pydatatask

Release 0.4.3

Audrey Dutcher

Apr 27, 2023

CONTENTS:

Welcome to pydatatask! These docs have two sections, a quickstart and an API reference.

CHAPTER

ONE

QUICKSTART

pydatatask is a library for building data pipelines. Sounds familiar? The cool part here is that you are not restricted in the way your data is stored or the way your tasks are executed.

1.1 Installing

```
pip install pydatatask
```

1.2 Nomenclature

A **task** is one phase of computation. It is parameterized (instantiated) by a single **job** that it is currently working on. A **pipeline** is a collection of tasks. Tasks read and write data from **repositories**, which are arbitrary key-value stores.

1.3 The way your data is stored

Repository classes are the core of pydatatask. You can store your data in any way you desire and as long as you can write a repository class to describe it, it can be used to drive a pipeline.

The notion of the "value" part of the key-value store abstraction is defined very, very loosely. The repository base class doesn't have an interface to get or store values, only to query for and delete keys. Instead, you have to know which repository subclass you're working with, and use its interfaces. For example, *MetadataRepository* assumes that its values are structured objects and loads them fully into memory, and *BlobRepository* provides a streaming interface to a flat address space.

Current in-tree repositories:

- In-memory dicts
- Files or directories on the local filesystem
- S3 or compatible buckets
- MongoDB collections
- Docker repositories
- Various combinators

1.4 The way your tasks are executed

A *Task* is connected to repositories through *links*. A link is a repository plus a collection of properties describing the repository's relationship to the task - i.e. whether it is input or output, whether it should inhibit dependent tasks from starting, etc.

Current in-tree task types:

- In-process python function execution
- Python function execution with the help of a concurrent.futures Executor
- Python function execution on a kubernetes cluster
- Script execution on a kubernetes cluster
- Script execution locally or over SSH

Most tasks define the notion of an **environment** which is used to template the task for the particular job that is being run.

1.5 Management of resources: the Session

A *Session* is a tool for managing multiple live resources. After constructing a session, you can register async resource manager routines. You will receive in return a callable which will return the live resource while the session is opened. This means that a pipeline and all its resources can be defined in a synchronous context, and then allocated and connected whenthe async context is activated.

1.6 Putting it together: the Pipeline object

A *Pipeline* is just an unordered collection of tasks paired with a Session. Relationships between the tasks are implicit, defined by which repositories they share.

1.7 Example

```
import os
import aiobotocore.session
import pydatatask

session = pydatatask.Session()

@session.resource
async def bucket():
    bucket_session = aiobotocore.session.get_session()
    async with bucket_session.create_client(
        's3',
        endpoint_url=os.getenv('BUCKET_ENDPOINT'),
        aws_access_key_id=os.getenv("BUCKET_USERNAME"),
        aws_secret_access_key=os.getenv("BUCKET_PASSWORD"),
    ) as client:
        yield client
```

(continues on next page)

(continued from previous page)

```
books_repo = pydatatask.S3BucketRepository(bucket, "books/", '.txt')
done_repo = pydatatask.YamlMetadataFileRepository('./results/')
reports_repo = pydatatask.FileRepository('./reports', '.txt')

@pydatatask.InProcessSyncTask('summary', done_repo)
async def summary(job: str, books: pydatatask.S3BucketRepository, reports: pydatatask.
↪FileRepository):
    paragraphs, lines, words, chars = 0, 0, 0, 0
    async with await books.open(job, 'r') as fp:
        data = await fp.read()
    for line in data.splitlines():
        if line.strip() == '':
            paragraphs += 1
        lines += 1
        words += len(line.split())
        chars += len(line)
    async with await reports.open(job, 'w') as fp:
        await fp.write(f'The book "{job}" has {paragraphs} paragraphs, {lines} lines,
↪{words} words, and {chars} characters.\n')

summary.link('books', books_repo, is_input=True)
summary.link('reports', reports_repo, is_output=True)

pipeline = pydatatask.Pipeline([summary], session)

if __name__ == '__main__':
    pydatatask.main(pipeline)
```

CHAPTER

TWO

PYDATATASK PACKAGE

Documentation for this package can be found at https://pydatatask.readthedocs.io/en/stable/

2.1 Submodules

2.1.1 pydatatask.fuse module

2.1.2 pydatatask.main module

The top-level script you write using pydatatask should call *pydatatask.main.main* in its `if __name__ == '__main__'` block. This will parse `sys.argv` and display the administration interface for the pipeline.

The help screen should look something like this:

```
$ python3 main.py --help
  usage: main.py [-h] {update,run,status,trace,rm,ls,cat,inject,launch,shell} ...

positional arguments:
  {update,run,status,trace,rm,ls,cat,inject,launch,shell}
    update              Keep the pipeline in motion
    run                 Run update in a loop until everything is quiet
    status              View the pipeline status
    trace               Track a job's progress through the pipeline
    rm                  Delete data from the pipeline
    ls                  List jobs in a repository
    cat                 Print data from a repository
    inject              Dump data into a repository
    launch              Manually start a task
    shell               Launch an interactive shell to interrogate the pipeline

options:
  -h, --help            show this help message and exit
```

pydatatask.main.**main**(*pipeline:* Pipeline, *instrument:* *Callable[[_SubParsersAction], None]* | *None* = *None*)

The pydatatask main function! Call this with the pipeline you've constructed to parse `sys.argv` and display the pipeline administration interface.

