# knit Documentation

*Release 0.2.2*

**Continuum Analytics**

**Sep 15, 2017**

# Contents

1 Motivation 3

2 Scope 5

3 Related Work 7

Knit enables data scientists to quickly launch, monitor, and destroy distributed programs on a YARN cluster.

Many YARN applications are simple distributed shell commands – running a shell command on several nodes in a cluster. Knit enables users to express and deploy applications and software environments managed under YARN through Python.

Knit is part of the **'Dask'_** project, committed to bringing cluster-based data science within easy reach of python developers.

CHAPTER 1

# Motivation

Knit was built to support batch-oriented non-JVM applications. For example, Knit can deploy Python based distributed applications such as IPython Parallel, with particular support for **'Dask'_**. Knit was built with the following motivations in mind:

- **PyData Support** Bring the PyData stack into the Hadoop/YARN ecosystem
- **Easy Setup:** Support a minimal installation effort and the common cases with easy to use Python interface.
- **Deployable Runtimes:** Build and ship self contained environments along with the application. Knit uses conda to resolve library dependencies and deploy user libraries without IT infrastructure and management

CHAPTER 2

# Scope

Knit enables data scientists to quickly launch, monitor, and destroy simple distributed programs.

Knit is not a full featured YARN solution. Knit focuses on the common case in scientific workloads of starting a distributed process on many workers for a relatively short period of time. Knit does handle some dynamic container management but it is not suitable for running long-term infrastructural applications.

# CHAPTER 3

# Related Work

- Apache Slider: General purpose YARN application with a focus on long-running applications/services: HBase, Accumulo, etc.
- kitten: General purpose YARN application with Lua based configuration

See *the quickstart* to get started.

## 3.1 Installation

The runtime requirements of `knit` are python, lxml, requests, py4j. Python versions 2.7, 3.5 and 3.6 are currently supported. Dask is required to launch a Dask cluster. These are all available via conda (py4j on the conda-forge channel).

Testing depends on `pytest`.

### 3.1.1 Easy

Use `pip` or `conda` to install:

```
$ conda install knit -c conda-forge
or
$ pip install knit --upgrade
```

For dask clusters, you also need dask itself:

```
$ conda install dask distributed
```

### 3.1.2 Source

The following steps can be used to install and run `knit` from source.

Update and install system dependencies (e.g., for debian systems):

```
$ sudo apt-get update
$ sudo apt-get install git maven openjdk-7-jdk -y
```

or install these via conda

```
$ conda install -y -c conda-forge setuptools maven openjdk
```

Clone git repository and build maven project:

```
$ git clone https://github.com/dask/knit
$ cd knit
$ python setup.py install mvn
```

### 3.1.3 Testing on Docker

If you would like to test this package, but don't have a YARN cluster hanging around, you could make a small test one in your machine. This is essentially how the Continuous Integration tests work.

> $ export CONTAINER_ID=‘docker run -d mdurant/hadoop‘ $ docker exec -it $CONTAINER_ID bash # conda install dask distributed -y # conda install -c conda-forge lxml py4j knit # py.test -vv knit

## 3.2 Quickstart

### 3.2.1 Install

Use `pip` or `conda` to install:

```
$ pip install knit --upgrade
$ conda install knit -c conda-forge
```

### 3.2.2 Commands

#### Start

Instantiate `knit` with valid ResourceManager/Namenode IP/Ports and create a command string to run in all YARN containers

```
>>> from knit import Knit
>>> k = Knit(autodetect=True) # autodetect IP/Ports for YARN/HADOOP
>>> cmd = 'date'
>>> k.start(cmd)
'application_1454900586318_0004'
```

`start` also takes parameters: `num_containers`, `memory`, `virtual_cores`, `env`, and `files`

