17220	NaN	106670	156760	3198600	949600	...	158.1	4.8	9.4	NaN	6i

New instances of the GUI are also available by instantiating `intake.DataBrowser`, where you can specify a set of catalogs to initially include.

The GUI contains:

- a list of catalogs. The “builtin” catalog, displayed by default, includes data-sets installed in the system, the same as `intake.cat`.
- a list of entries within the currently-selected catalog. Entries marked with " ->" are themselves catalogs, clicking them will load the catalog and update the left-hand listing. Clicking any other entry will display some basic information in the right-hand text box.
- The following controls:
  - Search: opens a sub-panel for finding entries in the currently-selected catalog (and it’s sub-catalogs)
  - Add: opens a sub-panel for adding catalogs to the interface, by either browsing for a local YAML file or by entering a URL for a catalog, which can be a remote file or Intake server
  - Remove: delete the currently-selected catalog from the list

### Search



The sub-panel opened by the Search button allows for the entry of free-form text. Upon execution of the search with the check button, the currently-selected catalog will be searched. Entries will be considered to match if any of the entered words is found in the description of the entry (and this is case-insensitive). If any entries match, a new entry will be made in the catalog list, with the suffix “\_search”.

Some catalogs can contain nested sub-catalogs. The Depth selector allows the search to be limited to the stated number of nesting levels. This may be necessary, since, in theory, catalogs can contain circular references, and therefore allow for infinite recursion.

### Add Catalogs

The Add button exposes a sub-panel with two main ways to add catalogs to the interface

In [2]: intake.gui

The screenshot shows the intake GUI interface. At the top, there are two catalog panels. The left panel shows the 'builtin' catalog with a search icon and a plus icon. The right panel shows the 'mnist' catalog with a list of datasets: 'sea\_ice\_origin', 'sea\_ice\_local', 'us\_crime', and 'states'. To the right of these is a metadata panel for 'mnist' with fields: 'container: numpy', 'description: MNIST digit images and labels', 'direct\_access: forbid', 'user\_parameters: [{"name": "train\_or\_test", "description": "Dataset to choose", "type": "str", "default": "train", "allowed": ["train", "test"]}]', and 'plugin:'. Below these is a file selector window. It has two tabs: 'Local' and 'Remote'. The 'Local' tab is active, showing a directory listing for '/Users/mdurant/temp/'. The files listed are: '.ipynb\_checkpoints/', 'ae-training/', 'another.zarr/', 'cat.yaml', 'dask-tutorial-scipy-2018/', 'dask-worker-space/', 'gcsfs-feedstock/', 'landsat.yaml', 'pyskycalc/', 'rust/', 'temp.zarr/', and 'us\_states.yml'. The 'us\_states.yml' file is selected and highlighted. There are 'x' and '✓' icons next to the file list.

A *file selector*. Use the folder icon to open this below the main GUI. You can navigate around the filesystem, and select the catalog file you need. Use the tick icon to accept the selection, or the cross button to close the selector without accepting the selection. If accepted, the catalog will appear in the listing, unless parsing of the file fails:

In [2]: intake.gui

The screenshot shows the intake GUI interface. The 'cat' catalog is selected in the left panel. The right panel shows a list of catalogs: 'tables0 ->', 'tables1 ->', 'tables2 ->', 'tables3 ->', 'tables4 ->', 'tables5', and 'numpy'. The 'numpy' catalog is selected. To the right is a metadata panel for 'numpy' with fields: 'container: ndarray', 'description:', 'direct\_access: forbid', 'user\_parameters: []', 'plugin: numpy', and 'description:'. Below the catalog panels is a URL editor window. It has two tabs: 'Local' and 'Remote'. The 'Remote' tab is active, showing a text input field with the URL 'gcs://mdtemp/pangeo.yaml' and '✓' and 'x' buttons.

A *URL editor*. Any URL is valid here, including cloud locations, "gcs://bucket/...", and intake servers, "intake://server:port". Without a protocol specifier, this can be a local path. Again, use the check button to accept the value, and attempt to load the file into the interface's catalogs list.

In [2]: intake.gui

The screenshot shows the intake GUI interface. The 'cat' catalog is selected in the left panel. The right panel shows a list of catalogs: 'tables0 ->', 'tables1 ->', 'tables2 ->', 'tables3 ->', 'tables4 ->', 'tables5', and 'numpy'. The 'numpy' catalog is selected. To the right is a metadata panel for 'numpy' with fields: 'container: ndarray', 'description:', 'direct\_access: forbid', 'user\_parameters: []', 'plugin: numpy', and 'description:'. Below the catalog panels is a URL editor window. It has two tabs: 'Local' and 'Remote'. The 'Remote' tab is active, showing a text input field with the URL 'gcs://mdtemp/pangeo.yaml' and '✓' and 'x' buttons.

Finally, you can add catalogs to the interface in code, using the `.add_cat()` method, which can take filenames, remote URLs or an existing `Catalog` instance.

## 5.4.2 Using the Selection

Once catalogs are loaded and the correct entry has been identified and selected, the item in question will be available as the `.item` attribute (`intake.gui.item`), which has informational methods available and can be opened as a data source, as with any catalog entry:

```
In [ ]: intake.gui.item.describe_open()
Out  : {'plugin': 'csv',
       'description': 'Arctic/Antarctic Sea Ice',
       'direct_access': 'forbid',
       'metadata': {},
       'args': {'urlpath': 'https://timeseries.weebly.com/uploads/2/1/0/8/21086414/
↪sea_ice.csv',
               'metadata': {}}}
```

```
In [ ]: s = intake.gui.item() # may specify parameters here
       s.read()
Out  : < some data >
```

```
In [ ]: intake.gui.item.plot() # or skip data source step
Out  : < graphics >
```

## 5.5 API

Auto-generated reference

### 5.5.1 End User

These are reference class and function definitions likely to be useful to everyone.

<code>intake.open_catalog(uri)</code>	Create a Catalog object
<code>intake.registry</code>	
<code>intake.open_</code>	
<code>intake.source.csv.CSVSource(urlpath[, ...])</code>	Read CSV files into dataframes
<code>intake.source.textfiles. TextFilesSource(urlpath)</code>	Read textfiles as sequence of lines
<code>intake.source.npy.NPySource(path[, dtype, ...])</code>	Read numpy binary files into an array
<code>intake.catalog.local. YAMLFileCatalog(path, ...)</code>	Catalog as described by a single YAML file
<code>intake.catalog.local. YAMLFilesCatalog(path)</code>	Catalog as described by a multiple YAML files
<code>intake.gui.DataBrowser([cats])</code>	Intake data set browser

`intake.open_catalog(uri=None, **kwargs)`  
Create a Catalog object

Can load YAML catalog files, connect to an intake server, or create any arbitrary Catalog subclass instance. In the general case, the user should supply `driver=` with a value from the plugins registry which has a container type of catalog. File locations can generally be remote, if specifying a URL protocol.

The default behaviour if not specifying the driver is as follows:



- if `uri` is a single string ending in “yml” or “yaml”, open it as a catalog file
- if `uri` is a list of strings, a string containing a glob character (“\*”) or a string not ending in “y(a)ml”, open as a set of catalog files. In the latter case, assume it is a directory.
- if `uri` begins with protocol “intake:”, connect to a remote Intake server
- otherwise, create a base Catalog object without entries.

### Parameters

**uri:** `str` Designator for the location of the catalog.

**kwargs:** passed to subclass instance, see documentation of the individual catalog classes.

### `intake.registry`

Mapping from plugin names to the DataSource classes that implement them. These are the names that should appear in the `driver:` key of each source definition in a catalog. See [Plugin Directory](#) for more details.

### `intake.open_`

Set of functions, one for each plugin, for direct opening of a data source. The names are derived from the names of the plugins in the registry at import time.

**class** `intake.source.csv.CSVSource` (*urlpath*, *csv\_kwargs=None*, *metadata=None*, *storage\_options=None*, *path\_as\_pattern=True*)

Read CSV files into dataframes

Prototype of sources reading dataframe data

**\_\_init\_\_** (*urlpath*, *csv\_kwargs=None*, *metadata=None*, *storage\_options=None*, *path\_as\_pattern=True*)

### Parameters

**urlpath** [`str` or iterable, location of data] May be a local path, or remote path if including a protocol specifier such as ‘s3://’. May include glob wildcards or format pattern strings. Some examples:

- `{{ CATALOG_DIR }}data/precipitation.csv`
- `s3://data/*.csv`
- `s3://data/precipitation_{state}_{zip}.csv`
- `s3://data/{year}/{month}/{day}/precipitation.csv`
- `{{ CATALOG_DIR }}data/precipitation_{date:%Y-%m-%d}.csv`

**csv\_kwargs** [`dict`] Any further arguments to pass to Dask’s `read_csv` (such as block size) or to the `CSV parser` in pandas (such as which columns to use, encoding, data-types)

**storage\_options** [`dict`] Any parameters that need to be passed to the remote data backend, such as credentials.

**path\_as\_pattern** [`bool` or `str`, optional] Whether to treat the path as a pattern (ie. `data_{field}.csv`) and create new columns in the output corresponding to pattern fields. If `str`, is treated as pattern to match on. Default is `True`.

### `discover()`

Open resource and populate the source attributes.

### `read()`

Load entire dataset into a container and return it

**read\_partition** (*i*)

Return a (offset\_tuple, container) corresponding to i-th partition.

Offset tuple is of same length as shape.

By default, assumes *i* should be an integer between zero and npartitions; override for more complex indexing schemes.

**to\_dask** ()

Return a dask container for this data source

**class** intake.source.textfiles.**TextFilesSource** (*urlpath*, *metadata=None*, *storage\_options=None*)

Read textfiles as sequence of lines

Prototype of sources reading sequential data.

Takes a set of files, and returns an iterator over the text in each of them. The files can be local or remote. Extra parameters for encoding, etc., go into *storage\_options*.

**\_\_init\_\_** (*urlpath*, *metadata=None*, *storage\_options=None*)

#### Parameters

**urlpath** [str or list(str)] Target files. Can be a glob-path (with “\*”) and include protocol specified (e.g., “s3://”). Can also be a list of absolute paths.

**storage\_options: dict** Options to pass to the file reader backend, including text-specific encoding arguments, and parameters specific to the remote file-system driver, if using.

**discover** ()

Open resource and populate the source attributes.

**read** ()

Load entire dataset into a container and return it

**read\_partition** (*i*)

Return a (offset\_tuple, container) corresponding to i-th partition.

Offset tuple is of same length as shape.

By default, assumes *i* should be an integer between zero and npartitions; override for more complex indexing schemes.