If you like, you can pass as the `instrument` argument a function which will add additional commands to the menu.

async pydatatask.main.**update**(*pipeline:* Pipeline)

async pydatatask.main.**cat_data**(*pipeline:* Pipeline, *data: str*, *job: str*)

async pydatatask.main.**list_data**(*pipeline:* Pipeline, *data: List[str]*)

async pydatatask.main.**delete_data**(*pipeline:* Pipeline, *data: str*, *recursive: bool*, *job: List[str]*)

async pydatatask.main.**inject_data**(*pipeline:* Pipeline, *data: str*, *job: str*)

async pydatatask.main.**print_status**(*pipeline:* Pipeline, *all_repos: bool*)

async pydatatask.main.**print_trace**(*pipeline:* Pipeline, *all_repos: bool*, *job: List[str]*)

async pydatatask.main.**launch**(*pipeline:* Pipeline, *task_name: str*, *job: str*, *sync: bool*, *meta: bool*, *force: bool*)

pydatatask.main.**shell**(*pipeline:* Pipeline)

async pydatatask.main.**run**(*pipeline:* Pipeline, *forever: bool*, *launch_once: bool*, *timeout: float | None*)

2.1.3 pydatatask.pipeline module

A pipeline is just an unordered collection of tasks. Relationships between the tasks are implicit, defined by which repositories they share.

class pydatatask.pipeline.**Pipeline**(*tasks: Iterable[Task]*, *session:* Session, *resources: Iterable[ResourceManager]*, *priority: Callable[[str, str], int] | None = None*)

Bases: object

The pipeline class.

Parameters

- **tasks** – The tasks which make up this pipeline.
- **session** – The session to open while this pipeline is active.
- **resources** – Any resource managers in use. You need to provide these so the pipeline can reset the rate-limiting at each update.
- **priority** – Optional: A function which takes a task and job name and returns an integer priority. No jobs will be scheduled unless all higher-priority jobs (larger numbers) have already been scheduled.

settings(*synchronous=False*, *metadata=True*)

This method can be called to set properties of the current run.

Parameters

- **synchronous** – Whether jobs will be started and completed in-process, waiting for their completion before a launch phase succeeds.
- **metadata** – Whether jobs will store their completion metadata.

async **open**()

Opens the pipeline and its associated resource session. This will be automatically when entering an `async with pipeline:` block.

async close()

Closes the pipeline and its associated resource session. This will be automatically called when exiting an `async with pipeline:` block.

async update() → bool

Perform one round of pipeline maintenance, running the update phase and then the launch phase. The pipeline must be opened for this function to run.

Returns
Whether there is any activity in the pipeline.

async update_only_update() → bool

Perform one round of the update phase of pipeline maintenance. The pipeline must be opened for this function to run.

Returns
Whether there were any live jobs.

async update_only_launch() → bool

Perform one round of the launch phase of pipeline maintenance. The pipeline must be opened for this function to run.

Returns
Whether there were any jobs launched or ready to launch.

async gather_ready_jobs(*task:* Task) → Set[str]

Collect all jobs that are ready to be launched for a given task.

graph() → DiGraph

Generate the dependency graph for a pipeline. This is a directed graph containing both repositories and tasks as nodes.

dependants(*node:* Repository | Task, *recursive: bool*) → Iterable[*Repository* | *Task*]

Iterate the list of repositories that are dependent on the given node of the dependency graph.

Parameters

- **node** – The starting point of the graph traversal.
- **recursive** – Whether to return only direct dependencies (False) or transitive dependencies (True) of node.

2.1.4 pydatatask.pod_manager module

In order for a *KubeTask* or a subclass to connect, authenticate, and manage pods in a kubernetes cluster, it needs several resource references. the *PodManager* simplifies tracking the lifetimes of these resources.

class pydatatask.pod_manager.**PodManager**(*app: str*, *namespace: str*, *config: Callable[[], Configuration] | None = None*)

Bases: object

A pod manager allows multiple tasks to share a connection to a kubernetes cluster and manage pods on it.

Parameters

- **app** – The app name string with which to label all created pods.
- **namespace** – The namespace in which to create and query pods.

- **config** – Optional: A callable returning a kubernetes configuration object. If not provided, will attempt to use the "default" configuration, i.e. what is available after calling `await kubernetes_asyncio.config.load_kube_config()`.

property api

The current API client.

property api_ws: WsApiClient

The current websocket-aware API client.

property v1: CoreV1Api

A CoreV1Api instance associated with the current API client.

property v1_ws: CoreV1Api

A CoreV1Api instance associated with the current websocket-aware API client.

async close()

Close the network connections associated with this podman.

async launch(*job*, *task*, *manifest*)

Create a pod with the given manifest, named and labeled for this podman's app and the given job and task.

async query(*job=None*, *task=None*) → List[V1Pod]

Return a list of pods labeled for this podman's app and (optional) the given job and task.

async delete(*pod: V1Pod*)

Destroy the given pod.

async logs(*pod: V1Pod*, *timeout=10*) → str

Retrieve the logs for the given pod.

2.1.5 pydatatask.proc_manager module

A process manager can be used with a ProcessTask to specify where and how the processes should run. All a process manager needs to specify is how to launch a process and manipulate the filesystem, and the ProcessTask will set up an appropriate environment for running the task and retrieve the results using this interface.

class pydatatask.proc_manager.**AbstractProcessManager**

Bases: object

The base class for process managers.

Processes are managed through arbitrary "process identifier" handle strings.

abstract async get_live_pids(*hint: Set[str]*) → Set[str]

Get a set of the live process identifiers. This may include processes which are not part of the current app/task, but MUST include all the processes in `hint` which are still alive.

abstract async spawn(*args: List[str]*, *environ: Dict[str, str]*, *cwd: str*, *return_code: str*, *stdin: str | None*, *stdout: str | None*, *stderr: str | _StderrIsStdout | None*) → str

Launch a process on the target system. This function MUST NOT wait until the process has terminated before returning.

Parameters

- **args** – The command line of the process to launch. `args[0]` is the executable to run.
- **environ** – A set of environment variables to add to the target process' environment.

- **cwd** – The directory to launch the process in.
- **return_code** – A filepath in which to store the process exit code, as an ascii integer.
- **stdin** – A filepath from which to read the process' stdin, or None for a null file.
- **stdout** – A filepath to which to write the process' stdout, or None for a null file.
- **stderr** – A filepath to which to write the process' stderr, None for a null file, or the constant *pydatatask.task.STDOUT* to interleave it into the stdout stream.