#### Status

After starting/submitting a command you can monitor its progress. The `status` method communicates with YARN's ResourceManager and returns a python dictionary with current monitoring data.

```
 >>> k.status()
{'allocatedMB': 512,
'allocatedVCores': 1,
'amContainerLogs': 'http://192.168.1.3:8042/node/containerlogs/container_
↪1454100653858_0011_01_000001/ubuntu',
'amHostHttpAddress': '192.168.1.3:8042',
'applicationTags': '',
'applicationType': 'YARN',
'clusterId': 1454100653858,
'diagnostics': '',
'elapsedTime': 123800,
'finalStatus': 'UNDEFINED',
'finishedTime': 0,
'id': 'application_1454100653858_0011',
'memorySeconds': 63247,
'name': 'knit',
'numAMContainerPreempted': 0,
'numNonAMContainerPreempted': 0,
'preemptedResourceMB': 0,
'preemptedResourceVCores': 0,
'progress': 0.0,
'queue': 'default',
'runningContainers': 1,
'startedTime': 1454276990907,
'state': 'ACCEPTED',
'trackingUI': 'UNASSIGNED',
'user': 'ubuntu',
'vcoreSeconds': 123}
```

Often we track the `state` of an application. Possible `states` include: `NEW`, `NEW_SAVING`, `SUBMITTED`, `ACCEPTED`, `RUNNING`, `FINISHED`, `FAILED`, `KILLED`

Further details on the current functioning of the cluster are available via the connected `yarn_api` class which can help with trouble shooting: `cluster_metrics()`, `nodes()`, `systems_logs`.

### Logs

We retrieve log data directly from a `RUNNING` Application Master:

```
>>> k.logs()
```

Or, if log aggregation is enabled, we retrieve the resulting aggregated log data stored in HDFS. Note: aggregated log data is only available **after** the application has finished or been terminated, usually with a small lag of a few seconds while log aggregation takes place.

### Kill

To stop an application from executing immediately, use the `kill` method:

```
>>> k.kill()
```

### 3.2.3 Python Applications

Python applications can be created by first making a conda environment for them to run within. This can be done directly with `CondaCreator` (and such environments are cached and reused) or with the `knit` instance itself.

A simple Python based application:

```
from knit import Knit
k = Knit()

env = k.create_env('test', packages=['python=3.5']])
cmd = 'python -c "import sys; print(sys.version_info); import random;␣
↪print(str(random.random()))"'
app_id = k.start(cmd, num_containers=2, env=env)
```

A long running Python application. Here we reuse the same environment create above:

```
from knit import Knit
k = Knit()

cmd = 'python -m SimpleHTTPServer'
app_id = k.start(cmd, num_containers=2, env=env)
```

### 3.2.4 Dask Cluster

Run a distributed dask cluster on YARN with a few lines like:

To start a dask cluster on YARN

```
from knit import dask_yarn
cluster = dask_yarn.DaskYARNCluster()
cluster.start(nworkers=4, memory=1024, cpus=2)
```

## 3.3 Usage

`Knit` can be used in several novel ways. Our primary concern is supporting easy deployment of distributed Python runtimes; though, we can also consider other languages (R, Julia, etc) should interest develop. Below are a few novels ways we can currently use `Knit`

### 3.3.1 Python

The example below use any Python found in the `$PATH`. This is usually the system Python (i.e., on a cluster where it has already been installed for you).

```
>>> import knit
>>> k = knit.Knit()
>>> cmd = "python -c 'import sys; print(sys.path); import socket; print(socket.
↪gethostname())'"
>>> appId = k.start(cmd)
```

### 3.3.2 Zipped Conda Envs

Often nodes managed under YARN may not have desired Python libraries or the Python binary at all! In these cases, we want to package up an environment to be shipped along with the command. `knit` allows us to declare a zipped directory with the following structure typical of Python environments:

```
$ ll dev/
drwxr-xr-x+ 23 ubuntu  ubuntu   782B Jan 30 17:55 bin
drwxr-xr-x+ 20 ubuntu  ubuntu   680B Jan 30 17:55 include
drwxr-xr-x+ 39 ubuntu  staff    1.3K Jan 30 17:55 lib
drwxr-xr-x+  4 ubuntu  staff    136B Jan 30 17:55 share
drwxr-xr-x+  6 ubuntu  ubuntu   204B Jan 30 17:55 ssl
```

```
>>> appId = k.start(cmd, env='<full-path>/dev.zip')
```

When we ship `<full-path>/dev.zip`, knit uploads `dev.zip` to a temporary directory within the user's home HDFS space e.g. `/users/ubuntu/.knitDeps` and the following bash ENVIRONMENT variables will be available:

