remotemanager

Louis Beal, BigDFT Team

Mar 21, 2025

CONTENTS:

`remotemanager` is a flexible and modular package dedicated to facilitating running python functions on remote systems.

This documentation will cover the functionality and utilities that are made available by this package. You may also wish to browse the source code, which can be found here.

See the introduction page covering the very basics:

CHAPTER

ONE

INTRO TO REMOTEMANAGER

`remotemanager` is the submission engine developed from the internal job submission system used within PyBigDFT.

The primary focus is helping you run massively parallel calculations on a remote machine, though features exist to allow a full workflow to be managed.

It is written entirely in Python, and implements a strongly object-oriented style. This leans into a high level of modularity, allowing componenents to be replaced or even used individually.

1.1 Core Features

When using `remotemanager`, there are two main objects the user will interact with, `Dataset` and `URL`.

1.1.1 Dataset

Dataset is a container object that is used to store information about your current calculation/workflow.

Calculations are described by defining a Python function, and `Dataset` stores this along with any additional data.

1.1.2 URL

The `URL` object (and derivatives) deal with connecting to the remote machine. These allow specification of all the important aspects of a machine from connection parameters to job submission specifics.

1.1.3 Structure

The basic overall structure of a Dataset is as shown below:

A basic overview of the structure of a `Dataset`. In this example, the function takes an input `name` and returns the string `hello {name}`. Runners can be added with the `append_run` method of `Dataset`, defining the input data.

Note

The `Database` within this example is not a "true" searchable database. It exists to checkpoint the current workflow, acting as a safeguard against data loss.

Warning

The database functionality enables the sending and receiving of Datasets. You should only run these from **people you trust.**

1.2 Requirements

There are some requirements to be aware of:

- A passwordless connection to the remote machine*
- python >= 3.7 on the local machine**
  - python >= 3.5 on the *remote* machine. (This will depend on the content of your Function, using features from higher python versions also increases this requirement)
- A `Linux` based operating system on the remote***

Note

* A passwordless connection can often by set up by way of ssh keys. This is further in the later section.

Note

** Python >=3.9 is recommended if you are using `sanzu`.

Note

*** Limited support exists for Windows, but only on the local machine.

1.3 License

`remotemanager` is open source and licensed under the MIT license. See the license page for more info.

1.4 Contributing

All contributions are welcome! If you spot a bug, issue or have a feature request don't hesitate to open an issue, or fork the main repo and submit a merge request!

1.5 Connecting to a Remote Machine

To run your functions on a machine other than your own, it is required that you are able to `ssh` into that machine without any further input.

For example, `ssh user@remote.address` should put you in a shell on that machine.

If you get a response asking for a password then the quickest solution to this is often to create and copy an ssh key over to the remote using ssh-copy-id.

To sum up the ssh system, these are the basic steps:

1. Create the ssh-key with `ssh-keygen`
2. Copy that key to your remote with `ssh-copy-id -i ~/.ssh/{key_name} user@remote.host`

However, if your remote system incorporates extra security (such as a password in addition to a key), sshpass can still allow functionality.

To do this, start with installing sshpass:

```
sudo apt install sshpass
```

Note

MacOS can raise issues when installing `sshpass`, citing security issues. There are mirrors which circumvent this block, you should search for a recent one if this is the case.

Then copy your password into a file with `400` permissions.

Warning

When connecting via sshpass it is advised to *not* input your password into the command directly as this can be captured in your bash history, exposing it. Steps should be taken to minimise the security issues such as storing the password in a "hidden", nondescript file such as `.file` under the proper permissions (`400`).

Now we have sshpass installed and our password file, we can connect by using:

```
sshpass -f <passwordfile> ssh user@remote.host
```

Added in version 0.3.7.

The inbuilt `URL` module has native support for `sshpass` files, simply give the `passfile` argument the *abspath* to your file:

```
url = URL(user = ..., host = ..., passfile = '~/passwordfile')
```

Added in version 0.5.7.

For an extra layer of separation, you can specify the path to your file within an environment variable with

```
export SSHPASSFILE='/path/to/.file
```

Then pass this to `passfile` with `os.environ['SSHPASSFILE']`

Alternatively, you can pass this variable directly to `URL` with `url=URL(..., envpass='SSHPASSFILE')`

Note

An explicit path passed to `passfile` will be prioritised over `envpass`

Similarly, for remotes who have a unique ssh key, the argument `keyfile` can be passed to point to that location. This will be added into the ssh call with the format `-i {keyfile}`

CHAPTER

TWO

QUICKSTART

This notebook will cover all the basic and most useful functionality available to get a user up and running as fast as possible.

2.1 Installation

Installation can be done via a pip install:

`pip install remotemanager` for the most recent stable version.

However if you would like the bleeding edge version, you can clone the `devel` branch of the git repository:

```
git clone --branch devel && pip install remotemanager
```

2.2 Function Definition

`remotemanager` executes user defined python functions at the location of choice. Below is a basic function example which will serve our purposes for this guide.

Important

The function must stand by itself when running, so any imports or necessary functionality should be contained within.

[1]:
```
def multiply(a, b):
    import time

    time.sleep(1)

    return a * b
```

2.3 Running Remotely

This function would run just fine on any workstation, but to run something more complex we would need to connect to some more powerful resources for this.

`remotemanager` provides the powerful `Computer` module for this purpose:

[2]:
```
from remotemanager import Computer
```

First, we must define a "template". This is the base from which a submission script will be generated.

The easiest way to create one of these templates, is to acquire a jobscript that you know works for your machine. A few suggestions for this:

- Machine documentation may have an example script to build from (or even a configurator!)
- If you have already run jobs, your own scripts should suffice, otherwise a colleague may have a example for you
- The helpdesk may be able to assist you in creating a jobscript for your use case

In this example, we will be taking an existing jobscript that we know works.

We will also parameterise just a single option, `#username#`. This syntax allows Computer to provide a "dynamic" input that can be changed.

The basic syntax for parameterisation is that anything between double `#hashes#` will be treated as a parameter and added to the computer. Here, for example, a variable called "hashes" would be created.

Important

Parameters will be sanitised to all lowercase. Therefore `#ARG#` == `#arg#`.

Important

Parameters must not clash with internal names, an error will be raised in this case. For example, we have to choose `#username#` here instead of `#user#`, since `user` is already an internal argument.

Note

This is covered in greater detail in the dedicated tutorial

[3]:
```
template = """#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=4
#SBATCH --time=00:30:00

#SBATCH --job-name=quickstart

#SBATCH --account=#username#
#SBATCH --partition=boost_usr_prod
#SBATCH --qos=normal

export OMP_NUM_THREADS=4

module load python/3.10.8--gcc--11.3.0
"""
```

Now, create a Computer. At a minimum you should specify:

- Host address (or `userhost=user@host`)
- The submitter that your job system uses. (defaults to `bash`, which will run on the login node)

[4]:
```
connection = Computer(
    user="user",
```

(continues on next page)

(continued from previous page)

```
    host='remote.hpc.url',
    submitter="sbatch",
    template=template
)

# note that template arguments must be specified after initialisation
connection.username = "myuser"
```

This example connection is pointed at an imaginary `user@remote.hpc.url`. However, this uses your `ssh` configuration, so you are able to connect to a machine in the same way that you would from a command line.