Returns
The process identifier of the process spawned.

abstract async kill(*pid: str*)

Terminate the process with the given identifier. This should do nothing if the process is not currently running. This should not prevent e.g. the `return_code` file from being populated.

abstract property basedir: Path

A path on the target system to a directory which can be freely manipulated by the app.

abstract async open(*path: Path*, *mode: Literal['r', 'rb', 'w', 'wb']*) → *AReadText* | *AWriteText* | *AReadStream* | *AWriteStream*

Open the file on the target system for reading or writing according to the given mode.

abstract async mkdir(*path: Path*)

Create a directory with the given path on the target system.

This should have the semantics of `mkdir -p`, i.e. create any necessary parent directories and also succeed if the directory already exists.

abstract async rmtree(*path: Path*)

Remove a directory and all its children with the given path on the target system.

class pydatatask.proc_manager.**LocalLinuxManager**(*app: str*, *local_path: Path | str = '/tmp/pydatatask'*)

Bases: *AbstractProcessManager*

A process manager to run tasks on the local linux machine. By default, it will create a directory `/tmp/pydatatask` in which to store data.

property basedir: Path

async get_live_pids(*hint: Set[str]*) → Set[str]

async spawn(*args*, *environ*, *cwd*, *return_code*, *stdin*, *stdout*, *stderr*)

async kill(*pid: str*)

async mkdir(*path: Path*)

async rmtree(*path: Path*)

async open(*path*, *mode*)

class pydatatask.proc_manager.**SSHLinuxManager**(*app: str*, *ssh: Callable[[], SSHClientConnection]*, *remote_path: Path | str = '/tmp/pydatatask'*)

Bases: *AbstractProcessManager*

A process manager that runs its processes on a remote linux machine, accessed over SSH. The SSH connection is parameterized by an `asyncssh.connection.SSHClientConnection` instance.

Sample usage:

```
session = pydatatask.Session()

@session.resource
async def ssh():
    async with asyncssh.connect(
        "localhost", port=self.port, username="weh", password="weh", known_
↪hosts=None
    ) as s:
        yield s

@session.resource
async def procman():
    yield pydatatask.SSHLinuxManager(self.test_id, ssh)

task = pydatatask.ProcessTask("mytask", procman, ...)
```

property ssh: SSHClientConnection

The asyncssh.SSHClientConnection instance associated. Will fail if the connection is provided by an unopened Session.

property basedir: Path

async open(*path*, *mode*)

async rmtree(*path: Path*)

async mkdir(*path: Path*)

async kill(*pid: str*)

async get_live_pids(*hint*)

async spawn(*args*, *environ*, *cwd*, *return_code*, *stdin*, *stdout*, *stderr*)

2.1.6 pydatatask.repository module

Repositories are arbitrary key-value stores. They are the data part of pydatatask. You can store your data in any way you desire and as long as you can write a Repository class to describe it, it can be used to drive a pipeline.

The notion of the "value" part of the key-value store abstraction is defined very, very loosely. The repository base class doesn't have an interface to get or store values, only to query for and delete keys. Instead, you have to know which repository subclass you're working with, and use its interfaces. For example, *MetadataRepository* assumes that its values are structured objects and loads them fully into memory, and *BlobRepository* provides a streaming interface to a flat address space.

class pydatatask.repository.**Repository**

Bases: ABC

A repository is a key-value store where the keys are names of jobs. Since the values have unspecified semantics, the only operations you can do on a generic repository are query for keys.

A repository can be async-iterated to get a listing of its members.

CHARSET = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'

CHARSET_START_END = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'

classmethod is_valid_job_id(*job: str*)

Determine whether the given job identifier is valid, i.e. that it contains only valid characters (numbers and letters by default).

async filter_jobs(*iterator: AsyncIterable[str]*) → AsyncIterable[str]

Apply *is_valid_job_id* as a filter to an async iterator.

async contains(*item*)

Determine whether the given job identifier is present in this repository.

The default implementation is quite inefficient; please override this if possible.

abstract async unfiltered_iter() → AsyncGenerator[str, None]

The core method of Repository. Implement this to produce an iterable of every string which could potentially be a job identifier present in this repository. When the repository is iterated directly, this will be filtered by *filter_jobs*.

abstract async info(*job*) → Any

Returns an arbitrary piece of data related to job. Notably, this is used during templating. This should do something meaningful even if the repository does not contain the requested job.

abstract async delete(*job*)

Delete the given job from the repository. This should succeed even if the job is not present in this repository.

async info_all() → Dict[str, Any]

Produce a mapping from every job present in the repository to its corresponding info. The default implementation is somewhat inefficient; please override it if there is a more effective way to load all info.

async validate()

Override this method to raise an exception if for any reason the repository is misconfigured. This will be automatically called by the pipeline on opening.

map(*func: Callable*, *filt: Callable[[str], Awaitable[bool]] | None = None*, *allow_deletes=False*) → MapRepository

Generate a `MapRepository` based on this repository and the given parameters.

class pydatatask.repository.**BlobRepository**

Bases: *Repository*, ABC

A blob repository has values which are flat data blobs that can be streamed for reading or writing.

abstract async open(*job: str*, *mode: Literal['r']*) → *AReadText*

abstract async open(*job: str*, *mode: Literal['rb']*) → *AReadStream*

abstract async open(*job: str*, *mode: Literal['w']*) → *AWriteText*

abstract async open(*job: str*, *mode: Literal['wb']*) → *AWriteStream*

Open the given job's value as a stream for reading or writing, in text or binary mode.

class pydatatask.repository.**MetadataRepository**

Bases: *Repository*, ABC

A metadata repository has values which are small, structured data, and loads them entirely into memory, returning the structured data from the *info* method.

abstract async info(*job*)

Retrieve the data with key `job` from the repository.

abstract async dump(*job*, *data*)

Insert `data` into the repository with key `job`.

class pydatatask.repository.**FileRepositoryBase**(*basedir*, *extension=''*, *case_insensitive=False*)

Bases: *Repository*, ABC

A file repository is a local directory where each job identifier is a filename, optionally suffixed with an extension before hitting the filesystem. This is an abstract base class for other file repositories which have more to say about what is found at these filepaths.

async contains(*item*)

async unfiltered_iter()

async validate()

fullpath(*job*) → Path

Construct the full local path of the file corresponding to `job`.

async info(*job*)

The templating info provided by a file repository is the full path to the corresponding file as a string.

class pydatatask.repository.**FileRepository**(*basedir*, *extension=''*, *case_insensitive=False*)

Bases: *FileRepositoryBase*, *BlobRepository*

A file repository whose members are files, treated as streamable blobs.

async open(*job*, *mode='r'*)

async delete(*job*)

class pydatatask.repository.**DirectoryRepository**(**args*, *discard_empty=False*, ***kwargs*)

Bases: *FileRepositoryBase*

A file repository whose members are directories.