**to\_dask** ()

Return a dask container for this data source

**class** intake.source.npy.**NPYSource** (*path*, *dtype=None*, *shape=None*, *chunks=None*, *storage\_options=None*, *metadata=None*)

Read numpy binary files into an array

Prototype source showing example of working with arrays

Each file becomes one or more partitions, but partitioning within a file is only along the largest dimension, to ensure contiguous data.

**\_\_init\_\_** (*path*, *dtype=None*, *shape=None*, *chunks=None*, *storage\_options=None*, *metadata=None*)

The parameters *dtype* and *shape* will be determined from the first file, if not given.

#### Parameters

**path: str of list of str** Location of data file(s), possibly including glob and protocol information

**dtype: str dtype spec** In known, the dtype (e.g., “int64” or “f4”).

**shape: tuple of int** If known, the length of each axis

**chunks: int** Size of chunks within a file along biggest dimension - need not be an exact factor of the length of that dimension

**storage\_options: dict** Passed to file-system backend.

**discover()**

Open resource and populate the source attributes.

**read()**

Load entire dataset into a container and return it

**read\_partition(i)**

Return a (offset\_tuple, container) corresponding to i-th partition.

Offset tuple is of same length as shape.

By default, assumes i should be an integer between zero and npartitions; override for more complex indexing schemes.

**to\_dask()**

Return a dask container for this data source

**class** intake.catalog.local.**YAMLFileCatalog**(path, \*\*kwargs)

Catalog as described by a single YAML file

**\_\_init\_\_**(path, \*\*kwargs)

#### Parameters

**path: str** Location of the file to parse (can be remote)

**reload()**

Reload catalog if sufficient time has passed

**walk**(sofar=None, prefix=None, depth=2)

Get all entries in this catalog and sub-catalogs

#### Parameters

**sofar: dict or None** Within recursion, use this dict for output

**prefix: list of str or None** Names of levels already visited

**depth: int** Number of levels to descend; needed to truncate circular references and for cleaner output

#### Returns

**Dict where the keys are the entry names in dotted syntax, and the values are entry instances.**

**class** intake.catalog.local.**YAMLFilesCatalog**(path, flatten=True, \*\*kwargs)

Catalog as described by a multiple YAML files

**\_\_init\_\_**(path, flatten=True, \*\*kwargs)

#### Parameters

**path: str** Location of the files to parse (can be remote), including possible glob (\*) character(s). Can also be list of paths, without glob characters.

**flatten: bool (True)** Whether to list all entries in the cats at the top level (True) or create sub-cats from each file (False).

**reload()**

Reload catalog if sufficient time has passed

**walk** (*sofar=None, prefix=None, depth=2*)

Get all entries in this catalog and sub-catalogs

#### Parameters

**sofar: dict or None** Within recursion, use this dict for output

**prefix: list of str or None** Names of levels already visited

**depth: int** Number of levels to descend; needed to truncate circular references and for cleaner output

#### Returns

**Dict where the keys are the entry names in dotted syntax, and the values are entry instances.**

**class** `intake.gui.DataBrowser` (*cats=None*)

Intake data set browser

### Examples

Display GUI in a notebook thus:

```
>>> gui = intake.gui.widgets.DataBrowser()
>>> gui.widget
```

Optionally you can specify some Catalog instances to begin with. If none are given, will use the default one, `intake.cat`.

After interacting, the value of `gui.item` will be the entry selected.

**add\_cat** (*ev=None*)

Add catalog instance to the browser

Also called, without a catalog instance, when the (+) button is pressed, which will attempt to read a catalog and, if successful, add it to the browser.

**file\_chosen** (*fn*)

Callback from AddCat panel

## 5.5.2 Base Classes

This is a reference API class listing, useful mainly for developers.

---

<code>intake.source.base.DataSource([metadata])</code>	An object which can produce data
<code>intake.source.base.PatternMixin</code>	Helper class to provide file-name parsing abilities to a driver class
<code>intake.source.base.AliasSource(target[, ...])</code>	Refer to another named source, unmodified
<code>intake.container.base.RemoteSource(url, ...)</code>	Base class for all DataSources living on an Intake server
<code>intake.catalog.Catalog(*args, **kwargs)</code>	Manages a hierarchy of data sources as a collective unit.

---

Continued on next page

Table 2 – continued from previous page

<code>intake.catalog.entry. CatalogEntry([getenv, ...])</code>	A single item appearing in a catalog
<code>intake.catalog.local. UserParameter(name, ...)</code>	A user-settable item that is passed to a DataSource upon instantiation.
<code>intake.auth.base.BaseAuth(*args)</code>	Base class for authorization
<code>intake.source.cache.BaseCache(driver, spec)</code>	Provides utilities for managing cached data files.
<code>intake.source.base.Schema(**kwargs)</code>	Holds details of data description for any type of data-source

**class** `intake.source.base.DataSource` (*metadata=None*)

An object which can produce data

This is the base class for all Intake plugins, including catalogs and remote (server) data objects. To produce a new plugin commonly involves subclassing this definition and overriding some or all of the methods.

This class is not useful in itself, most methods raise `NotImplemented`.

**plot**

Accessor for `HVPlot` methods. See *Plotting* for more details.

**close ()**

Close open resources corresponding to this data source.

**discover ()**

Open resource and populate the source attributes.

**hvplot**

Returns a `hvPlot` object to provide a high-level plotting API.

**plot**

Returns a `hvPlot` object to provide a high-level plotting API.

To display in a notebook, be sure to run `intake.output_notebook ()` first.

**plots**

List custom associated quick-plots

**read ()**

Load entire dataset into a container and return it

**read\_chunked ()**

Return iterator over container fragments of data source

**read\_partition (i)**

Return a (offset\_tuple, container) corresponding to i-th partition.

Offset tuple is of same length as shape.

By default, assumes `i` should be an integer between zero and `npartitions`; override for more complex indexing schemes.

**to\_dask ()**

Return a `dask` container for this data source

**yaml** (*with\_plugin=False*)

Return YAML representation of this data-source

The output may be roughly appropriate for inclusion in a YAML catalog. This is a best-effort implementation

**Parameters**

**with\_plugin: bool** If True, create a “plugins” section, for cases where this source is created with a plugin not expected to be in the global Intake registry.

**class** `intake.catalog.Catalog` (*\*args, \*\*kwargs*)

Manages a hierarchy of data sources as a collective unit.

A catalog is a set of available data sources for an individual entity (remote server, local file, or a local directory of files). This can be expanded to include a collection of subcatalogs, which are then managed as a single unit.

A catalog is created with a single URI or a collection of URIs. A URI can either be a URL or a file path.

Each catalog in the hierarchy is responsible for caching the most recent refresh time to prevent overeager queries.

#### Attributes

**metadata** [dict] Arbitrary information to carry along with the data source specs.

**discover** ()

Open resource and populate the source attributes.

**force\_reload** ()

Imperative reload data now

**reload** ()

Reload catalog if sufficient time has passed

**walk** (*sofar=None, prefix=None, depth=2*)

Get all entries in this catalog and sub-catalogs

#### Parameters

**sofar: dict or None** Within recursion, use this dict for output

**prefix: list of str or None** Names of levels already visited

**depth: int** Number of levels to descend; needed to truncate circular references and for cleaner output

#### Returns

**Dict where the keys are the entry names in dotted syntax, and the values are entry instances.**

**class** `intake.catalog.entry.CatalogEntry` (*getenv=True, getshell=True*)

A single item appearing in a catalog

This is the base class, used by local entries (i.e., read from a YAML file) and by remote entries (read from a server).

**describe** ()

Get a dictionary of attributes of this entry.

Returns: dict with keys

**container** [str] kind of container used by this data source

**description** [str] Markdown-friendly description of data source

**direct\_access** [str] Mode of remote access: forbid, allow, force

**user\_parameters** [list[dict]] List of user parameters defined by this entry

**describe\_open** (*\*\*user\_parameters*)

Get a dictionary describing how to open this data source.

#### Parameters

**user\_parameters** [dict] Values for user-configurable parameters for this data source

**Returns:** dict with keys

**plugin** [str] Name of data plugin to use

**description** [str] Markdown-friendly description of data source

**direct\_access** [str] Mode of remote access: forbid, allow, force

**metadata** [dict] Dictionary of metadata defined in the catalog for this data source

**args: dict** Dictionary of keyword arguments for the plugin

**get** (\*\**user\_parameters*)

Open the data source.

Equivalent to calling the catalog entry like a function.

#### Parameters

**user\_parameters** [dict] Values for user-configurable parameters for this data source

**Returns:** DataSource

#### plots

List custom associated quick-plots

**class** `intake.container.base.RemoteSource` (*url, headers, name, parameters, metadata=None, \*\*kwargs*)

Base class for all DataSources living on an Intake server

**to\_dask** ()

Return a dask container for this data source

**class** `intake.catalog.local.UserParameter` (*name, description, type, default=None, min=None, max=None, allowed=None*)

A user-settable item that is passed to a DataSource upon instantiation.

For string parameters, default may include special functions `func(args)`, which *may* be expanded from environment variables or by executing a shell command.

#### Parameters

**name: str** the key that appears in the DataSource argument strings

**description: str** narrative text

**type: str** one of list“(COERSION\_RULES)”

**default: type value** same type as `type`. If a str, may include special functions `env`, `shell`, `client_env`, `client_shell`.

**min, max: type value** for validation of user input

**allowed: list of type** for validation of user input

**describe** ()

Information about this parameter

**expand\_defaults** (*client=False, getenv=True, getshell=True*)

Compile `env`, `client_env`, `shell` and `client_shell` commands

**validate** (*value*)

Does value meet parameter requirements?

**class** `intake.auth.base.BaseAuth(*args)`

Base class for authorization

Subclass this and override the methods to implement a new type of auth.

This basic class allows all access.

**allow\_access** (*header, source, catalog*)

Is the given HTTP header allowed to access given data source

**Parameters**

**header:** **dict** The HTTP header from the incoming request

**source:** **CatalogEntry** The data source the user wants to access.

**catalog:** **Catalog** The catalog object containing this data source.

**allow\_connect** (*header*)

Is the requests header given allowed to talk to the server

**Parameters**

**header:** **dict** The HTTP header from the incoming request

**get\_case\_insensitive** (*dictionary, key, default=None*)

Case-insensitive search of a dictionary for key.

Returns the value if key match is found, otherwise default.

**class** `intake.source.cache.BaseCache(driver, spec, catdir=None, cache_dir=None, storage_options={})`

Provides utilities for managing cached data files.

Providers of caching functionality should derive from this, and appear as entries in `registry`. The principle methods to override are `_make_files()` and `_load()` and `_from_metadata()`.