For example, if there existed a machine which you connected to with `ssh machine`, then you are able to create a computer using:

```
connection = Computer("machine")
```

Important

`Computer` requires that you are able to ssh into the remote machine without any additional prompts from the remote. For connection difficulties regarding permssions, see the relevant section of the introduction.

Tip

The connection parameters inherit those from your ssh config. So if you are able to `ssh <host>`, you can create a `Computer` with `Computer("<host>")`.

Tip

Before using `Computer` for the first time on a machine, any immediate problems can be discovered by testing a basic command. Start with a simple `ssh user@remote "ls"` and see what comes back. If the terminal returns a sensible output without prompting for a password, a `Computer` should function as expected.

Now we have a connection ready to go, we can see an example of the script that would be produced:

[6]:
```
print(connection.script())
```

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=4
#SBATCH --time=00:30:00

#SBATCH --job-name=quickstart

#SBATCH --account=myuser
#SBATCH --partition=boost_usr_prod
#SBATCH --qos=normal

export OMP_NUM_THREADS=4

# module load python/3.10.8--gcc--11.3.0
```

2.4 Remote Commands

With the concept of this remote `connection`, we can excecute commands and (more importantly) our function on this machine.

For commands, url provides a `cmd` method, which will execute any strings given

[7]:
```
connection.cmd('echo "this command is executed on the remote"')
```

[7]:
```
this command is executed on the remote
```

2.5 Running Functions

For function execution, we require a `Dataset`.

> **Note**
>
> Think a `Dataset` as a container for a *function*.

Like URL, this can be imported directly from `remotemanager`:

[8]:
```
from remotemanager import Dataset
```

To create a dataset, pass your function to the `Dataset` constructor.

> **Note**
>
> When passing a function to the dataset, do not call it within the assigment. For example, call `Dataset(function=multiply)` *not* `Dataset(function=multiply())`

Here we are additionally specifying the `local_dir` and the `remote_dir`, which tells the Dataset where to put all relevant files on the local and remote machines, respectively.

> **Note**
>
> We will use `skip=False` in the Dataset creation, otherwise the `Dataset` will see the dataset we previously created and import its data rather than create itself anew.

[9]:
```
ds = Dataset(function=multiply,
             url=connection,
             local_dir='temp_local',  # Location where files will be "staged", before␣
↪sending to the remote
             remote_dir='temp_remote', # Location on the remote server where the run␣
↪will be executed
             skip=False
            )
```

> **Important**
>
> This dataset has no runs, as it is just a container for the function `multiply`. For this, we must add runners.

2.6 Creating runs

To add runs, we use the `Dataset.append_run()` method. This will take the arguments in `dict` format, and store them for later.

You may do this in any way you see fit, the important part is to pass a dictionary which contains all ncessary arguments for the running of your function:

[10]:
```
runs = [[21, 2],
        [64, 8],
        [10, 7]]

for run in runs:

    a = run[0]
    b = run[1]

    arguments = {'a': a, 'b': b}

    ds.append_run(arguments=arguments)
```

```
appended run runner-0
appended run runner-1
appended run runner-2
```

2.7 Running and Retrieving your results

Now we have created a dataset and appended some runs, we can launch the calculations. This is done via the Dataset.run() method

Once the runs have completed, you can retrieve your results with `ds.fetch_results()`, and access them via `ds.results` once this is done

Important

`fetch_results()` does not *return* your results, but collects the files and stores them within the runners.

[11]:
```
ds.run()
```

```
Running Dataset
assessing run for runner dataset-62eb4971-runner-0... running
assessing run for runner dataset-62eb4971-runner-1... running
assessing run for runner dataset-62eb4971-runner-2... running
Transferring 8 Files... Done
```

2.7.1 Wait

Calculations can take time, we can add an optional `wait` call here to await the dataset completion.

The first number is the check `interval`, the second is the maximum wait time (set to `None` for an indefinite wait).

[12]:
```
ds.wait(1, 10)
```

Now the run has completed, we must fetch the results before they are made available:

[13]:
```
ds.fetch_results()
```

```
Fetching results
Transferring 6 Files... Done
```

Results have been fetched from the remote, now we can access them.

[14]:
```
print(ds.results)
```

```
[42, 512, 70]
```

[15]:
```
ds.errors
```

[15]:
```
[None, None, None]
```

With this, you have all of the basic tools available to run python functions on a remote machine. See the other tutorials for more advanced usage

> **Warning**
>
> Be aware that on MacOS, you may receive some errors when transferring data. This is most likely due to MacOS natively using an old `rsync` version (<3.0.0). More information is available on this page.

CHAPTER

THREE

FAQ

Here you can find a non-exhaustive list of frequently asked questions and their answers.

3.1 FAQ

3.1.1 1. Why do we need another workflow manager?

`remotemanager` is first and foremost for facilitating the running of exploratory workflows on HPC resources (submission engine, vs workflow tool). Unsurprisingly, there is a lot of overlap of features with many existing workflow managers. The intention of `remotemanager` is to provide simple building blocks with a minimal learning curve; ideally you should find it intuitive and easy to extend. This will ease the work of prototyping and debugging new workflows or processes. Later, you can run your production calculations using `remotemanager`, or port it to a more "heavy" system.

3.1.2 2. What protocols are used behind the scenes?

Internally, `remotemanager` relies on a few basic protocols. These should not require any further installs on either your machine or the remote cluster. Additionally, if you have HPC experience, you may already be familiar with most of them:

1. `SSH` is used as a baseline for all communication and commands
2. `rsync` is the default method for transferring files (though `scp` is available in the base package)
3. `json` is used to serialise objects by default, though `yaml`, `dill` and `jsonpickle` are available

In the name of reducing the barrier to entry, care has been taken to avoid relying on dependencies which require a complex initial setup.

3.1.3 3. What is the meaning of the dataset filenames?

When running jobs, the primary data transfer method is via files. These files are automatically named (which can be somewhat confusing at first); however, there is a strict naming regime in place. The most important structure to search for is the dataset UUID. Within filenames, this will take the form of 8 hexadecimal characters. For example, the database file for a dataset could look something like `dataset-5f3ea4bc.yaml`. The UUID is based on a hash of the function to be run remotely. The exception to this rule is if you specify a `name` when creating your `Dataset` - this will then take precedence over the UUID.

On the remote machine, you will find files with names like `dataset-5f3ea4bc-runner-1-jobscript.sh`. You can inspect these files when troubleshooting to verify jobscripts were built appropriately, arguments were transfered successfully, etc.

3.1.4 4. Where can I see an example on how to define a Computer?

Computer definition can either be extremely basic or complex, depending on the requirements of your machine and any extra features you wish to add. There are tutorials available within the docs. It is always worth checking if someone has already created a `Computer` for your connection: they are transferable via YAML, so you may be able to skip the work (or contribute!).

3.1.5 5. How does 2FA work?

2FA is a forefront talking point in the HPC world, and is becoming more and more common.

`remotemanager` uses ssh keys as a primary factor, and can also interface with secondary factors using the `sshpass` library. See the relevant section for more info.

3.1.6 6. How can I decorate a previously defined function?

Sometimes you may want to run a function remotely that was defined in a preexisting library. To transform it to Sanzu version, you can use the decorator's internal function like so:

[2]:
```
from remotemanager.decorators.sanzufunction import SanzuFunction

def f():
    return 0

# now retroactively apply the decorator
f = SanzuFunction(f)
```

3.1.7 7. How do I fix rsync errors when running jobs?

By default, `rsync` is used as the internal `Transport`. While this allows us to take advantage of some its features, it has two known issues:

1. MacOS users have an outdated rsync version (`2.6.9`) by default
2. `rsync` has issues when using`sshpass`

For MacOS, if you have the ability to install packages on your machine, you can update your rsync. This should be as simple as `brew install rsync`. Ensure that `rsync --version >= 3.0.0`

If this is not a viable solution, there are other transport utilities available. `scp`, for example. You can assign this to a dataset just as you would a `URL`:

[3]:
```
from remotemanager import Dataset, URL
from remotemanager.transport import scp

def f():
    return

url = URL("user@host")
trn = scp()

ds = Dataset(f, url=url, transport=trn)
```

> **Note**
>
> More information is available at this link.

3.1.8 8. How do I deal with serialisation errors?

When using `remotemanager`, you may encounter errors like this:

[4]:
```
from uuid import UUID