Parameters

discard_empty – Whether only directories containing at least one member should be considered as "present" in the repository.

async mkdir(*job*)

Create an empty directory corresponding to `job`. Do nothing if the directory already exists.

async delete(*job*)

async contains(*item*)

async unfiltered_iter()

class pydatatask.repository.**S3BucketRepository**(*client: Callable[[], S3Client]*, *bucket: str*, *prefix: str = ''*, *suffix: str = ''*, *mimetype: str = 'application/octet-stream'*, *incluster_endpoint: str | None = None*)

Bases: *BlobRepository*

A repository where keys are paths in a S3 bucket. Provides a streaming interface to the corresponding blobs.

Parameters

- **client** – A callable returning an aiobotocore S3 client connected and authenticated to the server you wish to store things on.
- **bucket** – The name of the bucket from which to load and store.

- **prefix** – A prefix to put on the job name before translating it into a bucket path. If this is meant to be a directory name it should end with a slash character.
- **suffix** – A suffix to put on the job name before translating it into a bucket path. If this is meant to be a file extension it should start with a dot.
- **mimetype** – The MIME type to set the content when adding data.
- **incluster_endpoint** – Optional: An endpoint URL to provide as the result of info() queries instead of extracting the URL from `client`.

property client

The aiobotocore S3 client. This will raise an error if the client comes from a session which is not opened.

async contains(*item*)

async unfiltered_iter()

async validate()

object_name(*job*)

Return the object name for the given job.

async open(*job*, *mode='r'*)

async info(*job*)

Return an *S3BucketInfo* corresponding to the given job.

async delete(*job*)

class pydatatask.repository.**S3BucketInfo**(*endpoint: str*, *uri: str*, *bucket: str*, *prefix: str*, *suffix: str*)

Bases: `object`

The data structure returned from *S3BucketRepository.info()*.

Variables

- **uri** – The s3 URI of the current job's resource, e.g. `s3://bucket/prefix/job.ext`. `str(info)` will also return this.
- **endpoint** – The URL of the API server providing the S3 interface.
- **bucket** – The name of the bucket objects are stored in.
- **prefix** – How to prefix an object name such that it will fit into this repository.
- **suffix** – How to suffix an object name such that it will fit into this repository.

class pydatatask.repository.**MongoMetadataRepository**(*collection: Callable[[], AsyncIOMotorCollection]*, *subcollection: str | None*)

Bases: *MetadataRepository*

A metadata repository using a MongoDB collection as the backing store.

Parameters

- **collection** – A callable returning a motor async collection.
- **subcollection** – Optional: the name of a subcollection within the collection in which to store data.

property collection: AsyncIOMotorCollection

The motor async collection data will be stored in. If this is provided by an unopened session, raise an error.

async contains(*item*)

async delete(*job*)

async unfiltered_iter()

async info(*job*)

The info of a mongo metadata repository is the literal value stored in the repository with identifier `job`.

async info_all() → Dict[str, Any]

async dump(*job*, *data*)

class pydatatask.repository.**InProcessMetadataRepository**(*data: Dict[str, Any] | None = None*)

Bases: *MetadataRepository*

An incredibly simple metadata repository which stores all its values in a dict, and will let them vanish when the process terminates.

async info(*job*)

async dump(*job*, *data*)

async contains(*item*)

async delete(*job*)

async unfiltered_iter()

class pydatatask.repository.**InProcessBlobStream**(*repo:* InProcessBlobRepository, *job: str*)

Bases: object

A stream returned from an *BlobRepository.open* call from *InProcessBlobRepository*. Do not construct this manually.

async read(*n: int | None = None*) → bytes

Read up to `n` bytes from the stream.

async write(*data: bytes*)

Write `data` to the stream.

async close()

Close and release the stream, syncing the data back to the repository.

class pydatatask.repository.**InProcessBlobRepository**(*data: Dict[str, bytes] | None = None*)

Bases: *BlobRepository*

An incredibly simple blob repository which stores all its values in a dict, and will let them vanish when the process terminates.

async info(*job*)

There is no templating info for an *InProcessBlobRepository*.

async open(*job*, *mode='r'*)

async unfiltered_iter()

async contains(*item*)

async delete(*job*)

class pydatatask.repository.**DockerRepository**(*registry: Callable[[], DockerRegistryClientAsync]*, *domain: str*, *repository: str*)

Bases: *Repository*

A docker repository is, well, an actual docker repository hosted in some registry somewhere. Keys translate to tags on this repository.

Parameters

- **registry** – A callable returning a docker_registry_client_async client object with appropriate authentication information.
- **domain** – The registry domain to connect to, e.g. `index.docker.io`.
- **repository** – The repository to store images in within the domain, e.g. `myname/myrepo`.

property registry: DockerRegistryClientAsync

The `docker_registry_client_async` client object. If this is provided by an unopened session, raise an error.

async unfiltered_iter()

async info(*job*)

The info provided by a docker repository is a dict with two keys, "withdomain" and "withoutdomain". e.g.:

```
{ "withdomain": "docker.example.com/myname/myrepo:job", "withoutdomain":
↪"myname/myrepo:job" }
```

async delete(*job*)

class pydatatask.repository.**LiveKubeRepository**(*task:* KubeTask)

Bases: *Repository*

A repository where keys translate to `job` labels on running kube pods. This repository is constructed automatically by a *KubeTask* or subclass and is linked as the `live` repository. Do not construct this class manually.

async unfiltered_iter()

async contains(*item*)

async info(*job*)

Cannot template with live kube info. Implement this if you have something in mind.

async pods() → List[V1Pod]

A list of live pod objects corresponding to this repository.