**clear\_all** ()

Clears all cache and metadata.

**clear\_cache** (*urlpath*)

Clears cache and metadata for a given urlpath.

**Parameters**

**urlpath:** **str, location of data** May be a local path, or remote path if including a protocol specifier such as `'s3://'`. May include glob wildcards.

**get\_metadata** (*urlpath*)

**Parameters**

**urlpath:** **str, location of data** May be a local path, or remote path if including a protocol specifier such as `'s3://'`. May include glob wildcards.

**Returns**

**Metadata (dict) about a given urlpath.**

**load** (*urlpath, output=None, \*\*kwargs*)

Downloads data from a given url, generates a hashed filename, logs metadata, and caches it locally.

**Parameters**

**urlpath:** **str, location of data** May be a local path, or remote path if including a protocol specifier such as `'s3://'`. May include glob wildcards.

**output:** **bool** Whether to show progress bars; turn off for testing



## Returns

**List of local `cache_paths` to be opened instead of the remote file(s). If caching is disable, the `urlpath` is returned.**

**class** `intake.source.base.AliasSource` (*target, mapping=None, metadata=None, \*\*kwargs*)

Refer to another named source, unmodified

The purpose of an Alias is to be able to refer to other source(s) in the same catalog, perhaps leaving the choice of which target to load up to the user. This source makes no sense outside of a catalog.

In this case, the output of the target source is not modified, but this class acts as a prototype ‘derived’ source for processing the output of some standard driver.

After initial discovery, the source’s container and other details will be updated from the target; initially, the AliasSource container is not any standard.

`__init__` (*target, mapping=None, metadata=None, \*\*kwargs*)

## Parameters

**target: str** Name of the source to load, must be a key in the same catalog

**mapping: dict or None** If given, use this to map the string passed as `target` to entries in the catalog

**metadata: dict or None** Extra metadata to associate

**kwargs: passed on to the target**

**class** `intake.source.base.PatternMixin`

Helper class to provide file-name parsing abilities to a driver class

**class** `intake.source.base.Schema` (*\*\*kwargs*)

Holds details of data description for any type of data-source

This should always be pickleable, so that it can be sent from a server to a client, and contain all information needed to recreate a RemoteSource on the client.

## 5.6 Catalogs

Data catalogs provide an abstraction that allows you to externally define, and optionally share, descriptions of datasets, called *catalog entries*. A catalog entry for a dataset includes information like:

- The name of the Intake driver that can load the data
- Arguments to the `__init__()` method of the driver
- Metadata provided by the catalog author (such as field descriptions and types, or data provenance)

In addition, Intake allows datasets to be *parameterized* in the catalog. This is most commonly used to allow the user to filter down datasets at load time, rather than having to bring everything into memory first. The data parameters are defined by the catalog author, then templated into the arguments for `__init__()` to modify the data being loaded. This approach is less flexible for the end user than something like the [Blaze expression system](#), but also significantly reduces the implementation burden for driver authors.

### 5.6.1 YAML Format

Intake catalogs are typically described with YAML files. Here is an example:

```
metadata:
  version: 1
sources:
  example:
    description: test
    driver: random
    args: {}

  entry1_full:
    description: entry1 full
    metadata:
      foo: 'bar'
      bar: [1, 2, 3]
    driver: csv
    args: # passed to the open() method
      urlpath: '{{ CATALOG_DIR }}/entry1_*.csv'

  entry1_part:
    description: entry1 part
    parameters: # User defined parameters
      part:
        description: part of filename
        type: str
        default: "1"
        allowed: ["1", "2"]
    driver: csv
    args:
      urlpath: '{{ CATALOG_DIR }}/entry1_{{ part }}.csv'
```

## Templating

Intake catalog files support Jinja2 templating for driver arguments. Any occurrence of a substring like `{{field}}` will be replaced by the value of the user parameters with that same name. Some additional values are always available:

- `CATALOG_DIR`: The full path to the directory containing the YAML catalog file. This is especially useful for constructing paths relative to the catalog directory to locate data files and custom drivers.

## Metadata

Arbitrary extra descriptive information can go into the metadata section. Some fields will be claimed for internal use and some fields may be restricted to local reading; but for now the only field that is expected is `version`, which will be updated when a breaking change is made to the file format. Any catalog will have `.metadata` and `.version` attributes available.

## Extra drivers

The `driver`: entry of a data source specification can be a driver name, as has been shown in the examples so far. It can also be an absolute class path to use for the data source, in which case there will be no ambiguity about how to load the data. That is the preferred way to be explicit, when the driver name alone is not enough (see *Driver Selection*, below). However, it is also possible to specify extra modules and directories to scan for plugins, as an alternative method for finding driver classes.

In addition to using drivers already installed in the Python environment with `conda` or `pip` (see *Driver Discovery*), a catalog can also use additional drivers from arbitrary locations listed in the YAML file:

```

plugins:
  source:
    - module: intake.catalog.tests.example1_source
    - dir: '{{ CATALOG_DIR }}/example_plugin_dir'
sources:
  ...

```

The following import methods are allow:

- `- module: my.module.path`: The Python module to import and search for driver classes. This uses the standard notation of the Python `import` command and will search the `PYTHONPATH` in the same way.
- `- dir: /my/module/directory`: All of the `*.py` files in this directory will be executed, and any driver classes found will be added to the catalog’s plugin registry. It is common for the directory of Python files to be stored relative to the catalog file itself, so using the `CATALOG_DIR` variable will allow that relative path to be specified.

Each of the above methods can be used multiple times, and in combination, to load as many extra drivers as are needed. Most drivers should be installed as Python packages (enabling autodiscovery), but sometimes catalog-specific drivers may be needed to perform specific data transformations that are not broadly applicable enough to warrant creating a dedicated package. In those cases, the above options allow the drivers to be bundled with the catalog instead.

## Sources

The majority of a catalog file is composed of data sources, which are named data sets that can be loaded for the user. Catalog authors describe the contents of data set, how to load it, and optionally offer some customization of the returned data. Each data source has several attributes:

- `name`: The canonical name of the source. Best practice is to compose source names from valid Python identifiers. This allows Intake to support things like tab completion of data source names on catalog objects. For example, `monthly_downloads` is a good source name.
- `description`: Human readable description of the source. To help catalog browsing tools, the description should be Markdown.
- `driver`: Name of the Intake *Driver* to use with this source. Must either already be installed in the current Python environment (i.e. with `conda` or `pip`) or loaded in the `plugin` section of the file.
- `args`: Keyword arguments to the `open()` method of the driver. Arguments may use template expansion.
- `metadata`: Any metadata keys that should be attached to the data source when opened. These will be supplemented by additional metadata provided by the driver. Catalog authors can use whatever key names they would like, with the exception that keys starting with a leading underscore are reserved for future internal use by Intake.
- `direct_access`: Control whether the data is directly accessed by the client, or proxied through a catalog server. See *Remote Catalogs* for more details.
- `parameters`: A dictionary of data source parameters. See below for more details.

Parameters allow the user to customize the data returned by a data source. Most often, parameters are used to filter or reduce the data in specific ways defined by the catalog author. The parameters defined for a given data source are available for use in template strings, which can be used to alter the arguments provided to the driver. For example, a data source might accept a `postal_code` argument which is used to alter a database query, or select a particular group within a file. Users set parameters with keyword arguments to the `get()` method on the catalog object.

## Driver Selection

In some cases, it may be possible that multiple backends are capable of loading from the same data format or service. Sometimes, this may mean two drivers with unique names, or a single driver with a parameter to choose between the different backends.

However, it is possible that multiple drivers for reading a particular type of data also share the same driver name: for example, both the `intake-iris` and the `intake-xarray` packages contain drivers with the name `"netcdf"`, which are capable of reading the same files, but with different backends. Here we will describe the various possibilities of coping with this situation. Intake's plugin system makes it easy to encode such choices.

It may be acceptable to use any driver which claims to handle that data type, or to give the option of which driver to use to the user, or it may be necessary to specify which precise driver(s) are appropriate for that particular data. Intake allows all of these possibilities, even if the backend drivers require extra arguments.

Specifying a single driver explicitly, rather than using a generic name, would look like this:

```
sources:
  example:
    description: test
    driver: package.module.PluginClass
    args: {}
```

It is also possible to describe a list of drivers with the same syntax. The first one found will be the one used. Note that the class imports will only happen at data source instantiation.

```
sources:
  example:
    description: test
    driver:
      - package.module.PluginClass
      - another_package.PluginClass2
    args: {}
```

These alternative plugins can also be given data-source specific names, allowing the user to choose at load time with `driver=` as a parameter. Additional arguments may also be required for each option (which, as usual, may include user parameters); however, the same global arguments will be passed to all of the drivers listed.

```
sources:
  example:
    description: test
    driver:
      first:
        class: package.module.PluginClass
        args:
          specific_thing: 9
      second:
        class: another_package.PluginClass2
    args: {}
```

## Parameter Definition

To enable users to discover parameters on data sources, and to allow UIs to generate interfaces automatically, parameters have the following attributes in the catalog.

- `description`: Human-readable Markdown description of what the parameter means.

- `type`: The type of the parameter. Currently, this may be `bool`, `str`, `int`, `float`, `list[str]`, `list[int]`, `list[float]`, `datetime`.
- `default`: The default value for this parameter. Every parameter must have a default to ensure a catalog user can quickly see some sample data.
- `allowed` (optional): A list of allowed values for this parameter
- `min` (optional): Minimum value (inclusive) for the parameter
- `max` (optional): Maximum value (inclusive) for the parameter

Note both `allowed` and `min/max` should not be set for the same parameter.

Also the `datetime` type accepts multiple values:

- a Python `datetime` object
- an ISO8601 timestamp string
- an integer representing a Unix timestamp
- `now`, a string representing the current timestamp
- `today`, a string representing today at midnight UTC

The `default` field allows for special syntax to get information from the system. This is particularly useful for user credentials, which may be defined by environment variables or fetched by running some external command. The special syntax are:

- `env (USER)`: look in the environment for the named variable; in the example, this will be the username.
- `client_env (USER)`: exactly the same, except that when using a client-server topology, the value will come from the environment of the client.
- `shell (get_login thisuser -t)`: execute the command, and use the output as the value. The output will be trimmed of any trailing whitespace.
- `client_shell (get_login thisuser -t)`: exactly the same, except that when using a client-server topology, the value will come from the system of the client.

Since it may not be desirable to have the access of a catalog get information from the system, the keywords `getenv` and `getshell` (passed to `Catalog`) allow these mechanisms to be turned off, in which case the value of the default will still appear as the original template string (and so the user should override with a value they have obtained elsewhere). Note that in the case of a remote catalog, the client cannot see the values that will be evaluated on the server side, the evaluation only happens if the user did not override the value when accessing the data.