@SanzuFunction
def f(x):
    return x
f(UUID(int=16))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
File ~/remotemanager/remotemanager/dataset/runner.py:117, in Runner.__init__(self,␣
↪arguments, dbfile, parent, self_id, extra_files_send, extra_files_recv, verbose,␣
↪extra, **run_args)
    115 try:
    116     # check that the args can be sent via json
--> 117     self._args = json.loads(json.dumps(arguments))
    118     self._generate_uuid()

File /usr/local/lib/python3.12/json/__init__.py:231, in dumps(obj, skipkeys, ensure_
↪ascii, check_circular, allow_nan, cls, indent, separators, default, sort_keys, **kw)
    227 if (not skipkeys and ensure_ascii and
    228     check_circular and allow_nan and
    229     cls is None and indent is None and separators is None and
    230     default is None and not sort_keys and not kw):
--> 231     return _default_encoder.encode(obj)
    232 if cls is None:

File /usr/local/lib/python3.12/json/encoder.py:200, in JSONEncoder.encode(self, o)
    197 # This doesn't pass the iterator directly to ''.join() because the
    198 # exceptions aren't as detailed.  The list call should be roughly
    199 # equivalent to the PySequence_Fast that ''.join() would do.
--> 200 chunks = self.iterencode(o, _one_shot=True)
    201 if not isinstance(chunks, (list, tuple)):

File /usr/local/lib/python3.12/json/encoder.py:258, in JSONEncoder.iterencode(self, o,
↪ _one_shot)
    254     _iterencode = _make_iterencode(
    255         markers, self.default, _encoder, self.indent, floatstr,
    256         self.key_separator, self.item_separator, self.sort_keys,
    257         self.skipkeys, _one_shot)
--> 258 return _iterencode(o, 0)

File /usr/local/lib/python3.12/json/encoder.py:180, in JSONEncoder.default(self, o)
    162 """Implement this method in a subclass such that it returns
    163 a serializable object for ``o``, or calls the base implementation
    164 (to raise a ``TypeError``).
   (...)
    178
    179 """
--> 180 raise TypeError(f'Object of type {o.__class__.__name__} '
    181                 f'is not JSON serializable')

TypeError: Object of type UUID is not JSON serializable

During handling of the above exception, another exception occurred:
```

(continues on next page)

(continued from previous page)

```
TypeError                                 Traceback (most recent call last)
Cell In[4], line 6
      3 @SanzuFunction
      4 def f(x):
      5     return x
----> 6 f(UUID(int=16))

File ~/remotemanager/remotemanager/decorators/sanzufunction.py:39, in SanzuWrapper.__
↪call__(self, *args, **kwargs)
     35         raise ValueError(f"Got multiple values for arg {argname}")
     37     kwargs[argname] = arg
---> 39 runner = self._ds.append_run(kwargs, return_runner=True)
     40 runner.run()
     42 self._ds.wait(only_runner=runner)

File ~/remotemanager/remotemanager/dataset/dataset.py:729, in Dataset.append_run(self,
↪ args, arguments, name, extra_files_send, extra_files_recv, dependency_call,␣
↪verbose, quiet, skip, force, lazy, chain_run_args, extra, return_runner, **run_args)
    726 else:
    727     r_id = f"runner-{rnum}"
--> 729 tmp = Runner(
    730     arguments=args,
    731     dbfile=self.dbfile,
    732     parent=self,
    733     self_id=r_id,
    734     extra_files_send=extra_files_send,
    735     extra_files_recv=extra_files_recv,
    736     verbose=verbose,
    737     extra=extra,
    738     **run_args,
    739 )
    741 tmp.result_extension = self.serialiser.extension
    743 tmp = self.insert_runner(
    744     runner=tmp,
    745     skip=skip,
   (...)
    749     return_runner=return_runner,
    750 )

File ~/remotemanager/remotemanager/dataset/runner.py:129, in Runner.__init__(self,␣
↪arguments, dbfile, parent, self_id, extra_files_send, extra_files_recv, verbose,␣
↪extra, **run_args)
    126 if not os.path.isdir(self.parent.local_dir):
    127     os.makedirs(self.parent.local_dir)
--> 129 content = self.parent.serialiser.dumps(arguments)
    130 with open(lpath, self.serialiser.write_mode) as o:
    131     o.write(content)

File ~/remotemanager/remotemanager/serialisation/serialjson.py:13, in serialjson.
↪dumps(self, obj)
     11 def dumps(self, obj):
     12     obj = self.wrap_to_list(obj)
---> 13     return json.dumps(obj)

File /usr/local/lib/python3.12/json/__init__.py:231, in dumps(obj, skipkeys, ensure_
```

(continues on next page)

(continued from previous page)

```
↪ascii, check_circular, allow_nan, cls, indent, separators, default, sort_keys, **kw)
    226 # cached encoder
    227 if (not skipkeys and ensure_ascii and
    228     check_circular and allow_nan and
    229     cls is None and indent is None and separators is None and
    230     default is None and not sort_keys and not kw):
--> 231     return _default_encoder.encode(obj)
    232 if cls is None:
    233     cls = JSONEncoder

File /usr/local/lib/python3.12/json/encoder.py:200, in JSONEncoder.encode(self, o)
    196         return encode_basestring(o)
    197 # This doesn't pass the iterator directly to ''.join() because the
    198 # exceptions aren't as detailed.  The list call should be roughly
    199 # equivalent to the PySequence_Fast that ''.join() would do.
--> 200 chunks = self.iterencode(o, _one_shot=True)
    201 if not isinstance(chunks, (list, tuple)):
    202     chunks = list(chunks)

File /usr/local/lib/python3.12/json/encoder.py:258, in JSONEncoder.iterencode(self, o,
↪ _one_shot)
    253 else:
    254     _iterencode = _make_iterencode(
    255         markers, self.default, _encoder, self.indent, floatstr,
    256         self.key_separator, self.item_separator, self.sort_keys,
    257         self.skipkeys, _one_shot)
--> 258 return _iterencode(o, 0)

File /usr/local/lib/python3.12/json/encoder.py:180, in JSONEncoder.default(self, o)
    161 def default(self, o):
    162     """Implement this method in a subclass such that it returns
    163     a serializable object for ``o``, or calls the base implementation
    164     (to raise a ``TypeError``).
   (...)
    178
    179     """
--> 180     raise TypeError(f'Object of type {o.__class__.__name__} '
    181                     f'is not JSON serializable')