async delete(*job*)

Deleting a job from this repository will delete the pod.

class pydatatask.repository.**ExecutorLiveRepo**(*task:* ExecutorTask)

Bases: *Repository*

A repository where keys translate to running jobs in an ExecutorTask. This repository is constructed automatically and is linked as the `live` repository. Do not construct this class manually.

async **unfiltered_iter**()

async **contains**(*item*)

async **delete**(*job*)

Deleting a job from the repository will cancel the corresponding task.

async **info**(*job*)

There is no templating info for an *ExecutorLiveRepo*.

class pydatatask.repository.**AggregateOrRepository**(***children:* Repository)

Bases: *Repository*

A repository which is said to contain a job if any of its children also contain that job

async **unfiltered_iter**()

async **contains**(*item*)

async **info**(*job*)

The info provided by an aggregate Or repository is a dict mapping each child's name to that child's info.

async **delete**(*job*)

Deleting a job from an aggregate Or repository deletes the job from all of its children.

class pydatatask.repository.**AggregateAndRepository**(***children:* Repository)

Bases: *Repository*

A repository which is said to contain a job if all its children also contain that job

async **unfiltered_iter**()

async **contains**(*item*)

async **info**(*job*)

The info provided by an aggregate And repository is a dict mapping each child's name to that child's info.

async **delete**(*job*)

Deleting a job from an aggregate And repository deletes the job from all of its children.

class pydatatask.repository.**BlockingRepository**(*source:* Repository, *unless:* Repository, *enumerate_unless=True*)

Bases: *Repository*

A class that is said to contain a job if `source` contains it and `unless` does not contain it

async **unfiltered_iter**()

async **contains**(*item*)

async **info**(*job*)

async **delete**(*job*)

class pydatatask.repository.**YamlMetadataRepository**

Bases: *BlobRepository*, *MetadataRepository*, ABC

A metadata repository based on a blob repository. When info is accessed, it will **load the target file into memory**, parse it as yaml, and return the resulting object.

This is a base class, and must be overridden to implement the blob loading portion.

async **info**(*job*)

async **dump**(*job*, *data*)

class pydatatask.repository.**YamlMetadataFileRepository**(*filename*, *extension='.yaml'*, *case_insensitive=False*)

Bases: *YamlMetadataRepository*, *FileRepository*

A metadata repository based on a file blob repository.

class pydatatask.repository.**YamlMetadataS3Repository**(*client*, *bucket*, *prefix*, *suffix='.yaml'*, *mimetype='text/yaml'*)

Bases: *YamlMetadataRepository*, *S3BucketRepository*

A metadata repository based on a s3 bucket repository.

async **info**(*job*)

class pydatatask.repository.**RelatedItemRepository**(*base_repository: Repository*, *translator_repository: Repository*, *allow_deletes=False*, *prefetch_lookup=True*)

Bases: *Repository*

A repository which returns items from another repository based on following a related-item lookup.

Parameters

- **base_repository** – The repository from which to return results based on translated keys. The resulting repository will duck-type as the same type as the base.
- **translator_repository** – A repository whose info() will be used to translate keys: `info(job) == translated_job`.
- **allow_deletes** – Whether the delete operation on this repository does anything. If enabled, it will delete only from the base repository.
- **prefetch_lookup** – Whether to cache the entirety of the translator repository in memory to improve performance.

async **contains**(*item*)

async **delete**(*job*)

async **info**(*job*)

async **unfiltered_iter**()

2.1.7 pydatatask.resource_manager module

pydatatask defines the notion of resources, or numerical quantities of CPU and memory which can be allocated to a given job. This is mediated through a *ResourceManager*, an object which can atomically track increments and decrements from a quota, and reject a request if it would break the quota.

Typical usage is to construct a *ResourceManager* and pass it to a task constructor:

```
quota = pydatatask.ResourceManager(pydatatask.Resources.parse(cpu='1000m', mem='1Gi'))
task = pydatatask.ProcessTask("my_task", localhost, quota, ...)
```

class pydatatask.resource_manager.**ResourceType**(*value*)

Bases: Enum

An enum class indicating a type of resource. Presently can be CPU, MEM, or RATE.

CPU = 1

MEM = 2

RATE = 3

class pydatatask.resource_manager.**Resources**(*cpu: Decimal = Decimal('0')*, *mem: Decimal = Decimal('0')*, *launches: int = 1*)

Bases: object

A dataclass containing a quantity of resources.

Resources can be summed:

```
r = pydatatask.Resources.parse(1, 1, 100)
r += pydatatask.Resources.parse(2, 3, 0)
r -= pydatatask.Resources.parse(1, 1, 0)
assert r == pydatatask.Resources.parse(2, 3, 100)
```

cpu: Decimal = Decimal('0')

mem: Decimal = Decimal('0')

launches: int = 1

static parse(*cpu: str | float | int | Decimal*, *mem: str | float | int | Decimal*, *launches: str | float | int | Decimal | None = None*) → *Resources*

Construct a *Resources* instance by parsing the given quantities of CPU, memory, and launches.

excess(*limit:* Resources) → *ResourceType* | None

Determine if these resources are over a given limit.

Returns
The ResourceType of the first resource that is over-limit, or None if self is under-limit.

class pydatatask.resource_manager.**ResourceManager**(*quota:* Resources)

Bases: object

The ResourceManager tracks quotas of resources. Direct use of this class beyond construction will only be necessary if you are writing a custom Task class.

Parameters
quota – The resource limit that should be applied to the sum of all tasks using this manager.

register(*func: Callable[[], Awaitable[*Resources*]]*)

Register an async callback which will be used to load the current resource utilization on process start. The initial reported usage will be the sum of the result of all the registered callbacks.