## Caching Source Files Locally

To enable caching on the first read of remote data source files, `cache` specifications have the following attributes in the catalog.

- `argkey`: Of the keys in the `args` section in this same data source, which contains the URL(s) of the data to be cached.
- `regex`: A regular expression to match against the URL path, where the matching portion will be replaced by a path in the local cache directory.
- `type`: One of the keys in the cache registry [*intake.source.cache.registry*], referring to an implementation of caching behaviour. The default is “file” for the caching of one or more specific remote files.

Example:

```
test_cache:
  description: cache a csv file from the local filesystem
  driver: csv
  cache:
    - argkey: urlpath
      regex: '{{ CATALOG_DIR }}/cache_data'
      type: file
  args:
    urlpath: '{{ CATALOG_DIR }}/cache_data/states.csv'
```

The `cache_dir` defaults to `~/.intake/cache`, and can be specified in the intake configuration file or `INTAKE_CACHE_DIR` environment variable, or at runtime using the `"cache_dir"` key of the configuration. The special value `"catdir"` implies that cached files will appear in the same directory as the catalog file in which the data source is defined, within a directory named `"intake_cache"`. These will not appear in the cache usage reported by the CLI.

Caching can be disabled at runtime for all sources regardless of the catalog specification:

```
from intake.config import conf

conf['cache_disabled'] = True
```

By default, progress bars are shown during downloads if the package `tqdm` is available, but this can be disabled (e.g., for consoles that don't support complex text) with

```
conf['cache_download_progress'] = False
```

or, equivalently, the environment parameter `INTAKE_CACHE_PROGRESS`.

The “types” of caching that are supported are listed in `intake.source.cache.registry`, see the docstrings of each for specific parameters that should appear in the cache block.

## 5.6.2 Remote Access

(see also *Remote Data* for the implementation details)

Many drives support reading directly from remote data sources such as HTTP, S3 or GCS. In these cases, the path to read from is usually given with a protocol prefix such as `gcs://`. Additional dependencies will typically be required (`requests`, `s3fs`, `gcsfs`, etc.), any any data conda package should specify this. Further parameters may be necessary for communicating with the storage backend and, by convention, the driver should take a parameter `storage_options` containing arguments to pass to the backend.

As an example of using `storage_options`, the following two sources would allow for reading CSV data from S3 and GCS backends without authentication (anonymous access), respectively

```
sources:
  s3_csv:
    description: "Publicly accessible CSV data on S3; requires s3fs"
    args:
      urlpath: s3://bucket/path/*.csv
      storage_options:
        anon: true
  gcs_csv:
    description: "Publicly accessible CSV data on GCS; requires gcsfs"
    args:
      urlpath: gcs://bucket/path/*.csv
      storage_options:
        token: "anon"
```

### 5.6.3 Local Catalogs

A Catalog can be loaded from a YAML file on the local filesystem by creating a Catalog object:

```
from intake import load_catalog

cat = load_catalog('catalog.yaml')
```

Then sources can be listed:

```
list(cat)
```

and data sources are loaded via their name:

```
data = cat.entry_part1(part='1')
```

Intake also supports loading all of the files ending in `.yaml` and `.yml` in a directory, or by using an explicit glob-string. Note that the URL provided may refer to a remote storage systems by passing a protocol specifier such as `s3://`, `gcs://`:

```
cat = load_catalog('/research/my_project/catalog.d/')
```

Intake Catalog objects will automatically detect changes or new additions to catalog files and directories on disk. These changes will not affect already-opened data sources.

### 5.6.4 Remote Catalogs

Intake also includes a server which can share an Intake catalog over HTTP (or HTTPS with the help of a TLS-enabled reverse proxy). From the user perspective, remote catalogs function identically to local catalogs:

```
cat = open_catalog('intake://catalog1:5000')
list(cat)
```

The difference is that operations on the catalog translate to requests sent to the catalog server. Catalog servers provide access to data sources in one of two modes:

- **Direct access:** In this mode, the catalog server tells the client how to load the data, but the client uses its local drivers to make the connection. This requires the client has the required driver already installed *and* has direct access to the files or data servers that the driver will connect to.
- **Proxied access:** In this mode, the catalog server uses its local drivers to open the data source and stream the data over the network to the client. The client does not need *any* special drivers to read the data, and can read data from files and data servers that it cannot access, as long as the catalog server has the required access.

Whether a particular catalog entry supports direct or proxied access is determined by the `direct_access` option:

- `forbid` (default): Force all clients to proxy data through the catalog server
- `allow`: If the client has the required driver, access the source directly, otherwise proxy the data through the catalog server.
- `force`: Force all clients to access the data directly. If they do not have the required driver, an exception will be raised.

Note that when the client is loading a data source via direct access, the catalog server will need to send the driver arguments to the client. Do not include sensitive credentials in a data source that allows direct access.

## Client Authorization Plugins

Intake servers can check if clients are authorized to access the catalog as a whole, or individual catalog entries. Typically a matched pair of server-side plugin (called an “auth plugin”) and a client-side plugin (called a “client auth plugin) need to be enabled for authorization checks to work. This feature is still in early development, but see module `intake.auth.secret` for a demonstration pair of server and client classes implementation auth via a shared secret. See *Authorization Plugins*.

## 5.7 Command Line Tools

The package installs two executable commands: for starting the catalog server; and a client for accessing catalogs and manipulating the configuration.

### 5.7.1 Configuration

A file-based configuration service is available to Intake. This file is by default sought at the location `~/.intake/conf.yaml`, but either of the environment variables `INTAKE_CONF_DIR` or `INTAKE_CONF_FILE` can be used to specify another directory or file. If both are given, the latter takes priority.

At present, the configuration file might look as follows:

```
auth:
  class: "intake.auth.base.BaseAuth"
port: 5000
catalog_path:
  - /home/myusername/special_dir
```

These are the defaults, and any parameters not specified will take the values above

- the Intake Server will listen on port 5000 (this can be overridden on the command line, see below)
- and the auth system used will be the fully-qualified class given (which, for `BaseAuth`, always allows access). For further information on securing the Intake Server, see the *Authorization Plugins*.

See `intake.config.defaults` for a full list of keys and their default values.

### 5.7.2 Log Level

The logging level is configurable using Python’s built-in logging module.

The config option `'logging'` holds the current level for the intake logger, and can take values such as `'INFO'` or `'DEBUG'`. This can be set in the `conf.yaml` file of the config directory (e.g., `~/.intake/`), or overridden by the environment variable `INTAKE_LOG_LEVEL`.

Furthermore, the level and settings of the logger can be changed programmatically in code:

```
import logging
logger = logging.getLogger('intake')
logger.setLevel(logging.DEBUG)
logget.addHandler(..)
```



### 5.7.3 Intake Server

The server takes one or more catalog files as input and makes them available on port 5000 by default.

You can see the full description of the server command with:

```
>>> intake-server --help
usage: intake-server [-h] [-p PORT] [--sys-exit-on-sigterm] FILE [FILE ...]

Intake Catalog Server

positional arguments:
  FILE                  Name of catalog YAML file

optional arguments:
  -h, --help            show this help message and exit
  -p PORT, --port PORT  port number for server to listen on
  --sys-exit-on-sigterm internal flag used during unit testing to ensure
                        .coverage file is written
```

To start the server with a local catalog file, use the following:

```
>>> intake-server intake/catalog/tests/catalog1.yml
Creating catalog from:
  - intake/catalog/tests/catalog1.yml
catalog_args ['intake/catalog/tests/catalog1.yml']
Entries: entry1,entry1_part,use_example1
Listening on port 5000
```

You can use the catalog client (defined below) using:

```
$ intake list intake://localhost:5000
entry1
entry1_part
use_example1
```

### 5.7.4 Intake Client

While the Intake data sources will typically be accessed through the Python API, you can use the client to verify a catalog file.

Unlike the server command, the client has several subcommands to access a catalog. You can see the list of available subcommands with:

```
>>> intake --help
usage: intake {list,describe,exists,get,discover} ...
```

We go into further detail in the following sections.

#### List

This subcommand lists the names of all available catalog entries. This is useful since other subcommands require these names.

If you wish to see the details about each catalog entry, use the `--full` flag. This is equivalent to running the `intake describe` subcommand for all catalog entries.

```
>>> intake list --help
usage: intake list [-h] [--full] URI

positional arguments:
  URI          Catalog URI

optional arguments:
  -h, --help  show this help message and exit
  --full
```

```
>>> intake list intake/catalog/tests/catalog1.yml
entry1
entry1_part
use_example1
>>> intake list --full intake/catalog/tests/catalog1.yml
[entry1] container=dataframe
[entry1] description=entry1 full
[entry1] direct_access=forbid
[entry1] user_parameters=[]
[entry1_part] container=dataframe
[entry1_part] description=entry1 part
[entry1_part] direct_access=allow
[entry1_part] user_parameters=[{'default': '1', 'allowed': ['1', '2'], 'type': u'str',
↪ 'name': u'part', 'description': u'part of filename'}]
[use_example1] container=dataframe
[use_example1] description=example1 source plugin
[use_example1] direct_access=forbid
[use_example1] user_parameters=[]
```

## Describe

Given the name of a catalog entry, this subcommand lists the details of the respective catalog entry.

```
>>> intake describe --help
usage: intake describe [-h] URI NAME

positional arguments:
  URI          Catalog URI
  NAME        Catalog name

optional arguments:
  -h, --help  show this help message and exit
```

```
>>> intake describe intake/catalog/tests/catalog1.yml entry1
[entry1] container=dataframe
[entry1] description=entry1 full
[entry1] direct_access=forbid
[entry1] user_parameters=[]
```

## Discover

Given the name of a catalog entry, this subcommand returns a key-value description of the data source. The exact details are subject to change.

```
>>> intake discover --help
usage: intake discover [-h] URI NAME

positional arguments:
  URI          Catalog URI
  NAME        Catalog name

optional arguments:
  -h, --help  show this help message and exit
```

```
>>> intake discover intake/catalog/tests/catalog1.yml entry1
{'npartitions': 2, 'dtype': dtype([('name', 'O'), ('score', '<f8'), ('rank', '<i8')]),
 → 'shape': (None,), 'datashape':None, 'metadata': {'foo': 'bar', 'bar': [1, 2, 3]}}
```

## Exists

Given the name of a catalog entry, this subcommand returns whether or not the respective catalog entry is valid.

```
>>> intake exists --help
usage: intake exists [-h] URI NAME

positional arguments:
  URI          Catalog URI
  NAME        Catalog name

optional arguments:
  -h, --help  show this help message and exit
```

```
>>> intake exists intake/catalog/tests/catalog1.yml entry1
True
>>> intake exists intake/catalog/tests/catalog1.yml entry2
False
```

## Get

Given the name of a catalog entry, this subcommand outputs the entire data source to standard output.

```
>>> intake get --help
usage: intake get [-h] URI NAME

positional arguments:
  URI          Catalog URI
  NAME        Catalog name

optional arguments:
  -h, --help  show this help message and exit
```

```
>>> intake get intake/catalog/tests/catalog1.yml entry1
  name  score  rank
0  Alice1  100.5   1
1   Bob1   50.3    2
2  Charlie1 25.0    3
3    Eve1  25.0    3
```

(continues on next page)

(continued from previous page)

4	Alice2	100.5	1
5	Bob2	50.3	2
6	Charlie2	25.0	3
7	Eve2	25.0	3

## Config and Cache

CLI functions starting with `intake cache` and `intake config` are available to provide information about the system: the locations and value of configuration parameters, and the state of cached files.