TypeError: Object of type UUID is not JSON serializable
```

`remotemanager` uses `JSON` as the default way to send data too and from the machine. Unfortunately, custom datatypes cannot be serialized this way. To this end, we provide the `SerialDill` or `JSONPickle` serialisers. You can swap to these schemes by importing them from `remotemanager.serialisation`, and adding them at the `Dataset` definition. See the relevant section.

Note: Re-defining your dataset this way will not resubmit any jobs on `run()`.

[5]:
```
from remotemanager.serialisation import serialjsonpickle
@SanzuFunction(serialiser=serialjsonpickle())
def g(x):
    return x
g(UUID(int=16))
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-f5ec0403-runner-0... running
```

(continues on next page)

(continued from previous page)

```
Transferring 6 Files... Done
Fetching results
Transferring 1 File... Done
```

[5]:
```
UUID('00000000-0000-0000-0000-000000000010')
```

3.1.9 9. My function didn't work, how do I see the error?

`remotemanager` attempts to handle errors in the same way as results. If something is raised on the remote side, it will be captured in the `errors` property of the dataset.

Note that as these behave like results, you may need to call `fetch_results()` before you can see your error.

3.1.10 10. My error is missing information, how do I get more?

Errors by default return only the last line of the actual error to increase readability. However, the last line of the traceback is not always enough, so checking the full string is wise. For this, the `full_error` property exists.

This property is attached to the `Runner` rather than the `Dataset`. The easiest way to access this is to use the `Dataset.failed` property, which will return a list of all failed runs.

[6]:
```
from remotemanager import Dataset

def f(x):
    # this function will raise an exception
    if x < 0:
        raise RuntimeError("pretend this error is much longer")
    else:
        return x

ds = Dataset(f, skip=False)
ds.append_run({"x": -1});
ds.append_run({"x": 1});
ds.run(); ds.wait(1, 10); ds.fetch_results()

print(ds.errors)
```

```
appended run runner-0
appended run runner-1
Running Dataset
assessing run for runner dataset-44c604f9-runner-0... running
assessing run for runner dataset-44c604f9-runner-1... running
Transferring 7 Files... Done
Fetching results
Transferring 1 File... Done
['RuntimeError: pretend this error is much longer', None]
```

[7]:
```
for runner in ds.failed:
    print(f"runner: {runner}")
    print("error:")
    print(runner.full_error)
```

```
runner: dataset-44c604f9-runner-0
error:
Traceback (most recent call last):
  File "/home/test/remotemanager/docs/source/temp_runner_remote/dataset-44c604f9-
```

(continues on next page)

(continued from previous page)

```
↪runner-0-run.py", line 20, in <module>
    result = repo.f(**kwargs)
             ^^^^^^^^^^^^^^^^
  File "/home/test/remotemanager/docs/source/temp_runner_remote/dataset-44c604f9-repo.
↪py", line 179, in f
    raise RuntimeError("pretend this error is much longer")
RuntimeError: pretend this error is much longer
```

3.1.11 11. I made a mistake, how do I start over?

When prototyping it is often easier to fail fast and start over! There are a few methods available to assist with this. The "manual" method is to go into the file system and delete the database file associated with your `Dataset`. This will look something like `dataset-{8 character UUID}.yaml`, unless you named your dataset. If your folder is complex, you can ask the dataset to do that for you with a `hard_reset()` call. This will attempt to delete the local database file, the runners and any associated results/errors.

3.1.12 12. Some of my Runners failed, but not others. What do I do?

When running large datasets, sometimes you can have some runners fail but not others. In this case, `Dataset` and `Run()` offer some tools. The simplest case is if the failures are because of a machine issue, or resource issue. We can simulate this by setting up a Dataset that will fail if a file is present.

[9]:
```
from remotemanager import Dataset

import os

def f(inp, t=0):
    import os

    fail_flag = f"fail_{inp}"
    if os.path.exists(fail_flag):
        raise ValueError(f"found file {fail_flag}, raise error")

    return inp

ds = Dataset(f, skip=False)

ds.append_run({"inp": 1})
ds.append_run({"inp": 2})

try:
    os.makedirs(ds.remote_dir)
except FileExistsError:
    pass

# make runner 2 Fail
with open(os.path.join(ds.remote_dir, "fail_2"), "w+") as o:
    o.write("")

ds.run()
ds.wait(1, 10)
ds.fetch_results()
ds.results
```

```
appended run runner-0
appended run runner-1
Running Dataset
assessing run for runner dataset-5c161123-runner-0... running
assessing run for runner dataset-5c161123-runner-1... running
Transferring 7 Files... Done
Fetching results
Transferring 1 File... Done
Warning! Found 1 error(s), also check the `errors` property!
```

[9]:
```
[1, RunnerFailedError('ValueError: found file fail_2, raise error')]
```

To rerun only failed runners, you can use `Dataset.retry_failed()`. This will look for runners that are marked as failed, and run only those ones.

[10]:
```
os.remove(os.path.join(ds.remote_dir, "fail_2"))

ds.retry_failed()
ds.wait(1, 10)
ds.fetch_results()
ds.results
```

```
Running Dataset
assessing run for runner dataset-5c161123-runner-1... force running
Transferring 5 Files... Done
Fetching results
Transferring 1 File... Done
```

[10]:
```
[1, 2]
```

Alternatively, if a runner fails thanks to its arguments, the best option is to delete that runner and add a new one with the proper args. See the dedicated Failure Tutorial for more info.

3.1.13 11. I updated to version 0.11.x and now my Computers don't import. What do I do?

Version 0.11.0 changed a lot of things with how Computers are defined. It is a simple process to update, and should preserve all configurations. See the section regarding this.

Or see the tutorials for usage help:

CHAPTER

FOUR

DATASET USAGE

4.1 The Dataset Object

`Dataset` is the primary class of the package. It is a general purpose "container" which stores your function, runs and their results/errors.

For the purposes of this tutorial, we will bring back our basic `multiply` function you may have seen in the quickstart guide. Though we will amend it such that the delay time is adjustable:

[2]:
```
def multiply(a, b, t=1):
    import time

    time.sleep(t)

    return a * b
```

We can now set up a `Dataset`. The only *required* argument is the function, though there are many other optional arguments, most of which we shall also cover in this tutorial. See the Dataset API documentation for full details.

Again for this tutorial we will be using a local url, this enables the functions to run anywhere and be tested.

> **Note**
>
> At a basic level , the `URL` is a connection to your machine, and can be swapped out at any time to change machines. In theory any function which runs on `URL('machine.a')` will also run just the same on `URL('machine.b')`.

[3]:
```
import time
from remotemanager import Dataset, URL

url = URL('localhost')