If you are writing a Task class, you should call this in your constructor to save a reference to a method on your class which retrieves the current resource utilization by that task.

async flush()

Forget the cached amount of resources currently being used. Next call to reserve or relinquish will calculate usage anew.

async reserve(*request:* Resources) → *ResourceType* | None

Atomically reserve the given amount of resources and return None, or do nothing and return the limiting resource type if any resource would be over-quota.

async relinquish(*request:* Resources)

Atomically mark the given amount of resources as available.

pydatatask.resource_manager.**parse_quantity**(*quantity*)

Parse kubernetes canonical form quantity like 200Mi to a decimal number. Supported SI suffixes: base1024: Ki | Mi | Gi | Ti | Pi | Ei base1000: n | u | m | "" | k | M | G | T | P | E

See https://github.com/kubernetes/apimachinery/blob/master/pkg/api/resource/quantity.go

Input: quantity: string. kubernetes canonical form quantity

Returns: Decimal

Raises: ValueError on invalid or unknown input

2.1.8 pydatatask.session module

A session is a tool for managing multiple live resources. Async resource manager routines can be registered, and in their place will be left a callable which will return the live resource while the session is opened.

For example:

```
session = pydatatask.Session()

@session.resource
async def my_file_dst():
    with open('/tmp/my_file_dst', 'wb') as fp:
        yield fp

@session.resource
async def my_file_src():
    with open('/tmp/my_file_src', 'rb') as fp:
        yield fp

async with session:
    my_file_dst().write(my_file_src().read())
```

The advantage over using normal contextlib context managers is that this produces a function reference which can be passed into other locations in sync contexts with the promise that it will not be called until the session is opened.

If a session is passed to a pipeline, it will be opened and closed when the pipeline is opened and closed.

Sessions cannot be opened more than once. But this doesn't have to be the way! If you have a use case, complain in a GitHub issue, and I'll see what can be done.

class pydatatask.session.**Session**

Bases: object

The session class. See module docs for usage information.

resource(*manager: Callable[[], AsyncGenerator[T, None]]*) → Callable[[], T]

Decorator for resource managers. Should be called with an async function that will yield exactly one object, the live constructed resource, and then tear that resource down on completion.

async open()

Open the session, initializing all the resources.

This is automatically called when entering an `async with session:` block.

async close()

Close the session, tearing down all the resources.

This is automatically called when exiting an `async with session:` block.

2.1.9 pydatatask.task module

A Task is a unit of execution which can act on multiple repositories.

You define a task by instantiating a Task subclass and passing it to a Pipeline object.

Tasks are related to Repositories by Links. Links are created by `Task.link("my_link_name", my_repository, **disposition)`. The main disposition kwargs you'll want to use are `is_input` and `is_output`. See *Task.link* for more information.

For a shortcut for linking the output of one task as the input of another task, see *Task.plug*.

pydatatask.task.**STDOUT = <pydatatask.task._StderrIsStdout object>**

class pydatatask.task.**Link**(*repo:* Repository, *is_input:* *bool* = *False*, *is_output:* *bool* = *False*, *is_status:* *bool* = *False*, *inhibits_start:* *bool* = *False*, *required_for_start:* *bool* = *False*, *inhibits_output:* *bool* = *False*, *required_for_output:* *bool* = *False*)

Bases: `object`

The dataclass for holding linked repositories and their disposition metadata. Don't create these manually, instead use *Task.link*.

repo: *Repository*

is_input: `bool` = **False**

is_output: `bool` = **False**

is_status: `bool` = **False**

inhibits_start: `bool` = **False**

required_for_start: `bool` = **False**

inhibits_output: `bool` = **False**

required_for_output: `bool` = **False**

class pydatatask.task.**Task**(*name:* *str*, *ready:* Repository | *None* = *None*, *disabled:* *bool* = *False*)

Bases: `object`

The Task base class.

property ready

Return the repository whose job membership is used to determine whether a task instance should be launched. If an override is provided to the constructor, this is that, otherwise it is `AND(*requred_for_start, NOT(OR(*inhibits_start)))`.

link(*name: str*, *repo:* Repository, *is_input: bool = False*, *is_output: bool = False*, *is_status: bool = False*, *inhibits_start: bool = False*, *required_for_start: bool | None = None*, *inhibits_output: bool = False*, *required_for_output: bool | None = None*)

Create a link between this task and a repository.

Parameters

- **name** – The name of the link. Used only in the admin interface.
- **repo** – The repository to link.
- **is_input** – Whether this repository contains data which is consumed by the task. Default False.
- **is_output** – Whether this repository is populated by the task. Default False.
- **is_status** – Whether this task is populated with task-ephemeral data. Default False.
- **inhibits_start** – Whether membership in this repository should be used in the default `ready` repository to prevent jobs for being launched. Default False.
- **required_for_start** – Whether membership in this repository should be used in the default `ready` repository to allow jobs to be launched. If unspecified, defaults to `is_input`.
- **inhibits_output** – Whether this repository should become `inhibits_start` in tasks this task is plugged into. Default False.
- **required_for_output** – Whether this repository should become *required_for_start* in tasks this task is plugged into. If unspecified, defaults to `is_output`.

plug(*output:* Task, *output_links: Iterable[str] | None = None*, *meta: bool = True*, *translator:* Repository | *None = None*, *translate_allow_deletes=False*, *translate_prefetch_lookup=True*)

Link the output repositories from *output* as inputs to this task.

Parameters

- **output** – The task to plug into this one.
- **output_links** – Optional: An iterable allowlist of links to only use.
- **meta** – Whether to transfer repository inhibit- and required_for- dispositions. Default True.
- **translator** – Optional: A repository used to transform job identifiers. If this is provided, a job will be said to be present in the resulting linked repository if the source repository contains the key `await translator.info(job)`.
- **translate_allow_deletes** – If translator is provided, this controls whether attempts to delete from the translated repository will do anything. Default False.
- **translate_prefetch_lookup** – If translator is provided, this controls whether the translated repository will pre-fetch the list of translations on first access. Default True.

property input

A mapping from link name to repository for all links marked `is_input`.

property output

A mapping from link name to repository for all links marked `is_output`.

property status

A mapping from link name to repository for all links marked `is_status`.

property inhibits_start

A mapping from link name to repository for all links marked `inhibits_start`.

property required_for_start

A mapping from link name to repository for all links marked `required_for_start`.

property inhibits_output

A mapping from link name to repository for all links marked `inhibits_output`.

property required_for_output

A mapping from link name to repository for all links marked `required_for_output`.

async validate()

Raise an exception if for any reason the task is misconfigured. This is guaranteed to be called exactly once per pipeline, so it is safe to use for setup and initialization in an async context.

abstract async update() → bool

Part one of the pipeline maintenance loop. Override this to perform any maintenance operations on the set of live tasks. Typically, this entails reaping finished processes.