- `$CONDA_PREFIX`: full path to prefix location of zipped directory
- `$PYTHON_BIN`: full path to Python binary

With the ENVIRONMENT variables available users can build more nuanced commands like the following:

```
>>> cmd = '$PYTHON_BIN $CONDA_PREFIX/bin/dask-worker 8786'
```

`knit` also provides a convenience method with `conda` to help build zipped environments. The following builds an environment `env.zip` with Python 3.5 and a variety of popular data Python libraries:

```
>>> env_zip = k.create_env(env_name='dev', packages=['python=3', 'distributed',
...                                                  'dask', 'pandas', 'scikit-learn
↪'])
```

### 3.3.3 Adding Files

`Knit` can also pass local files to each container.

```
>>> files = ['creds.txt', 'data.csv']
>>> k.start(cmd, files=files)
```

With the above, we are send files `creds.txt` and `data.csv` to each container and can reference them as local file paths in the `cmd` command.

### 3.3.4 Dask Clusters

The previous methods can be combined to launch a full distributed dask cluster on YARN with code like the following

```
from knit import DaskYARNCluster
cluster = DaskYARNCluster(env='my/conda/env.zip')
cluster.start(8, cpu=2, memory=2048)
```

The object `cluster` starts a dask scheduler, and can also be used to start or stop more containers than the original 8 referenced above. The same set of config options apply as for a `Knit` object, in addition to conda creation options, which will define the environment in which the workers run.

To start a dask client in the same session, you can simply do

```
from dask.distributed import Client
c = Client(cluster)
```

and use as usual, or look at `cluster.scheduler_address` for clients connecting from other sessions.

Note that DaskYARNCluster can also be used as a context manager, which will ensure that it gets closed (and the corresponding YARN application killed) when the `with` context finishes.

## 3.4 API

| | |
|---|---|
| `CondaCreator`([conda_root, conda_envs, ...]) | Create Conda Env |
| `CondaCreator.create_env`(env_name[, ...]) | Create zipped directory of a conda environment |
| `CondaCreator.zip_env`(env_path) | Zip env directory |

| | |
|---|---|
| `YARNAPI`(rm, rm_port[, scheme]) | |
| `YARNAPI.apps` | App IDs known to YARN |
| `YARNAPI.app_containers`([app_id, info]) | Get list of container information for given app. |
| `YARNAPI.logs`(app_id[, shell, retries, delay]) | Collect logs from RM (if running) |
| `YARNAPI.container_status`(container_id) | Ask the YARN shell about the given container |
| `YARNAPI.status`(app_id) | Get status of an application |
| `YARNAPI.kill_all`([knit_only]) | Kill a set of applications |
| `YARNAPI.kill`(app_id) | Method to kill a yarn application |

| | |
|---|---|
| *Knit*([autodetect, upload_always, hdfs_home, ...]) | Connection to HDFS/YARN. |
| *Knit.start*(cmd[, num_containers, ...]) | Method to start a yarn app with a distributed shell |
| *Knit.logs*([shell]) | Collect logs from RM (if running) |
| *Knit.status*() | Get status of an application |
| *Knit.kill*() | Method to kill a yarn application |
| *Knit.create_env*(env_name[, packages, ...]) | Create zipped directory of a conda environment |

**class** knit.core.**Knit**(*autodetect=True*, *upload_always=False*, *hdfs_home=None*, *knit_home='/home/docs/checkouts/readthedocs.org/user_builds/knit/checkouts/latest/knit/java_libs'*, *pars=None*, ***kwargs*)

Connection to HDFS/YARN. Launches a single "application" master with a number of worker containers.

Parameter definition (nn, nn_port, rm, rm_port): those parameters given to __init__ take priority. If autodetect=True, Knit will attempt to fill out the others from system configuration files; fallback values are provided if this fails.

**Parameters** **nn: str**

Namenode hostname/ip

**nn_port: int**

Namenode Port (default: 9000)

**rm: str**

Resource Manager hostname

**rm_port: int**

Resource Manager port (default: 8088)

**user: str ('root')**

The user name from point of view of HDFS. This is only used when checking for the existence of knit files on HDFS, since they are stored in the user's home directory.