ds = Dataset(function=multiply,
             url=url,
             script='#!/bin/bash',
             submitter='bash',
             local_dir='temp_ds_staging',
             remote_dir='temp_ds_remote',
             name='tutorial_dataset',
             skip=False)
```

The arguments shown here are likely to be the ones used the most. So in short:

- `url`: The remote connection, if it is not given, a default `localhost` "connection" will be created for you.

- `local_dir`: This is the directory that will be used to "stage" your files before sending to the remote. Defaults to `temp_runner_local`.
- `remote_dir`: Remote directory where files will be sent to. Defaults to `temp_runner_remote`.
- `name`: Datasets can be named, which makes their files easier to locate. By default, and files will simply use the uuid of the dataset/runner to differentiate.
- `skip`: Contextual argument, if set to `False`, will disable the Dataset init "skip", forcing it to delete the existing database and start anew.

4.1.1 Extra Variables

If you wish to run on a machine which has a scheduler system, you can use the `script` variable to pass your jobscript. Though there are more advanced features in place to generate dynamic jobscripts, see the Scheduler Tutorial for more info.

You can also specify a `run_dir`, which will be an internal directory within `remote_dir`. By default this is not specified and runs will run within the `remote_dir`.

`dbfile` allows you to force the dataset to store its database within a specific filename, should you wish to keep track of this. Otherwise, it defaults to `{self.name}-{self.short_uuid}.yaml`.

4.1.2 Appending Runs

Before running your function you must append runs containing any arguments.

`Dataset.append_run()` allows for run creation, and at minimum requires a `dict` containing the required arguments for your function.

So in our case, a dictionary containing arguments for `a` and `b` are necessary for a run to begin. As `t` has a default value of 1, it is optional. The structure below will append 3 runs displaying this behaviour:

> **Note**
>
> This is also true for runs that take no arguments, simply call `append_run()`

[5]:
```
runs = [{'a': 10, 'b': 5},
        {'a': 5.7, 'b': 8.4},
        {'a': 4, 'b': 4, 't': 6}]

for run in runs:
    ds.append_run(args=run)
```

```
appended run runner-0
appended run runner-1
appended run runner-2
```

> **Note**
>
> There is also the alias `arguments` for `args`

Additonally, if you wish to run scripts within unique folders, you can specify a `run_dir` when appending runs. If this attribute is present, this folder will be created within the remote dir and the function will be run from within. You may need to adjust your scripts and additional files to suit this run behaviour.

The Runner object

Now we have a dataset which is able to be run and return our results. Before we do this, it is worth stepping through some useful debugging tools.

Firstly, how to query what runs you already have. This can be done by accessing the property `Dataset.runners`:

[6]:
```
ds.runners
```

[6]:
```
[tutorial_dataset-a2c088ba-runner-0,
 tutorial_dataset-a2c088ba-runner-1,
 tutorial_dataset-a2c088ba-runner-2]
```

There is also the `runner_dict` property, which returns the same information in dict(append id: runner) format

[7]:
```
ds.runner_dict
```

[7]:
```
{'runner-0': tutorial_dataset-a2c088ba-runner-0,
 'runner-1': tutorial_dataset-a2c088ba-runner-1,
 'runner-2': tutorial_dataset-a2c088ba-runner-2}
```

Lazy Append

Added in version 0.8.4.

If you have a lot of runners to append (especially ones with large arguments), the base `append_run` can begin to slow down drastically. For such situations, you can call a context manager to wrap your run appends.

Here we copy the dataset (so as not to add too much bloat to the tutorial), then add 10 more runs:

[9]:
```
import copy
example_ds = copy.deepcopy(ds)

with example_ds.lazy_append() as la:
    for i in range(10):
        la.append_run({'a': i, 'b': 0})

print(len(example_ds.runners))

del example_ds
```

```
Of 13 appends: 13 appended
See append_log for more info
13
```

There is also a `lazy` option which can be used, which does the same thing. However there is a requirement that once you are done appending runs, you must add a `finish_append()` call, which finalises the appends all at once as though they were called normally.

> **Warning**
>
> Omitting the `finish_append()` after using a lazy append will not raise an error, but can cause strange behaviour.

4.2 Running the Dataset

Running of the datasets is done via the `Dataset.run()` method. This gives you one final opportunity to override any run arguments, as it provides another `run_args` catch for extra keyword args.

> **Note**
>
> Be aware of the argument expansion limitation that exists with rsync versions below `3.0.0`. If you get errors during transfer, be sure to check `rsync --version` >= 3.

[10]:
```
ds.run()
```

```
Running Dataset
assessing run for runner tutorial_dataset-a2c088ba-runner-0... running
assessing run for runner tutorial_dataset-a2c088ba-runner-1... running
assessing run for runner tutorial_dataset-a2c088ba-runner-2... running
Transferring 8 Files... Done
```

If you're following along on your machine you may have noticed that this call completed instantly, yet our function has a `time.sleep` line in it. We would expect to have to wait 8s for this (1+1+6s delays).

This is because the dataset run defaults to be asynchronous, and as you can imagine, this can be updated by passing this as a `run_arg` wherever you wish.

4.2.1 Waiting for Completion

Calculations can take time. You have two (non exclusive) options for dealing with this:

- Leave the notebook for a while and rerun when you think the jobs have finished
- Use `wait`

Rerunning the notebook at any time will cause the inbuilt `skip` methods to kick in and make sure that any running or completed jobs are not resubmitted. This means that you can submit and leave the notebook. At rerun, and any `fetch_results` which failed before will grab the results this time.

> **Note**
>
> Rerunning the notebook works fine provided you have not specified `skip=False` of `force=True` anywhere.

You can also use the `wait` keyword. This is a one line wrapper for a block that looks similar to this:

```
interval = 2
timeout = 10

t0 = time.time()
while not ds.all_finished():
    time.sleep(interval)

    if time.time() - t0 > timeout:
        break
```

This periodically checks for completed runs every `interval` seconds. It is also a blocking call until `ds.all_finished` returns True, or more time than `timeout` has passed.

[11]:
```
ds.wait(interval=2, timeout=10)
```

The call here means to check every 2 seconds, and raise a timeout error after 10 total seconds have passed.

> **Note**
>
> By default, `wait` waits for *any* completion, including failures. You can restrict this to wait for a total success (timing out if there are failures) by passing `success_only=True`.

4.2.2 Asynchronous

Asynchronous behaviour also means that each runner is running simultaneously, this can put excess load on machines not designed for it, or simply may not be what you want for your workflow. To avoid this, we can use `asynchronous=False`

Additionally here, we must use the `force=True` keyword to ensure that the runs go through, as the previous runs are marked as `complete`. Be careful using this keyword in your workflows with long jobs, as if they are still running and complete before your more recent run, it wil cause the results to be "injected".

[13]:
```
ds.reset_runs(wipe=True)

t0 = time.perf_counter()

ds.run(asynchronous=False)

dt = time.perf_counter() - t0

# we expect that the synchronous run will take around 1+1+6=8s
expected_time = 8
# the test suite can take extra time here, need to leave ~2s of room
assert abs(dt - expected_time) < 2, f"run completed in {dt}s"

print(f"run completed in {dt:.2f}s")
```

```
Running Dataset
assessing run for runner tutorial_dataset-a2c088ba-runner-0... running
assessing run for runner tutorial_dataset-a2c088ba-runner-1... running
assessing run for runner tutorial_dataset-a2c088ba-runner-2... running
Transferring 8 Files... Done
run completed in 8.11s
```

While not particularly useful in a wide range of use cases, there may be a situation case where you want to *wait* for a short run to complete, and this also displays the amending of run variables nicely.

One final way you are able to set the run args is via the `set_run_arg` method

[14]:
```
ds.set_run_arg('asynchronous', True)
print(ds.run_args["asynchronous"])

ds.set_run_arg('new_option', 'value!')
print(ds.run_args["new_option"])
```

```
True
value!
```

4.3 Collecting Results

There are functions indended to be used after a run has been called, to interact with the run, or the results.

We shall cover:

- is_finished
- all_finished
- fetch_results
- results
- errors

4.3.1 Dataset.is_finished

This property will return a boolean list of the `is_finished` method of the runners. Runners are considered `finished` when they have either returned a result, or failed with an error.

4.3.2 Dataset.all_finished

This property returns the all() of `Dataset.is_finished`

To demonstrate these, we shall re-run and see what the state looks like at a few time intervals. But first, we must make sure that the results are not already present.

[15]:
```
print('wiping result files...')