## 5.8 Making Drivers

The goal of the Intake plugin system is to make it very simple to implement a *Driver* for a new data source, without any special knowledge of Dask or the Intake catalog system.

### 5.8.1 Assumptions

Although Intake is very flexible about data, there are some basic assumptions that a driver must satisfy.

#### Data Model

Intake currently supports 3 kinds of containers, represented the most common data models used in Python:

- dataframe
- ndarray
- python (list of Python objects, usually dictionaries)

Although a driver can load *any* type of data into any container, and new container types can be added to the list above, it is reasonable to expect that the number of container types remains small. Declaring a container type is only informational for the user when read locally, but streaming of data from a server requires that the container type be known to both server and client.

A given driver must only return one kind of container. If a file format (such as HDF5) could reasonably be interpreted as two different data models depending on usage (such as a dataframe or an ndarray), then two different drivers need to be created with different names. If a driver returns the `python` container, it should document what Python objects will appear in the list.

The source of data should be essentially permanent and immutable. That is, loading the data should not destroy or modify the data, nor should closing the data source destroy the data either. When a data source is serialized and sent to another host, it will need to be reopened at the destination, which may cause queries to be re-executed and files to be reopened. Data sources that treat readers as “consumers” and remove data once read will cause erratic behavior, so Intake is not suitable for accessing things like FIFO message queues.

#### Schema

The schema of a data source is a detailed description of the data, which can be known by loading only metadata or by loading only some small representative portion of the data. It is information to present to the user about the data that they are considering loading, and may be important in the case of server-client communication. In the latter context, the contents of the schema must be serializable by `msgpack` (i.e., numbers, strings, lists and dictionaries only).

There may be unknown parts of the schema before the whole data is read. drivers may require this unknown information in the `__init__()` method (or the catalog spec), or do some kind of partial data inspection to determine the schema; or more simply, may be given as unknown `None` values. Regardless of method used, the time spent figuring out the schema ahead of time should be short and not scale with the size of the data.

Typical fields in a schema dictionary are `npartitions`, `dtype`, `shape`, etc., which will be more appropriate for some drivers/data-types than others.

## Partitioning

Data sources are assumed to be *partitionable*. A data partition is a randomly accessible fragment of the data. In the case of sequential and data-frame sources, partitions are numbered, starting from zero, and correspond to contiguous chunks of data divided along the first dimension of the data structure. In general, any partitioning scheme is conceivable, such as a tuple-of-ints to index the chunks of a large numerical array.

Not all data sources can be partitioned. For example, file formats without sufficient indexing often can only be read from beginning to end. In these cases, the `DataSource` object should report that there is only 1 partition. However, it often makes sense for a data source to be able to represent a directory of files, in which case each file will correspond to one partition.

## Metadata

Once opened, a `DataSource` object can have arbitrary metadata associated with it. The metadata for a data source should be a dictionary that can be serialized as JSON. This metadata comes from the following sources:

1. A data catalog entry can associate fixed metadata with the data source. This is helpful for data formats that do not have any support for metadata within the file format.
2. The driver handling the data source may have some general metadata associated with the state of the system at the time of access, available even before loading any data-specific information.
2. A driver can add additional metadata when the schema is loaded for the data source. This allows metadata embedded in the data source to be exported.

From the user perspective, all of the metadata should be loaded once the data source has loaded the rest of the schema (after `discover()`, `read()`, `to_dask()`, etc have been called).

### 5.8.2 Subclassing `intake.source.base.DataSource`

Every Intake driver class should be a subclass of `intake.source.base.DataSource`. The class should have the following attributes to identify itself:

- `name`: The short name of the driver. This should be a valid python identifier. You should not include the word `intake` in the driver name.
- `version`: A version string for the driver. This may be reported to the user by tools based on Intake, but has no semantic importance.
- `container`: The container type of data sources created by this object, e.g., `dataframe`, `ndarray`, or `python`, one of the keys of `intake.container.container_map`. For simplicity, a driver may only return one typed of container. If a particular source of data could be used in multiple ways (such as HDF5 files interpreted as dataframes or as ndarrays), two drivers must be created. These two drivers can be part of the same Python package.
- `partition_access`: Do the data sources returned by this driver have multiple partitions? This may help tools in the future make more optimal decisions about how to present data. If in doubt (or the answer depends on `init` arguments), `True` will always result in correct behavior, even if the data source has only one partition.

The `__init__()` method should always accept a keyword argument `metadata`, a dictionary of metadata from the catalog to associate with the source. This dictionary must be serializable as JSON.

The base `DataSource` class has a small number of methods which should be overridden. Here is an example producing a data-frame:

```
class FooSource(intake.source.base.DataSource):
    container = 'dataframe'
    name = 'foo'
    version = '0.0.1'
    partition_access = True

    def __init__(self, a, b, metadata=None):
        # Do init here with a and b
        super(FooSource, self).__init__(
            metadata=metadata
        )

    def _get_schema(self):
        return intake.source.base.Schema(
            datashape=None,
            dtype={'x': "int64", 'y': "int64"},
            shape=(None, 2),
            npartitions=2,
            extra_metadata=dict(c=3, d=4)
        )

    def _get_partition(self, i):
        # Return the appropriate container of data here
        return pd.DataFrame({'x': [1, 2, 3], 'y': [10, 20, 30]})

    def read(self):
        self._load_metadata()
        return pd.concat([self.read_partition(i) for i in self.npartitions])

    def _close(self):
        # close any files, sockets, etc
        pass
```

Most of the work typically happens in the following methods:

- `__init__()`: Should be very lightweight and fast. No files or network resources should be opened, and no significant memory should be allocated yet. Data sources are often serialized immediately. The default implementation of the pickle protocol in the base class will record all the arguments to `__init__()` and recreate the object with those arguments when unpickled, assuming the class has no side effects.
- `_get_schema()`: May open files and network resources and return as much of the schema as possible in small amount of *approximately* constant time. The `npartitions` and `extra_metadata` attributes must be correct when `_get_schema` returns. Further keys such as `dtype`, `shape`, etc., should reflect the container type of the data-source, and can be `None` if not easily knowable, or include `None` for some elements. This method should call the `_get_cache` method, if caching on first time read is supported by the driver. For example:

```
urlpath, *_ = self._get_cache(self.urlpath)
```

Will return the location of the cached urlpath for the first matching cache specified in the catalog source.

- `_get_partition(self, i)`: Should return all of the data from partition id `i`, where `i` is typically an integer, but may be something more complex. The base class will automatically verify that `i` is in the range

[0, npartitions), so no range checking is required in the typical case.

- `_close(self)`: Close any network or file handles and deallocate any significant memory. Note that these resources may be need to be reopened/reallocated if a read is called again later.

The full set of user methods of interest are as follows:

- `discover(self)`: Read the source attributes, like `npartitions`, etc. As with `_get_schema()` above, this method is assumed to be fast, and make a best effort to set attributes. The output should be serializable, if the source is to be used on a server; the details contained will be used for creating a remote-source on the client.
- `read(self)`: Return all the data in memory in one in-memory container.
- `read_chunked(self)`: Return an iterator that returns contiguous chunks of the data. The chunking is generally assumed to be at the partition level, but could be finer grained if desired.
- `read_partition(self, i)`: Returns the data for a given partition id. It is assumed that reading a given partition does not require reading the data that precedes it. If `i` is out of range, an `IndexError` should be raised.
- `to_dask(self)`: Return a (lazy) Dask data structure corresponding to this data source. It should be assumed that the data can be read from the Dask workers, so the loads can be done in future tasks. For further information, see the [Dask documentation](#).
- `close(self)`: Close network or file handles and deallocate memory. If other methods are called after `close()`, the source is automatically reopened.

It is also important to note that source attributes should be set after `read()`, `read_chunked()`, `read_partition()` and `to_dask()`, even if `discover()` was not called by the user.

### 5.8.3 Driver Discovery

When Intake is imported, it will search the Python module path (by default includes `site-packages` and other directories in your `$PYTHONPATH`) for packages starting with `intake_` and discover `DataSource` subclasses inside those packages to register. drivers will be registered based on the `'name'` attribute of the object. By convention, drivers should have names that are lowercase, valid Python identifiers that do not contain the word `intake`.

After the discovery phase, Intake will automatically create `open_[driver_name]` convenience functions under the `intake` module namespace. Calling a function like `open_csv()` is equivalent to instantiating the corresponding data-source class.

To take advantage of driver discovery, give your installed package a name that starts with `intake_` and define your driver class(es) in the `__init__.py` of the package.

### 5.8.4 Remote Data

For drivers loading from files, the author should be aware that it is easy to implement loading from files stored in remote services. A simplistic case is demonstrated by the included CSV driver, which simply passes a URL to Dask, which in turn can interpret the URL as a remote data service, and use the `storage_options` as required (see the Dask documentation on [remote data](#)).

More advanced usage, where a Dask loader does not already exist, will likely rely on `dask.bytes.open_files`. Use this function to produce lazy `OpenFile` object for local or remote data, based on a URL, which will have a protocol designation and possibly contain glob `"*"` characters. Additional parameters may be passed to `open_files`, which should, by convention, be supplied by a driver argument named `storage_options` (a dictionary).

To use an `OpenFile` object, make it concrete by using a context:

```
# at setup, to discover the number of files/partitions
set_of_open_files = dask.bytes.open_files(urlpath, mode='rb', **storage_options)

# when actually loading data; here we loop over all files, but maybe we just do one_
↳partition
for an_open_file in set_of_open_files:
    # `with` causes the object to become concrete until the end of the block
    with an_open_file as f:
        # do things with f, which is a file-like object
        f.seek(); f.read()
```

The `textfiles` builtin drivers implements this mechanism, as an example.