**hdfs_home: str**

Explicit location of a writable directory in HDFS to store files. Defaults to the user 'home': hdfs://user/<username>/

**replication_factor: int (3)**

replication factor for files upload to HDFS (default: 3)

**autodetect: bool**

Autodetect configuration

**upload_always: bool(=False)**

If True, will upload conda environment zip always; otherwise will attempt to check for the file's existence in HDFS (using the hdfs3 library, if present) and not upload if that matches the existing local file in size and is newer.

**knit_home: str**

Location of knit's jar

**Note: for now, only one Knit instance can live in a single process because**

**of how py4j interfaces with the JVM.**

### Examples

```
>>> k = Knit()
>>> app_id = k.start('sleep 100', num_containers=5, memory=1024)
```

**add_containers**(*num_containers=1*, *virtual_cores=1*, *memory=128*)
Method to add containers to an already running yarn app

**num_containers: int** Number of containers YARN should request (default: 1) * A container should be requested with the number of cores it can

saturate, i.e.

- the average number of threads it expects to have runnable at a time.

**virtual_cores: int** Number of virtual cores per container (default: 1) * A node's capacity should be configured with virtual cores equal to * its number of physical cores.

**memory: int** Memory per container (default: 128) * The unit for memory is megabytes.

**check_env_needs_upload**(*env_path*)
Upload is needed if zip file does not exist in HDFS or is older

**static create_env**(*env_name*, *packages=None*, *remove=False*, *channels=None*, *conda_pars=None*)
Create zipped directory of a conda environment

**Parameters** **env_name** : str

**packages** : list

**conda_root: str**

Location of conda installation. If None, will download miniconda and produce an isolated environment.

**remove** : bool

remove possible conda environment before creating

**channels** : list of str

conda channels to use (defaults to your conda setup)

**conda_pars: dict**

Further pars to pass to CondaCreator

**Returns** path: str

path to zipped conda environment

### Examples

```
>>> k = Knit()
>>> pkg_path = k.create_env(env_name='dev',
...                         packages=['distributed', 'dask', 'pandas'])
```

**get_container_statuses**()
Get status info for each container

Returns dict where the values are the raw text output.

**get_containers**()
Method to return active containers

**Returns** container_list: List

List of dicts with each container's details

**hdfs**
An instance of HDFileSystem

Useful for checking on the contents of the staging directory. Will be automatically generated using this instance's configuration, but can instead directly set `self._hdfs` if necessary.

Note: if the namenode/port is not defined in the conf, will not attempt a connection, since it can take a while trying to connect to localhost:8020.

**kill**()
Method to kill a yarn application

**Returns** bool:

True if successful, False otherwise.

**list_envs**()
List knit conda environments already in HDFS

Looks staging directory for zip-files

**Returns: list of dict** Details for each zip-file.

**logs**(*shell=False*)
Collect logs from RM (if running) With shell=True, collect logs from HDFS after job completion

**Parameters** **shell: bool**

Shell out to yarn CLI (default False)

**Returns** log: dictionary

logs from each container (when possible)

**print_logs**(*shell=False*)
print out a more console-friendly version of logs()

**remove_containers**(*container_id*)
Method to remove containers from a running yarn app

Calls removeContainers in ApplicationMaster.scala

Be careful removing the ...0001 container. This is where the applicationMaster is running

**Parameters** **container_id: str**

**Returns** None

**runtime_status**()
Get runtime status of an application

**Returns** str:

status of application

**start**(*cmd*, *num_containers=1*, *virtual_cores=1*, *memory=128*, *env=''*, *files=[]*, *app_name='knit'*, *queue='default'*, *checks=True*)
Method to start a yarn app with a distributed shell

**Parameters** **cmd: str**

command to run in each yarn container

**num_containers: int**

Number of containers YARN should request (default: 1) * A container should be requested with the number of cores it can

saturate, i.e.

- the average number of threads it expects to have runnable at a time.

**virtual_cores: int**

Number of virtual cores per container (default: 1) * A node's capacity should be configured with virtual cores equal to * its number of physical cores.

**memory: int**

Memory per container (default: 128) * The unit for memory is megabytes.