# this function will clear any runner results and optionally wipe local files
ds.reset_runs(wipe=True)
```

```
wiping result files...
```

Lets add a run that will fail, to demonstrate how errors are handled

[16]:
```
# we can't multiply an int by None, so this should fail
ds.append_run({'a': 0, 'b': None})
```

```
appended run runner-3
```

[17]:
```
time.sleep(1)  # this short sleep prevents earlier runs getting in the way

print('calcs launched, waiting before checking completion')
ds.run(asynchronous=True)

time.sleep(2)

print('\nafter 2s, state is now:')
print(ds.is_finished)
print('all_finished:', ds.all_finished)

time.sleep(5)

print('\nafter 7s, state is now:')
print(ds.is_finished)
print('all_finished:', ds.all_finished)
```

```
calcs launched, waiting before checking completion
Running Dataset
assessing run for runner tutorial_dataset-a2c088ba-runner-0... running
assessing run for runner tutorial_dataset-a2c088ba-runner-1... running
assessing run for runner tutorial_dataset-a2c088ba-runner-2... running
assessing run for runner tutorial_dataset-a2c088ba-runner-3... running
Transferring 10 Files... Done

after 2s, state is now:
[True, True, False, True]
all_finished: False

after 7s, state is now:
[True, True, True, True]
all_finished: True
```

It may seem counter-intuitive that the runs are all completed at 7s, but if we recall that they were launched asynchronously by default, the whole run would take around 6s (our maximum delay time).

The remaining functions

Dataset.fetch_results()

This function will attempt to grab any results from files or function objects that are attached to the dataset, storing them in the `results` property

Dataset.results

This property allows optimised access to the results of the previous run. When `results` is queried, it also checks to see if there are any errors, and warns you if any are found.

Dataset.errors

Similar to `results`, this stores a list of the error content if available.

[18]:
```
ds.fetch_results()
```

```
Fetching results
Transferring 4 Files... Done
```

[19]:
```
ds.results
```

```
Warning! Found 1 error(s), also check the `errors` property!
```

[19]:
```
[50,
 47.88,
 16,
 RunnerFailedError('TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
↪')]
```

[20]:
```
ds.errors
```

[20]:
```
[None,
 None,
 None,
 "TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'"]
```

4.4 Further features

While we touched on the runner availability earlier, we skipped over a feature which may be helpful for debugging purposes. The `Runner` object has a `history` property which prints a {time: state} dict that contains information about all state changes the runner has experienced.

This runner has been run and rerun a few times now, so the history will be quite full. On a fresh Dataset, a flag will be set to wipe this history.

[21]:
```
ds.runners[0].history
```

[21]:
```
{'2024-07-26 15:03:30/0': 'created',
 '2024-07-26 15:03:30/1': 'staged',
 '2024-07-26 15:03:30/2': 'submit pending',
 '2024-07-26 15:03:30/3': 'submitted',
 '2024-07-26 15:03:30/4': 'started',
 '2024-07-26 15:03:31/0': 'completed',
 '2024-07-26 15:03:38/0': 'reset',
 '2024-07-26 15:03:38/1': 'staged',
 '2024-07-26 15:03:38/2': 'submit pending',
 '2024-07-26 15:03:38/3': 'submitted',
 '2024-07-26 15:03:38/4': 'started',
 '2024-07-26 15:03:39/0': 'completed',
 '2024-07-26 15:03:46/0': 'reset',
 '2024-07-26 15:03:47/0': 'staged',
 '2024-07-26 15:03:47/1': 'submit pending',
 '2024-07-26 15:03:47/2': 'submitted',
 '2024-07-26 15:03:47/3': 'started',
 '2024-07-26 15:03:48/0': 'completed',
 '2024-07-26 15:03:49/0': 'completed',
 '2024-07-26 15:03:54/0': 'satisfied'}
```

here you can see the state history for the first runner in the list, showing the three runs, the creation time of the resultfile on the remote, and the final completion state where the results were loaded back into the runner

If you just require a list of states (for example, checking if a runner has passed through a state), there is the property `Runner.status_list`

4.4.1 Swapping out the serialiser

This is now covered in more depth in the dedicated tutorial.

4.4.2 Access to the commands used to execute the runs

Once you have run a dataset, you can access the command used to execute the bash scripts. This can be useful for debugging purposes.

[22]:
```
print('raw command:', ds.run_cmd.sent)
print('returned stdout:', ds.run_cmd.stdout)
print('returned stderr:', ds.run_cmd.stderr)
```

```
raw command: cd temp_ds_remote && bash tutorial_dataset-a2c088ba-master.sh
returned stdout:
returned stderr:
```

4.4.3 Running a single runner

While it was mentioned previously that the runners themselves should ideally not be touched, and all interaction should be done via the `Dataset`, it *is* possible to run a single runner if necessary.

> **Warning**
>
> this process is inefficient and should only be used if absolutely required. It may be preferable to clear the results of the offending runner using `reset_runs()` and rerunning with `skip=True`

[23]:
```
# store what the current last submission time is
last_submitted_initial = ds.runners[0].last_submitted
```

[24]:
```
ds.reset_runs()  # clear results to demonstrate

ds.runners[0].run(asynchronous=False)
time.sleep(1)
ds.fetch_results()
```

```
Running Dataset
assessing run for runner tutorial_dataset-a2c088ba-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

[25]:
```
# get the new last submission time
last_submitted_after = ds.runners[0].last_submitted
```

This quick assertion makes sure that the runner that was resubmitted actually has a different submission time.

[26]:
```
assert last_submitted_initial != last_submitted_after
```

We can again here demonstrate the use of `check_all_runner_states`, as we have only run one, checking for full completion will return False. Obviously in this case, `all_finished` will do the job, but you can query here for any state, such as `submitted`.

[27]:
```
print(ds.check_all_runner_states('completed'))
```

```
False
```

CHAPTER

FIVE

RUN ARGS

5.1 meta arguments

Now you have a concept of the `Function`, `Dataset` and `Runners`, it's time to talk about `run_args`. These are "extra" arguments that go alongside your function, but do not interact directly with it.

This can create some confusing terminology, so lets be explicit. Whenever `args` are discussed, this is referring to the actual `Function` arguments, i.e. what the `Runner` is storing. `run_args` deal with things like the remote directory and resource requests.

5.2 Native Arguments

Here we will cover the `run_args` that are natively understood by a run.

While you can implement your own functionality (more on that later), the following arguments are common to all runs.

5.2.1 Skip & Force

This is a special "contextual" run arg. There are three situations where there args are relevant.

Dataset init

By default, when defining `Dataset(...)`, a search is done to see if a matching Dataset has already been created. If this is the case, the current creation will be "skipped", and the Dataset will instead be unpacked from the previous state.

Setting `skip=False` will ensure a new `Dataset` is created, deleting the old database in the process.

`force=True` is ignored here, only `skip` has any function.

> **Note**
>
> It is advised to use `Dataset(..., skip=False)` while testing, as it ensures consistent behaviour. Only once you care about the result should you drop this argument (or change it to `True`).

Run append

Any `Runner` that already exists cannot be added to a `Dataset`.