Returns True if literally anything interesting happened, or if there are any live tasks.

abstract async launch(*job*)

Launch a job. Override this to begin execution of the provided job. Error handling will be done for you.

class pydatatask.task.**KubeTask**(*name: str*, *podman: Callable[[], PodManager]*, *resources: ResourceManager*, *template: str | Path*, *logs: BlobRepository | None*, *done: MetadataRepository | None*, *window: timedelta = datetime.timedelta(seconds=60)*, *timeout: timedelta | None = None*, *env: Dict[str, Any] | None = None*, *ready: Repository | None = None*)

Bases: *Task*

A task which runs a kubernetes pod.

Will automatically link a *LiveKubeRepository* as "live" with `inhibits_start, inhibits_output, is_status`

Parameters

- **name** – The name of the task.
- **podman** – A callable returning a PodManager to use to connect to the cluster.
- **resources** – A ResourceManager instance. Tasks launched will contribute to its quota and be denied if they would break the quota.
- **template** – YAML markup for a pod manifest template, either as a string or a path to a file.
- **logs** – Optional: A BlobRepository to dump pod logs to on completion. Linked as "logs" with `inhibits_start, required_for_output, is_status`.
- **done** – A MetadataRepository in which to dump some information about the pod's lifetime and termination on completion. Linked as "done" with `inhibits_start, required_for_output, is_status`.
- **window** – Optional: How far back into the past to look in order to determine whether we have recently launched too many pods too quickly.
- **timeout** – Optional: When a pod is found to have been running continuously for this amount of time, it will be timed out and stopped. The method *handle_timeout* will be called in-process.

- **env** – Optional: Additional keys to add to the template environment.
- **ready** – Optional: A repository from which to read task-ready status.

It is highly recommended to provide one or more of `done` or `logs` so that at least one link is present with `inhibits_start`.

property podman: *PodManager*

The pod manager instance for this task. Will raise an error if the manager is provided by an unopened session.

async launch(*job*)

async delete(*pod: V1Pod*)

Kill a pod and relinquish its resources without marking the task as complete.

async update()

async handle_timeout(*pod: V1Pod*)

You may override this method in a subclass, and it will be called whenever a pod times out. You can use this method to e.g. scrape in-progress data out of the pod via an exec.

class pydatatask.task.**ProcessTask**(*name: str*, *manager: Callable[[], AbstractProcessManager]*, *resource_manager:* ResourceManager, *job_resources:* Resources, *pids:* MetadataRepository, *template: str*, *window: timedelta = datetime.timedelta(seconds=60)*, *environ: Dict[str, str] | None = None*, *done:* MetadataRepository | *None = None*, *stdin:* BlobRepository | *None = None*, *stdout:* BlobRepository | *None = None*, *stderr:* BlobRepository | *_StderrIsStdout* | *None = None*, *ready:* Repository | *None = None*)

Bases: *Task*

A task that runs a script. The interpreter is specified by the shebang, or the default shell if none present. The execution environment for the task is defined by the ProcessManager instance provided as an argument.

Parameters

- **name** – The name of this task.
- **manager** – A callable returnint the process manager with control over the target execution environment.
- **resource_manager** – A ResourceManager instance. Tasks launched will contribute to its quota and be denied if they would break the quota.
- **job_resources** – The amount of resources an individual job should contribute to the quota. Note that this is currently **not enforced** target-side, so jobs may actually take up more resources than assigned.
- **pids** – A metadata repository used to store the current live-status of processes. Will automatically be linked as "pids" with `is_status, inhibits_start, inhibits_output`.
- **template** – YAML markup for the template of a script to run, either as a string or a path to a file.
- **environ** – Additional environment variables to set on the target machine before running the task.
- **window** – How recently a process must have been launched in order to contribute to the process rate-limiting.

- **done** – Optional: A metadata repository in which to dump some information about the process's lifetime and termination on completion. Linked as "done" with `inhibits_start, required_for_output, is_status`.
- **stdin** – Optional: A blob repository from which to source the process' standard input. The content will be preloaded and transferred to the target environment, so the target does not need to be authenticated to this repository. Linked as "stdin" with `is_input`.
- **stdout** – Optional: A blob repository into which to dump the process' standard output. The content will be transferred from the target environment on completion, so the target does not need to be authenticated to this repository. Linked as "stdout" with `is_output`.
- **stderr** – Optional: A blob repository into which to dump the process' standard error, or the constant *pydatatask.task.STDOUT* to indicate that the stream should be interleaved with stdout. Otherwise, the content will be transferred from the target environment on completion, so the target does not need to be authenticated to this repository. Linked as "stderr" with `is_output`.
- **ready** – Optional: A repository from which to read task-ready status.

It is highly recommended to provide at least one of `done`, `stdout`, or `stderr`, so that at least one link is present with `inhibits_start`.

property manager: *AbstractProcessManager*

The process manager for this task. Will raise an error if the manager comes from a session which is closed.

property stderr: *BlobRepository* | None

The repository into which stderr will be dumped, or None if it will go to the null device.

property basedir: Path

The path in the target environment that will be used to store information about this task.

async update()

async launch(*job*)

class pydatatask.task.**InProcessSyncTask**(*name: str*, *done:* MetadataRepository, *ready:* Repository | *None* = *None*, *func: FunctionTaskProtocol* | *None* = *None*)

Bases: *Task*

A task which runs in-process. Typical usage of this task might look like the following:

```
@pydatatask.InProcessSyncTask("my_task", done_repo)
async def my_task(job: str, inp: pydatatask.MetadataRepository, out: pydatatask.
↪MetadataRepository):
    await out.dump(job, await inp.info(job))

my_task.link("inp", repo_input, is_input=True)
my_task.link("out", repo_output, is_output=True)
```

Parameters

- **name** – The name of this task.
- **done** – A metadata repository to store some information about a job's runtime and termination on completion.
- **ready** – Optional: A repository from which to read task-ready status.