**env: string**

Full Path to zipped Python environment

**files: list**

list of files to be include in each container

**app_name: String**

Application name shown in YARN (default: “knit”)

**queue: String**

RM Queue to use while scheduling (default: “default”)

**checks: bool=True**

Whether to run pre-flight checks before submitting app to YARN

**Returns** applicationId: str

A yarn application ID string

**status**()

Get status of an application

**Returns** log: dictionary

status of application

**wait_for_completion**(*timeout=10*)

Wait for completion of the yarn application

**Returns** bool:

True if successful, False otherwise

| |
|---|
| `DaskYARNCluster` |
| `DaskYARNCluster.start` |
| `DaskYARNCluster.stop` |
| `DaskYARNCluster.close` |
| `DaskYARNCluster.add_workers` |
| `DaskYARNCluster.remove_worker` |

## 3.5 Configuration

Several methods are available for configuring `Knit`.

The simplest is to load values from system `.xml` files. Knit will search typical locations and reads default configuration parameters from there. The file locations may also be specified with the environment variables `HADOOP_CONF_DIR`, which is the directory containing the XLM files, `HADOOP_INSTALL`, in which case the files are expected in subdirectory `hadoop/conf/`.

It is also possible to pass parameters when instantiating `Knit` or `DaskYARNCluster`. You can either provide individual common overrides (e.g., `rm='myhost'`) or provide a whole configuration as a dictionary (`pars={}`) with the same key names as typically contained in the XML config files. These parameters will take precedence over any loaded from files, or you can disable using the default configuration at all with `autodetect=False`.

### 3.5.1 Connection with hdfs3

Some operations, such as checking for uploaded conda environments, optionally make use of hdfs3. The configuration system, above, and that for hdfs3 are very similar, so you may well not have to make any extra steps to get this working correctly for you. However, you may well wish to be more explicit about the configuration of the HDFileSystem instance you want knit to use. In this case, create the instance as usual, and assign it to the Knit instance as follows

```
hdfs = HDFileSystem(...)
k = Knit(...)
k._hdfs = hdfs
```

or, similarly for a Dask cluster

```
cluster = DaskYARNCluster(...)
cluster.knit._hdfs = hdfs
```

The special environment variable `LIBHDFS3_CONF` will be automatically set when parsing the config files, if possible. Since the library is only loaded upon the first instantiation of a HDFileSystem, you still have the option to change its value in `os.environ`.

## 3.6 Examples

### 3.6.1 IPython Parallel

Install IPython Parallel and start IP Controller:

```
$ conda install ipyparallel
or
$ pip ipyparallel
$ ipcontroller --ip=*
```

IPController will create a file: `ipcontroller-engine.json` which contains metadata and security information needed by worker nodes to connect back to the controller. In a separate shell or terminal we use knit to ship a self-contained environment with `ipyparallel` (and other dependenices) and start `ipengine`

```
>>> from knit import Knit
>>> k = Knit(autodetect=True)
>>> env = k.create_env(env_name='ipyparallel', packages=['numpy', 'ipyparallel',
↪'python=3'])
>>> controller = '<HOMEDIR>/.ipython/profile_default/security/ipcontroller-engine.json
↪'
>>> cmd = '$PYTHON_BIN $CONDA_PREFIX/bin/ipengine --file=ipcontroller-engine.json'
>>> app_id = k.start(cmd, env=env, files=[controller], num_containers=3)
```

IPython Parallel is now running in 3 containers on our YARN managed cluster:

```
>>> from ipyparallel import Client
>>> c = Client()
>>> c.ids
[2, 3, 4]
```

# Index


## A

add_containers() (knit.core.Knit method), 13

## C

check_env_needs_upload() (knit.core.Knit method), 13
create_env() (knit.core.Knit static method), 13

## G

get_container_statuses() (knit.core.Knit method), 14
get_containers() (knit.core.Knit method), 14

## H

hdfs (knit.core.Knit attribute), 14

## K

kill() (knit.core.Knit method), 14
Knit (class in knit.core), 12

## L

list_envs() (knit.core.Knit method), 14
logs() (knit.core.Knit method), 14

## P

print_logs() (knit.core.Knit method), 15

## R

remove_containers() (knit.core.Knit method), 15
runtime_status() (knit.core.Knit method), 15

## S

start() (knit.core.Knit method), 15
status() (knit.core.Knit method), 16

## W

wait_for_completion() (knit.core.Knit method), 16