## 5.8.5 Structured File Paths

The CSV driver sets up an example of how to gather data which is encoded in file paths like ('data\_{site}\_.csv') and return that data in the output. Other drivers could also follow the same structure where data is being loaded from a set of filenames. Typically this would apply to data-frame output. This is possible as long as the driver has access to each of the file paths at some point in `_get_schema`. Once the file paths are known, the driver developer can use the helper functions defined in `intake.source.utils` to get the values for each field in the pattern for each file in the list. These values should then be added to the data, a process which normally would happen within the `_get_schema` method.

The `PatternMixin` defines driver properties such as `urlpath`, `path_as_pattern`, and `pattern`. The implementation might look something like this:

```
from intake.source.utils import reverse_formats

class FooSource(intake.source.base.DataSource, intake.source.base.PatternMixin):
    def __init__(self, a, b, path_as_pattern, urlpath, metadata=None):
        # Do init here with a and b
        self.path_as_pattern = path_as_pattern
        self.urlpath = urlpath

        super(FooSource, self).__init__(
            container='dataframe',
            metadata=metadata
        )
    def _get_schema(self):
        # read in the data
        values_by_field = reverse_formats(self.pattern, file_paths)
        # add these fields and map values to the data
        return data
```

Since `dask` already has a specific method for including the file paths in the output dataframe, in the CSV driver we set `include_path_column=True`, to get a dataframe where one of the columns contains all the file paths. In this case, *add these fields and values to data* is a mapping between the categorical file paths column and the `values_by_field`.

In other drivers where each file is read in independently the driver developer can set the new fields on the data from each file before concatenating. This pattern looks more like:

```
from intake.source.utils import reverse_format

class FooSource(intake.source.base.DataSource):
    ...
```

(continues on next page)



(continued from previous page)

```

def _get_schema(self):
    # get list of file paths
    for path in file_paths:
        # read in the file
        values_by_field = reverse_format(self.pattern, path)
        # add these fields and values to the data
    # concatenate the datasets
    return data

```

To toggle on and off this path as pattern behavior, the CSV and intake-xarray drivers uses the bool `path_as_pattern` keyword argument.

## 5.9 Authorization Plugins

Authorization plugins are classes that can be used to customize access permissions to the Intake catalog server. The Intake server and client communicate over HTTP, so when security is a concern, the *most important* step to take is to put a TLS-enabled reverse proxy (like `nginx`) in front of the Intake server to encrypt all communication.

Whether or not the connection is encrypted, the Intake server by default allows all clients to list the full catalog, and open any of the entries. For many use cases, this is sufficient, but if the visibility of catalog entries needs to be limited based on some criteria, a server- (and/or client-) side authorization plugin can be used.

### 5.9.1 Server Side

An Intake server can have exactly one server side plugin enabled at startup. The plugin is activated using the Intake configuration file, which lists the class name and the keyword arguments it takes. For example, the “shared secret” plugin would be configured this way:

```

auth:
  class: intake.auth.secret.Secret
  kwargs:
    secret: A_SECRET_HASH

```

This plugin is very simplistic, and exists as a demonstration of how an auth plugin might function for more realistic scenarios.

For more information about configuring the Intake server, see [Configuration](#).

The server auth plugin has two methods. The `allow_connect()` method decides whether to allow a client to make any request to the server at all, and the `allow_access()` method decides whether the client is allowed to see a particular catalog entry in the listing and whether they are allowed to open that data source. Note that for catalog entries which allow direct access to the data (via network or shared filesystem), the Intake authorization plugins have no impact on the visibility of the underlying data, only the entries in the catalog.

The actual implementation of a plugin is very short. Here is a simplified version of the shared secret auth plugin:

```

class SecretAuth(BaseAuth):
    def __init__(self, secret, key='intake-secret'):
        self.secret = secret
        self.key = key

    def allow_connect(self, header):

```

(continues on next page)

(continued from previous page)

```

    try:
        return self.get_case_insensitive(header, self.key, '') \
            == self.secret
    except:
        return False

def allow_access(self, header, source, catalog):
    try:
        return self.get_case_insensitive(header, self.key, '') \
            == self.secret
    except:
        return False

```

The `header` argument is a dictionary of HTTP headers that were present in the client request. In this case, the plugin is looking for a special `intake-secret` header which contains the shared secret token. Because HTTP header names are not case sensitive, the `BaseAuth` class provides a helper method `get_case_insensitive()`, which will match dictionary keys in a case-insensitive way.

The `allow_access` method also takes two additional arguments. The `source` argument is the instance of `LocalCatalogEntry` for the data source being checked. Most commonly auth plugins will want to inspect the `_metadata` dictionary for information used to make the authorization decision. Note that it is entirely up to the plugin author to decide what sections they want to require in the metadata section. The `catalog` argument is the instance of `Catalog` that contains the catalog entry. Typically, plugins will want to use information from the `catalog.metadata` dictionary to control global defaults, although this is also up to the plugin.

## 5.9.2 Client Side

Although server side auth plugins can function entirely independently, some authorization schemes will require the client to add special HTTP headers for the server to look for. To facilitate this, the `Catalog` constructor accepts an optional `auth` parameter with an instance of a client auth plugin class.

The corresponding client plugin for the shared secret use case describe above looks like:

```

class SecretClientAuth(BaseClientAuth):
    def __init__(self, secret, key='intake-secret'):
        self.secret = secret
        self.key = key

    def get_headers(self):
        return {self.key: self.secret}

```

It defines a single method, `get_headers()`, which is called to get a dictionary of additional headers to add to the HTTP request to the catalog server. To use this plugin, we would do the following:

```

import intake
from intake.auth.secret import SecretClientAuth

auth = SecretClientAuth('A_SECRET_HASH')
cat = intake.Catalog('http://example.com:5000', auth=auth)

```

Now all requests made to the remote catalog will contain the `intake-secret` header.

## 5.10 Making Data Packages

Combined with the [Conda Package Manger](#), Intake makes it possible to create *data packages* which can be installed and upgraded just like software packages. This offers several advantages:

- Distributing datasets becomes as easy as installing software
- Data packages can be versioned, improving reproducibility in some cases
- Data packages can depend on the libraries required for reading
- Data packages can be self-describing using Intake catalog files
- Applications that need certain datasets can include data packages in their dependency list

In this tutorial, we give a walkthrough to enable you to distribute any dataset to others, so that they can access the data using Intake without worrying about where it resides or how it should be loaded.

### 5.10.1 Defining a Package

The steps involved in creating a data package are:

1. Identifying a dataset, which can be accessed via a URL or included directly as one or more files in the package.
2. Creating a package containing an intake catalog file, `meta.yaml` (description of the data, version, requirements, etc.) and a script to copy the data.
3. Building the package using the command `conda build`.
4. Uploading the package to a package repository such as [Anaconda Cloud](#) or your own private repository.

Data packages are standard conda packages that install an Intake catalog file into the user's conda environment (`$CONDA_PREFIX/share/intake`). A data package does not necessarily imply there are data files inside the package. A data package could describe remote data sources (such as files in S3) and take up very little space on disk.

These packages are considered `noarch` packages, so that one package can be installed on any platform, with any version of Python (or no Python at all). The easiest way to create such a package is using a [conda build](#) recipe.

Conda-build recipes are stored in a directory that contains a files like:

- `meta.yaml` - description of package metadata
- `build.sh` - script for building/installing package contents (on Linux/macOS)
- other files needed by the package (catalog files and data files for data packages)

An example that packages up data from a Github repository would look like this:

```
# meta.yaml
package:
  version: '1.0.0'
  name: 'data-us-states'

source:
  git_rev: v1.0.0
  git_url: https://github.com/CivilServiceUSA/us-states

build:
  number: 0
  noarch: generic
```

(continues on next page)

(continued from previous page)

```
requirements:
  run:
    - intake
  build: []

about:
  description: Data about US states from CivilServices (https://civil.services/)
  license: MIT
  license_family: MIT
  summary: Data about US states from CivilServices
```

The key parts of a data package recipe (different from typical conda recipes) is the `build` section:

```
build:
  number: 0
  noarch: generic
```

This will create a package that can be installed on any platform, regardless of the platform where the package is built. If you need to rebuild a package, the build number can be incremented to ensure users get the latest version when they conda update.

The corresponding `build.sh` file in the recipe looks like this:

```
#!/bin/bash

mkdir -p $PREFIX/share/intake/civilservices
cp $SRC_DIR/data/states.csv $PREFIX/share/intake/civilservices
cp $RECIPE_DIR/us_states.yaml $PREFIX/share/intake/
```

The `$SRC_DIR` variable refers to any source tree checked out (from Github or other service), and the `$RECIPE_DIR` refers to the directory where the `meta.yaml` is located.

Finishing out this example, the catalog file for this data source looks like this:

```
sources:
  states:
    description: US state information from [CivilServices](https://civil.services/)
    driver: csv
    args:
      urlpath: '{{ CATALOG_DIR }}/civilservices/states.csv'
    metadata:
      origin_url: 'https://github.com/CivilServiceUSA/us-states/blob/v1.0.0/data/
↪states.csv'
```

The `{{ CATALOG_DIR }}` Jinja2 variable is used to construct a path relative to where the catalog file was installed.

To build the package, you must have `conda-build` installed:

```
conda install conda-build
```

Building the package requires no special arguments:

```
conda build my_recipe_dir
```

Conda-build will display the path of the built package, which you will need to upload it.

If you want your data package to be publicly available on [Anaconda Cloud](#), you can install the `anaconda-client` utility:

```
conda install anaconda-client
```

Then you can register your Anaconda Cloud credentials and upload the package:

```
anaconda login
anaconda upload /Users/intake_user/anaconda/conda-bld/noarch/data-us-states-1.0.0-0.
↪tar.bz2
```

## 5.10.2 Best Practices

### Versioning

- Versions for data packages should be used to indicate changes in the data values or schema. This allows applications to easily pin to the specific data version they depend on.
- Package build numbers should be used to indicate changes in the packaging of the data (fixes to conda package metadata, like dependencies). If you need to change the data format (like CSV to Parquet), this can be indicated with a new build number, but only if the data contents and schema are identical even after the format change. (When in doubt, assign a new version number.)
- Putting data files into a package ensures reproducibility by allowing a version number to be associated with files on disk. This can consume quite a bit of disk space for the user, however. Conda does use hard-links when installing packages into an environment, so the disk space used by a data package will not multiply as it is added to more environments in the same Anaconda installation.