With `skip=False` runner will be appended anyway. This does not overwrite the existing runner, and allows for multiple copies of the same run.

`force=True` acts as an inverted alias of `skip`. i.e. `skip=False == force=True`

Run()

When running a `Dataset`, `is_finished` is called to get the states of any runners. Any that are already running or have completed will not be submitted.

`skip=False` allows runners which are already submitted to be resubmitted

In general `force=True` functions as an inverted alias of `skip`. However there is an additional keyword argument `force_ignores_success` which is required to resubmit runners considered as "succeeded". This is an extra safeguard against overwriting data.

Important

`force_ignores_success` is required for `skip=False`/`force=True` to function on runners which are considered to have succeeded. This is a runner which has successfully returned a result file.

5.2.2 Dirs

The most commonly set run_args are the `*_dir` family. These designate where your run files will end up and it is recommended to change these from defaults when doing a full run. `remotemanger` can create a lot of small files, which can make directory navigation cumbersome, even with proper segmentation.

local_dir

This directory is on *your* machine, and dictates where the runners will "stage" from. When running, files are first written to this directory then sent to the remote.

remote_dir

This directory is the main one on the *remote* machine, and is where all the main run files are copied to.

run_dir

This directory is not always used, it exists *within* the `remote_dir`, and is where the run will actually be executed.

Warning

Be careful using run_dir with runs where the file system needs to be interacted with. A good example is when sending extra files, you will need to access them using `../file`, for example.

5.2.3 Run modifiers

Asynchronous

True by default, ensures that runs are executed in parallel. Set to False to force a dataset to execute its runners one after another (only functions when `submitter="bash"`)

5.3 Argument Hierarchy

`run_args` can be set at multiple levels.

- `Dataset` - This is the "top level" storage, all runners inherit from this dictionary
- `Runner` - Runners can have their own "local" run_args, just for that run
- `Run/Temporary` - when running a Dataset, you can also pass arguments into the run. These are considered "temporary" arguments, and will be dropped after the run completes.

Lets demonstrate what this looks like, starting with the defaults:

[1]:
```
from remotemanager import Dataset

def function(inp):
    return inp

# skip=False will be used heavily throughout the tutorials
# it is recommended that you also do so when experimenting
ds = Dataset(function, skip=False)

ds.append_run({"inp": 1})
```

```
appended run runner-0
```

[2]:
```
print(ds.run_args)
```

```
{'skip': True, 'force': False, 'asynchronous': True, 'local_dir': 'temp_runner_local',
↪ 'remote_dir': 'temp_runner_remote'}
```

The defaults here mean:

- Runs will try to skip (if they already have results)
- Runs will not be forced
- Jobs will be run asynchronously
- The local staging directory is `temp_runner_local`
- The remote running directory is `temp_runner_remote`

By comparison, the `Runner` object will appear to have no run_args, since these are "overrides" that are set at the runner level.

[3]:
```
print(ds.runners[0].run_args)
```

```
{}
```

When running a job, these arguments are combined into a single dictionary.

This can be seen at `derived_run_args`:

[4]:
```
print(ds.runners[0].derived_run_args)
```

```
{'skip': True, 'force': False, 'asynchronous': True, 'local_dir': 'temp_runner_local',
↪ 'remote_dir': 'temp_runner_remote'}
```

5.4 Setting run_args

Now we know the default values, how do we change them?

Firstly, any argument passed to `Dataset`, `append_run`, or `run()` that is *not* part of those functions will be treated as a `run_arg`. However you can update them after initialisation.

There are multiple ways to update or set args. The most obvious way is to directly update the `run_arg` dictionaries, but there also functions that can do this more "explicity".

> **Note**
>
> These functions exist on both `Dataset` and `Runner`.

Lets start by demonstrating a direct method:

[5]:
```
ds.run_args["direct"] = True

for k, v in ds.run_args.items():
    print(k, v)
```

```
skip True
force False
asynchronous True
local_dir temp_runner_local
remote_dir temp_runner_remote
direct True
```

5.4.1 set_run_args

This function can take a list of keys and values, and set them. You can also pass a single (key, val) pair.

[6]:
```
ds.set_run_args(["a", "b", "c"], [1, 2, 3])

ds.set_run_args("d", 4)

for k, v in ds.run_args.items():
    print(k, v)
```

```
skip True
force False
asynchronous True
local_dir temp_runner_local
remote_dir temp_runner_remote
direct True
a 1
b 2
c 3
d 4
```

5.4.2 update_run_args

This function takes a dictionary of arguments and updates the inner run_args with it. Useful for setting a large set of arguments at once.

[7]:
```
ds.update_run_args({"a": 10, "b": 11, "c": 12, "d": 13})

for k, v in ds.run_args.items():
    print(k, v)
```

```
skip True
force False
asynchronous True
local_dir temp_runner_local
remote_dir temp_runner_remote
direct True
a 10
b 11
c 12
d 13
```

5.5 Custom run_args

Unhandled `run_args` will be ignored by a run. However if you are using a Computer that accepts arguments for its `script()` method, they can be used there.

The main use for this dynamic ability is for scheduler resources, and this is covered in depth within the Scheduler Tutorial.

5.6 Runner overrides

The `run_args` of `Runner` act as "local" overrides for whatever is set in `Dataset`.

We can demonstrate this by setting a value on the runner.

[8]:
```
ds.runners[0].run_args["d"] = "foo"

print("Dataset args:", ds.run_args.get("d", None))

print("Runner args:", ds.runners[0].run_args.get("d", None))

print("Derived args:", ds.runners[0].derived_run_args.get("d", None))
```

```
Dataset args: 13
Runner args: foo
Derived args: foo
```

At the `Dataset` level, the value of `d` is still 13. However, on the runner on which we override the value, it is now "foo".

Any *other* runners, will retain the `Dataset` level value.

[9]:
```
ds.append_run({"inp": 2})

print("Derived args:", ds.runners[1].derived_run_args.get("d", None))
```

```
appended run runner-1
Derived args: 13
```

5.7 Temporary Run() Args

As was mentioned previously, you can also pass the same args to the `Run()` call of a Dataset. While difficult to demonstrate here, you can verify it by setting a `remote_dir` to something and then updating that arg within `Run()`.

Args set this way are discarded after the run, and are considered "temporary" by the `Dataset`, whereas args set any other way are saved when the Dataset is.

CHAPTER

SIX

DATABASE AND BACKUP

6.1 Database Files

When using Datasets, you may have noticed that a file of the form `dataset-abcd1234.yaml` is created in the working directory where the dataset is being run. This is what's known as a `Database` file, and is where the `Dataset` stores info.

This functionality allows the `skip` options to function across notebook restarts.

However, the `Database` is more powerful than just a storage of important data, in fact it actually stores almost *all* data.

6.1.1 Dataset Permanence

It is because of this total info store that we can recreate a `Dataset` from a `Database` file at any time. This is in fact what happens when you restart your notebook with skip=True in the Dataset initialisation (the default).

But this allows us to use this functionality in reverse, "packing" a dataset into a file that can be transferred or backed up.

Lets create a dataset to play with:

[1]:
```
from remotemanager import Dataset

def function(a, b):
    return a * b

ds = Dataset(function, skip=False)
```

> **Note**
>
> It is important to note here the behaviour of the `skip` parameter. When a dataset is created, it will search for a `Database` file that matches the parameters given. If found, it will unpack itself from that file by default. If `skip=False`, it will instead delete that file and recreate itself in place.