- **func** – Optional: The async function to run as the task body, if you don't want to use this task as a decorator.

async validate()

async update()

async launch(*job*)

class pydatatask.task.**ExecutorTask**(*name: str*, *executor: Executor*, *done:* MetadataRepository, *ready:* Repository | *None* = *None*, *func: Callable* | *None* = *None*)

Bases: *Task*

A task which runs python functions in a `concurrent.futures.Executor`. This has not been tested on anything but the `concurrent.futures.ThreadPoolExecutor`, so beware!

See *InProcessSyncTask* for information on how to use instances of this class as decorators for their bodies.

It is expected that the executor will perform all necessary resource quota management.

Parameters

- **name** – The name of this task.
- **executor** – The executor to run jobs in.
- **done** – A metadata repository to store some information about a job's runtime and termination on completion.
- **ready** – Optional: A repository from which to read task-ready status.
- **func** – Optional: The async function to run as the task body, if you don't want to use this task as a decorator.

async update()

async validate()

async launch(*job*)

async cancel(*job*)

Stop the current job from running, or do nothing if it is not running.

class pydatatask.task.**KubeFunctionTask**(*name: str*, *podman: Callable[[], PodManager]*, *resources:* ResourceManager, *template: str* | *Path*, *logs:* BlobRepository | *None* = *None*, *kube_done:* MetadataRepository | *None* = *None*, *func_done:* MetadataRepository | *None* = *None*, *env: Dict[str, Any]* | *None* = *None*, *func: Callable* | *None* = *None*)

Bases: *KubeTask*

A task which runs a python function on a kubernetes cluster. Requires a pod template which will execute a python script calling *pydatatask.main.main*. This works by running `python3 main.py launch [task] [job] --sync`.

Sample usage:

```
@KubeFunctionTask(
    "my_task",
    podman,
    resman,
    '''
```

(continues on next page)

(continued from previous page)

```
        apiVersion: v1
        kind: Pod
        spec:
          containers:
            - name: leader
              image: "docker.example.com/my/image"
              command:
                - python3
                - {{argv0}}
                - launch
                - "{{task}}"
                - "{{job}}"
                - "--force"
                - "--sync"
              resources:
                requests:
                  cpu: 100m
                  memory: 256Mi
    ''',
    repo_logs,
    repo_done,
    repo_func_done,
)
async def my_task(job: str, inp: pydatatask.MetadataRepository, out: pydatatask.
↪MetadataRepository):
    await out.dump(job, await inp.info(job))

my_task.link("inp", repo_input, is_input=True)
my_task.link("out", repo_output, is_output=True)
```

Parameters

- **name** – The name of this task.
- **podman** – A callable returning a PodManager to use to connect to the cluster.
- **resources** – A ResourceManager instance. Tasks launched will contribute to its quota and be denied if they would break the quota.
- **template** – YAML markup for a pod manifest template that will run *pydatatask.main.main* as `python3 main.py launch [task] [job] --sync --force`, either as a string or a path to a file.
- **logs** – Optional: A BlobRepository to dump pod logs to on completion. Linked as "logs" with `inhibits_start, required_for_output, is_status`.
- **kube_done** – Optional: A MetadataRepository in which to dump some information about the pod's lifetime and termination on completion. Linked as "done" with `inhibits_start, required_for_output, is_status`.
- **func_done** – Optional: A MetadataRepository in which to dump some information about the function's lifetime and termination on completion. Linked as "func_done" with `inhibits_start, required_for_output, is_status`.
- **env** – Optional: Additional keys to add to the template environment.
- **ready** – Optional: A repository from which to read task-ready status.

- **func** – Optional: The async function to run as the task body, if you don't want to use this task as a decorator.

It is highly recommended to provide at least one of `kube_done`, `func_done`, or `logs`, so that at least one link is present with `inhibits_start`.

async validate()

async launch(*job*)

2.1.10 pydatatask.utils module

Various utility classes and functions that are used throughout the codebase but don't belong anywhere in particular.

class pydatatask.utils.**AReadStream**(**args*, ***kwargs*)

Bases: Protocol, AbstractAsyncContextManager

A protocol for reading data from an asynchronous stream.

async read(*n: int | None = None*) → bytes

Read and return up to `n` bytes, or if unspecified, the rest of the stream.

async close() → None

Close and release the stream.

class pydatatask.utils.**AWriteStream**(**args*, ***kwargs*)

Bases: Protocol, AbstractAsyncContextManager

A protocol for writing data to an asynchronous stream.

async write(*data: bytes*)

Write `data` to the stream.

async close() → None

Close and release the stream.

async pydatatask.utils.**async_copyfile**(*copyfrom: AReadStream*, *copyto: AWriteStream*, *blocksize=1048576*)

Stream data from `copyfrom` to `copyto`.

class pydatatask.utils.**AReadText**(*base: AReadStream*, *encoding: str = 'utf-8'*, *errors='strict'*, *chunksize=4096*)

Bases: object

An async version of `io.TextIOWrapper` which can only handle reading.

async read(*n: int | None = None*) → str

Read up to `n` chars from the string, or the rest of the stream if not provided.

async close()

Close and release the stream.

class pydatatask.utils.**AWriteText**(*base: AWriteStream*, *encoding='utf-8'*, *errors='strict'*)

Bases: object

An async version of `io.TextIOWrapper` which can only handle writing.

async write(*data: str*)

Write `data` to the stream.

async close()

Close and release the stream.

async pydatatask.utils.**async_copyfile_str**(*copyfrom: AReadText*, *copyto: AWriteText*, *blocksize=1048576*)

Stream text from `copyfrom` to `copyto`.

async pydatatask.utils.**roundrobin**(*iterables: List*)

An async version of the itertools roundrobin recipe.

roundrobin('ABC', 'D', 'EF') –> A D E B F C

CHAPTER

THREE

LINKS

- GitHub
- PyPI

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

I

K

L

M

O

P

Q

R

S