### Packaging

- Packages that refer to remote data sources (such as databases and REST APIs) need to think about authentication. Do not include authentication credentials inside a data package. They should be obtained from the environment.
- Data packages should depend on the Intake plugins required to read the data, or Intake itself.
- Although it is technically possible to embed plugin code into a data package, this is discouraged. It is better to break that code out into a separate package so that it can be updated independent of the data.
- Anaconda Cloud accounts have disk usage limits, so be careful uploading data packages there. You may want to host them on a separate web server or cloud storage bucket. [conda index](#) will help you construct the required JSON metadata to host conda packages.

## 5.11 Plotting

Intake provides a plotting API based on the [hvPlot](#) library, which closely mirrors the pandas plotting API but generates interactive plots using [HoloViews](#) and [Bokeh](#).

The [hvPlot website](#) provides comprehensive documentation on using the plotting API to quickly visualize and explore small and large datasets. The main features offered by the plotting API include:

- Support for tabular data stored in pandas and dask dataframes
- Support for gridded data stored in xarray backed nD-arrays
- Support for plotting large datasets with [datashader](#)

Using Intake alongside hvPlot allows declaratively persisting plot declarations and default options in the regular catalog.yaml files.

### 5.11.1 Setup

For detailed installation instructions see the [getting started](#) section in the hvPlot documentation. To start with install hvplot using conda:

```
conda install -c pyviz hvplot
```

or using pip:

```
pip install hvplot
```

### 5.11.2 Usage

The plotting API is designed to work well in and outside the Jupyter notebook, however when using it in JupyterLab the PyViz lab extension must be installed first:

```
jupyter labextension install @pyviz/jupyterlab_pyviz
```

For detailed instructions on displaying plots in the notebook and from the Python command prompt see the [hvPlot user guide](#).

### Python Command Prompt & Scripts

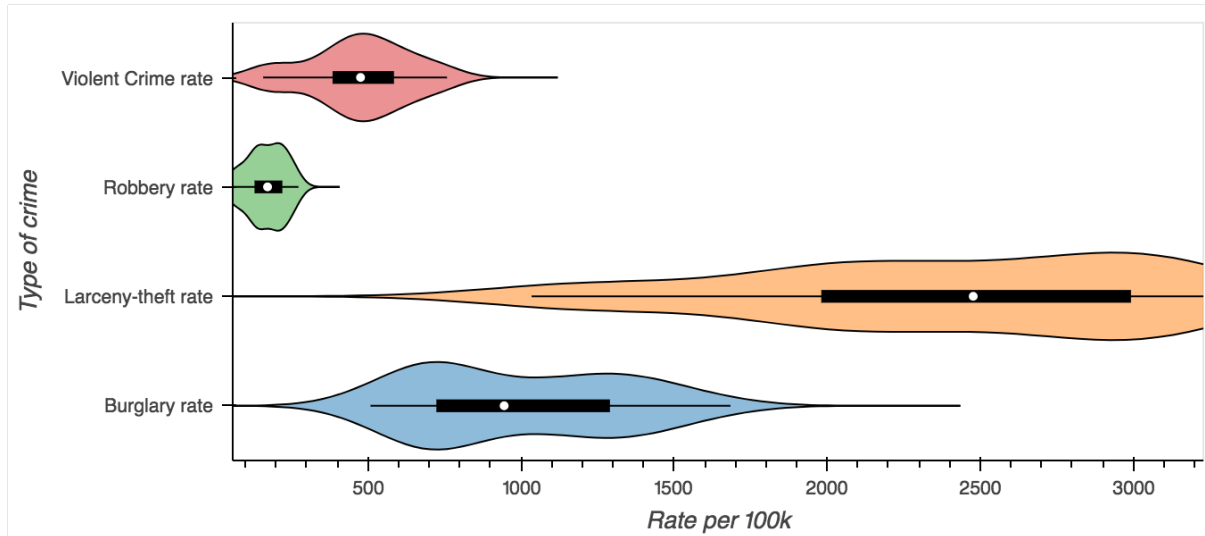
Assuming the US Crime dataset has been installed (in this repo's *examples/* directory, or from conda with *conda install -c intake us\_crime*):

Once installed the plot API can be used, by using the `.plot` method on an intake DataSource:

```
import intake
import hvplot as hp

crime = intake.cat.us_crime
columns = ['Burglary rate', 'Larceny-theft rate', 'Robbery rate', 'Violent Crime rate
↔']

violin = crime.plot.violin(y=columns, group_label='Type of crime',
                           value_label='Rate per 100k', invert=True)
hp.show(violin)
```



## Notebook

Inside the notebook plots will display themselves, however the notebook extension must be loaded first. The extension may be loaded by importing `hvplot.intake` module or explicitly loading the holoviews extension, or by calling `intake.output_notebook()`:

```
# To load the extension run this import
import hvplot.intake

# Or load the holoviews extension directly
import holoviews as hv
hv.extension('bokeh')

# convenience function
import intake
intake.output_notebook()

crime = intake.cat.us_crime
columns = ['Violent Crime rate', 'Robbery rate', 'Burglary rate']
crime.plot(x='Year', y=columns, value_label='Rate (per 100k people)')
```

## Predefined Plots

Some catalogs will define plots appropriate to a specific data source. These will be specified such that the user gets the right view with the right columns and labels, without having to investigate the data in details - this is idea for quick-look plotting when browsing sources.

```
import intake
intake.us_crime.plots
```

Returns `[example]`. This works whether accessing the entry object or the source instance. To visualise

```
intake.us_crime.plot.example()
```

### 5.11.3 Persisting metadata

Intake allows catalog yaml files to declare metadata fields for each data source which are made available alongside the actual dataset. The plotting API reserves certain fields to define default plot options, to label and annotate the data fields in a dataset and to declare pre-defined plots.

#### Declaring defaults

The first set of metadata used by the plotting API is the *plot* field in the metadata section. Any options found in the metadata field will apply to all plots generated from that data source, allowing the definition of plotting defaults. For example when plotting a fairly large dataset such as the NYC Taxi data, it might be desirable to enable datashader by default ensuring that any plot that supports it is datashaded. The syntax to declare default plot options is as follows:

```
sources:
  nyc_taxi:
    description: NYC Taxi dataset
    driver: parquet
    args:
      urlpath: 's3://datashader-data/nyc_taxi_wide.parq'
    metadata:
      plot:
        datashade: true
```

#### Declaring data fields

The columns of a CSV or parquet file or the coordinates and data variables in a NetCDF file often have shortened, or cryptic names with underscores. They also do not provide additional information about the units of the data or the range of values, therefore the catalog yaml specification also provides the ability to define additional information about the *fields* in a dataset.

Valid attributes that may be defined for the data *fields* include:

- *label*: A readable label for the field which will be used to label axes and widgets
- *unit*: A unit associated with the values inside a data field
- *range*: A range associated with a field declaring limits which will override those computed from the data

Just like the default plot options the *fields* may be declared under the metadata section of a data source:

```
sources:
  nyc_taxi:
    description: NYC Taxi dataset
    driver: parquet
    args:
      urlpath: 's3://datashader-data/nyc_taxi_wide.parq'
    metadata:
      fields:
        dropoff_x:
          label: Longitude
        dropoff_y:
          label: Latitude
        total_fare:
          label: Fare
          unit: $
```



## Declaring custom plots

As shown in the [hvPlot user guide](#), the plotting API provides a variety of plot types, which can be declared using the *kind* argument or via convenience methods on the plotting API, e.g. `cat.source.plot.scatter()`. In addition to declaring default plot options and field metadata data sources may also declare custom plot, which will be made available as methods on the plotting API. In this way a catalogue may declare any number of custom plots alongside a datasource.

To make this more concrete consider the following custom plot declaration on the *plots* field in the metadata section:

```
sources:
  nyc_taxi:
    description: NYC Taxi dataset
    driver: parquet
    args:
      urlpath: 's3://datashader-data/nyc_taxi_wide.parq'
    metadata:
      plots:
        dropoff_scatter:
          kind: scatter
          x: dropoff_x
          y: dropoff_y
          datashade: True
          width: 800
          height: 600
```

This declarative specification creates a new custom plot called *dropoff\_scatter*, which will be available on the catalog under `cat.nyc_taxi.plot.dropoff_scatter()`. Calling this method on the plot API will automatically generate a datashaded scatter plot of the dropoff locations in the NYC taxi dataset.

Of course the three metadata fields may also be used together, declaring global defaults under the *plot* field, annotations for the data *fields* under the *fields* key and custom plots via the *plots* field.

## 5.12 Plugin Directory

This is a list of known projects which install driver plugins for Intake:

- [intake-astro](#) Table and array loading of FITS astronomical data
- [intake-accumulo](#) Apache Accumulo clustered data storage
- [intake-avro](#): Apache Avro data serialization format
- [intake-dynamodb](#)
- [intake-elasticsearch](#): Elasticsearch search and analytics engine
- [intake-geopandas](#): load ESRI Shape Files with geopandas
- [intake-hbase](#): Apache HBase database
- [intake-iris](#) load netCDF and GRIB files with IRIS
- [intake-mongo](#): MongoDB noSQL query
- [intake-netflow](#): Netflow packet format
- [intake-odbc](#): ODBC database
- [intake-parquet](#): Apache Parquet file format
- [intake-pcap](#): PCAP network packet format

- `intake-postgres`: PostgreSQL database
- `intake-s3-manifests`
- `intake-solr`: Apache Solr search platform
- `intake-sql`: Generic SQL queries via SQLAlchemy
- `intake-splunk`: Splunk machine data query
- `intake-xarray`: load netCDF, Zarr and other multi-dimensional data

Don't see your favorite format? See *Making Drivers* for how to create new plugins.

Note that if you want your plugin listed here, open an issue in the [Intake issue repository](#). We also have a [plugin wishlist Github issue](#) that shows the breadth of plugins we hope to see for Intake.

## 5.13 Roadmap

Some high-level work that we expect to be achieved on the time-scale of months. This list is not exhaustive, but rather aims to whet the appetite for what Intake can be in the future.

Since Intake aims to be a community of data-oriented pythoners, nothing written here is laid in stone, and users and devs are encouraged to make their opinions known!

### 5.13.1 Broaden the coverage of formats

Data-type drivers are easy to write, but still require some effort, and therefore reasonable impetus to get the work done. Conversations over the coming months can help determine the drivers that should be created by the Intake team, and those that might be contributed by the community.

Similarly, there are many third-party data services, some of which are paid and proprietary. Currently, `SQLCatalog` is the only example, and therefore reference, of this, but Intake will become more universally useful when it can act as a bridge to several other systems.

### 5.13.2 Integration with Apache Spark

The spark ecosystems and Intake will co-operate nicely! Firstly, Spark sources (i.e., named tables) will become standard data sources, so that the data can be streamed from Spark to a python process, and the data-sets referenced in a catalog as usual. These data-sets will necessarily be data-frame type, although an RDD-to-sequential method may also be possible

Later, automatic streaming of data *into* Spark should be possible also, with a `to_spark()` method appearing on data-frame (and maybe sequence, later) type sources.

### 5.13.3 Derived Data-sets

Often, we can conceive of a data-type as being a modified version of another data-type. For example: the “csv” plugin produced data-frames from a set of files in the CSV format, while another plugin takes data-frames with a particular set of fields as input, and produces new data-frames based on some model predictions.

Rather than allow a general pipeline with arbitrary code specified in catalogues, we aim to allow the creation of arbitrary *plugins*, where the inputs are the outputs of other data-sources. This way, the logic stays in the code of the plugin, which can be distributed as python/conda packages as usual, but a path is in place to generate “second-order”

data products. Naturally, such derived plugins ought to be thorough about describing the process in the metadata of the resultant data-source. Preliminary work: [PR](#).

## 5.14 Glossary

**Catalog** a collection of entries, each of which corresponds to a specific *Data-set*. Within Intake, a catalog is most commonly defined in a *YAML* file, but there are other possibilities, such as connecting to an Intake server or another third-party data service, like a SQL database. Thus, catalogs form a hierarchy: any catalog can contain other, nested catalogs.

**Conda** a package and environment management package for the python ecosystem, see the [conda website](#). Conda ensures dependencies and correct versions are installed for you, provides precompiled, binary-compatible software, and extends to many languages beyond python, such as R, javascript and C.

**Conda package** a single installable item which the *Conda* application can install. A package may include a *Catalog*, data-files and maybe some additional code. It will also include a specification of the dependencies that it requires (e.g., Intake and any additional *Driver*), so that Conda can install those automatically. Packages can be created locally, or can be found on [anaconda.org](#) or other package repositories.

**Container** one of a small number of internal data formats, each *Driver* outputs its data in one of these. The containers correspond to familiar data structures for end-analysis, such as list-of-dicts, Numpy nd-array or Pandas data-frame.

**Data-set** a specific collection of data. The type of data (tabular, multi-dimensional or something else) and the format (file type, data service type) are all attributes of the data-set. In addition, in the context of Intake, data-sets are usually entries within a *Catalog* with additional descriptive text and metadata and a specification of *how* to load the data.

**Data Source** an Intake specification for a specific *Data-set*. In most cases, the two terms are synonymous.

**Data User** a person who uses data to produce models and other inferences/conclusions. This person generally uses standard python analysis packages like Numpy, Pandas, SKLearn and may produce graphical output. They will want to be able to find the right data for a given job, and for the data to be available in a standard format as quickly and easily as possible. In many organisations, the appropriate job title may be Data Scientist, but research scientists and BI/analysts also fit this description.

**Data Provider** the main objective here is to curate data sources, get them into appropriate formats, describe the contents, and disseminate the data to those that need to use them. Such a person may care about the specifics of the storage format and backing store, the right number of fields to keep and removing bad data. They may have a good idea of the best way to visualise any give data-set. In an organisation, this job may be known as Data Engineer, but it could as easily be done by a member of the IT team. These people are the most likely to author *Catalogs*.

**Developer** a person who writes or fixes code. In the context of Intake, a developer may make new format *Drivers*, create authentication systems or add functionality to Intake itself. They can take existing code for loading data in other projects, and use Intake to add extra functionality to it, for instance, remote data access, parallel processing, or file-name parsing.

**Driver** each catalog entry describes a specific data-set, to be opened in a specific way. The thing that will do the work of reading the data is known as a driver, often referred to using a simple name such as “csv”. Intake has a plugin architecture, and new drivers can be created or installed, and specific catalogs/data-sets may require particular drivers for their contained data-sets. If installed as *Conda* packages, then these requirements will be automatically installed for you. The driver’s output will be a *Container*, and often the code is a simpler layer over existing functionality in a third-party package.

**GUI** Intake comes with a graphical user interface for finding and selecting data-sets, based on jupyter-widgets; see [GUI](#).

**IT** many organisations will have a dedicated IT team. As a generalisation, such a team may have control of the computing infrastructure and security (sys-ops), and may well act as gate-keepers when exposing data for use by other colleagues. Commonly, IT has stronger policy enforcement requirements than other groups, for instance requiring all data-set copy actions to be logged centrally.

**Plugin** modular extra functionality for Intake, provided by a package that is installed separately. The most common type of plugin will be for a *Driver* to load some particular data format; but other parts of Intake are pluggable, such as authentication mechanisms for the server.

**Server** Intake can either read catalogs and data directly, or can talk with an Intake server. The server will provide data source specifications (i.e., a remote *Catalog*), and may also provide the raw data, if the client is not able or not allowed to access it directly. As such, the server can act as a gatekeeper of the data for security and monitoring purposes. The implementation of the server in Intake is accessible as the `intake-server` command, and acts as a reference: other implementations can easily be created for specific circumstances.

**YAML** a text-based format for expressing data with a dictionary (key-value) and list structure, with a limited number of data-types, such as strings and numbers. YAML uses indentations to nest objects, making it easy to read and write for humans, compares to JSON. Intake's catalogs and config are usually expressed in YAML files.

## CHAPTER 6

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



## Symbols

`__init__()` (*intake.catalog.local.YAMLFileCatalog method*), 23  
`__init__()` (*intake.catalog.local.YAMLFilesCatalog method*), 23  
`__init__()` (*intake.source.base.AliasSource method*), 29  
`__init__()` (*intake.source.csv.CSVSource method*), 21  
`__init__()` (*intake.source.npy.NPySource method*), 22  
`__init__()` (*intake.source.textfiles.TextFilesSource method*), 22

## A

`add_cat()` (*intake.gui.DataBrowser method*), 24  
*AliasSource* (*class in intake.source.base*), 29  
`allow_access()` (*intake.auth.base.BaseAuth method*), 28  
`allow_connect()` (*intake.auth.base.BaseAuth method*), 28

## B

*BaseAuth* (*class in intake.auth.base*), 27  
*BaseCache* (*class in intake.source.cache*), 28

## C

*Catalog*, 55  
*Catalog* (*class in intake.catalog*), 26  
*CatalogEntry* (*class in intake.catalog.entry*), 26  
`clear_all()` (*intake.source.cache.BaseCache method*), 28  
`clear_cache()` (*intake.source.cache.BaseCache method*), 28  
`close()` (*intake.source.base.DataSource method*), 25  
Conda, 55  
Conda package, 55  
Container, 55  
*CSVSource* (*class in intake.source.csv*), 21

## D

Data Provider, 55  
Data Source, 55  
Data User, 55  
Data-set, 55  
*DataBrowser* (*class in intake.gui*), 24  
*DataSource* (*class in intake.source.base*), 25  
`describe()` (*intake.catalog.entry.CatalogEntry method*), 26  
`describe()` (*intake.catalog.local.UserParameter method*), 27  
`describe_open()` (*intake.catalog.entry.CatalogEntry method*), 26  
Developer, 55  
`discover()` (*intake.catalog.Catalog method*), 26  
`discover()` (*intake.source.base.DataSource method*), 25  
`discover()` (*intake.source.csv.CSVSource method*), 21  
`discover()` (*intake.source.npy.NPySource method*), 23  
`discover()` (*intake.source.textfiles.TextFilesSource method*), 22  
Driver, 55

## E

`expand_defaults()` (*intake.catalog.local.UserParameter method*), 27

## F

`file_chosen()` (*intake.gui.DataBrowser method*), 24  
`force_reload()` (*intake.catalog.Catalog method*), 26

## G

`get()` (*intake.catalog.entry.CatalogEntry method*), 27  
`get_case_insensitive()` (*intake.auth.base.BaseAuth method*), 28  
`get_metadata()` (*intake.source.cache.BaseCache method*), 28

GUI, [55](#)

## H

hvplot (*intake.source.base.DataSource* attribute), [25](#)

## I

IT, [56](#)

## L

load() (*intake.source.cache.BaseCache* method), [28](#)

## N

NPySource (*class in intake.source.npy*), [22](#)

## O

open\_ (*intake* attribute), [21](#)

open\_catalog() (*in module intake*), [20](#)

## P

PatternMixin (*class in intake.source.base*), [29](#)

plot (*intake.source.base.DataSource* attribute), [25](#)

plots (*intake.catalog.entry.CatalogEntry* attribute), [27](#)

plots (*intake.source.base.DataSource* attribute), [25](#)

Plugin, [56](#)

## R

read() (*intake.source.base.DataSource* method), [25](#)

read() (*intake.source.csv.CSVSource* method), [21](#)

read() (*intake.source.npy.NPySource* method), [23](#)

read() (*intake.source.textfiles.TextFilesSource* method), [22](#)

read\_chunked() (*intake.source.base.DataSource* method), [25](#)

read\_partition() (*intake.source.base.DataSource* method), [25](#)

read\_partition() (*intake.source.csv.CSVSource* method), [21](#)

read\_partition() (*intake.source.npy.NPySource* method), [23](#)

read\_partition() (*intake.source.textfiles.TextFilesSource* method), [22](#)

registry (*intake* attribute), [21](#)

reload() (*intake.catalog.Catalog* method), [26](#)

reload() (*intake.catalog.local.YAMLFileCatalog* method), [23](#)

reload() (*intake.catalog.local.YAMLFilesCatalog* method), [23](#)

RemoteSource (*class in intake.container.base*), [27](#)

## S

Schema (*class in intake.source.base*), [29](#)

Server, [56](#)

## T

TextFilesSource (*class in intake.source.textfiles*), [22](#)

to\_dask() (*intake.container.base.RemoteSource* method), [27](#)

to\_dask() (*intake.source.base.DataSource* method), [25](#)

to\_dask() (*intake.source.csv.CSVSource* method), [22](#)

to\_dask() (*intake.source.npy.NPySource* method), [23](#)

to\_dask() (*intake.source.textfiles.TextFilesSource* method), [22](#)

## U

UserParameter (*class in intake.catalog.local*), [27](#)

## V

validate() (*intake.catalog.local.UserParameter* method), [27](#)

## W

walk() (*intake.catalog.Catalog* method), [26](#)

walk() (*intake.catalog.local.YAMLFileCatalog* method), [23](#)

walk() (*intake.catalog.local.YAMLFilesCatalog* method), [24](#)

## Y

YAML, [56](#)

yaml() (*intake.source.base.DataSource* method), [25](#)

YAMLFileCatalog (*class in intake.catalog.local*), [23](#)

YAMLFilesCatalog (*class in intake.catalog.local*), [23](#)