[2]:
```
runs = [
    [1, 10],
    [7, 5],
    [12, 3]
]

for run in runs:
    ds.append_run({"a": run[0], "b": run[1]})
# run the dataset
```

(continues on next page)

(continued from previous page)

```
ds.run()
# wait for the completion, checking every 1 second, up to a maximum of 10 seconds
ds.wait(1, 10)
# collect the results
ds.fetch_results()
# check the results
ds.results
```

```
appended run runner-0
appended run runner-1
appended run runner-2
Running Dataset
assessing run for runner dataset-d8ecb370-runner-0... running
assessing run for runner dataset-d8ecb370-runner-1... running
assessing run for runner dataset-d8ecb370-runner-2... running
Transferring 8 Files... Done
Fetching results
Transferring 6 Files... Done
```

[2]:
```
[10, 35, 36]
```

Now we have a completed run, lets explore some situations where the `Database` helps us.

Notebook Restarts

The most common use of these files is done automatically for you if a notebook is killed and restarted. If a Dataset is created without `skip=False`, it will recreate itself if it can. Lets simulate a restart here by deleting the dataset:

[3]:
```
del ds
```

Now, the dataset no longer exists within the notebook, exactly as if we had killed the notebook and restarted. Lets recreate it as we are rerunning:

[4]:
```
ds = Dataset(function)

ds.results
```

[4]:
```
[10, 35, 36]
```

Since the dataset was recreated, it still contains everything necessary to continue as if it was never deleted. If we tried to run, it will skip, since the runs have already succeeded:

[5]:
```
ds.run()
```

```
Running Dataset
assessing run for runner dataset-d8ecb370-runner-0... ignoring run for successful␣
↪runner
assessing run for runner dataset-d8ecb370-runner-1... ignoring run for successful␣
↪runner
assessing run for runner dataset-d8ecb370-runner-2... ignoring run for successful␣
↪runner
```

In short, this means that the often intensive and long calculations are independent of the notebook. You do not risk resubmitting a large job if you accidentally close your notebook and rerun.

Notebook Transfers

Since the notebook and the database are just files, this also allows you to transfer your datasets to another machine or person. Simply copy across the notebook, along with the database, and `remotemanager` will attempt to run as if nothing has changed.

Important

Note that while the Dataset will attempt to run as normal, outside factors such as the python environment can still affect the runtime.

Important

If your Dataset requires (or creates) extra files that are needed for your workflow, be sure to have these at the same relative location to the new working directory. Later in this tutorial we will cover `Dataset.backup`, which automates more of this process for you.

6.2 Renaming the File

The automatically generated filename for the dataset can be complicated to remember. If you have multiple datasets running, even impossible to distinguish. It is possible to influence this file name in many different ways:

- Give the Dataset a name
- Set the `dbfile` parameter
- Pack to a custom file

Lets go through these now.

6.2.1 Naming the Dataset

Datasets can be given a `name` parameter, which makes their files easier to identify. First, lets take a look at the filename for the dataset created earlier.

[6]:
```
ds.dbfile
```

[6]:
```
'dataset-d8ecb370.yaml'
```

Not exactly memorable. Lets recreate with `skip=False` and give the new Dataset a name:

[7]:
```
ds = Dataset(function, name="functiontest", skip=False)

ds.dbfile
```

[7]:
```
'dataset-functiontest-291a69ef.yaml'
```

Now our name has been added to the filename, making it somewhat easier to find.

6.2.2 Specifying the filename

If you want to go one step further and customise the filename, the `dbfile` parameter that we've been checking can also be set at initialisation. This sets the filename that is used, so it can be whatever you want.

[8]:
```
ds = Dataset(function, dbfile="dataset_custom_filename", skip=False)

ds.dbfile
```

[8]:
```
'dataset_custom_filename.yaml'
```

> **Note**
>
> Since Databases are in yaml format, if you omit this ending from your dbfile, it will add it for you.

6.2.3 Packing to a Custom File

The functionalities that are used for the Database file are open for use by the user, and they do not always have to target the same file. You can pack and recreate from a file of your choosing, without touching the Database.

[9]:
```
ds.pack(file="temporary_dataset_pack")

import os
os.path.isfile("temporary_dataset_pack")
```

```
dumping payload to temporary_dataset_pack
```

[9]:
```
True
```

This method of storage does not enforce the yaml file extension, though the actual file content is still of the yaml format internally.

We can recreate from this file using `Dataset.from_file()`

Added in version 0.13.4: After the changes to how Computer is serialised, `from_file` now requires a `url` to be passed. Otherwise, a default (localhost) one will be created.

[10]:
```
del ds

ds = Dataset.from_file("temporary_dataset_pack")

ds.dbfile
```

[10]:
```
'dataset_custom_filename.yaml'
```

Note how the dbfile has not changed, as this pack/recreate is considered a "temporary" method of transfer.

6.3 Backup and Restore

Added in version 0.9.16.

It was mentioned earlier that there is a more advanced method for backup and restore than that which we have just covered. This system automatically handles returned files in addition to the dataset itself, so is more robust in the face of a Dataset which also uses files.

You should keep in mind that this method only handles *returned* files. This only includes:

```
- result
- error
- extra_files_recv
```

To demonstrate this, it is best to create a dataset that *does* return files:

[11]:
```
def to_file(inp, fname):
    with open(fname, "w+") as o:
        o.write(str(inp))

ds = Dataset(to_file, skip=False)

ds.append_run({"inp": "test", "fname": "test.out"}, extra_files_recv = "test.out")

ds.run()
ds.wait(1, 10)
ds.fetch_results()
ds.results
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-34bf7acd-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 3 Files... Done
```

[11]:
```
[None]
```

Our function here does not return anything, so the `results` property holds no data. The real information is within the file that is returned.

> **Note**
>
> It is considered good practice to have your functions returns something. This can make it much easier to fix problems. In this case, it would be wise to have the function return `fname` at the least, so we know that the function has completed as expected.

To access our "result", we should read the content of the returned file. The extra files are a special `TrackedFile` class which can help with this.

Lets get the runner that we want to see (index 0), then check its list of extra files to recieve. Since there's only one, we take the first index again, and print the `content` property of the `TrackedFile` that is there.

[12]:
```
print(ds.runners[0].extra_files_recv[0].content)
```

```
test
```

6.3.1 Limitations of the Database

Since the database only handles the properties of the Dataset and its runners directly, these extra files are only "tracked". So if they were to be deleted, moved, or renamed, the Dataset is essentially broken. If we delete the local file, even if we restore from a pack, the file contents will be gone:

[13]:
```
ds.pack(file="dataset_with_files_backup.yaml")

try:
    os.remove(ds.runners[0].extra_files_recv[0].local)
```

(continues on next page)

(continued from previous page)

```
except FileNotFoundError:
    print("could not remove file")

ds = Dataset.from_file(file="dataset_with_files_backup.yaml")

print(ds.runners[0].extra_files_recv[0].content)
```

```
dumping payload to dataset_with_files_backup.yaml
None
```

6.3.2 Backup and Restore

So in this situation, if we wanted to ensure the safety of our data, we should use the `backup` method. Lets fetch the results again to repopulate the files and demonstrate:

[14]:
```
ds.fetch_results()

print(ds.runners[0].extra_files_recv[0].content)
```

```
Fetching results
Transferring 2 Files... Done
test
```

Now, do as before, using backup and restore.

[15]:
```
ds.backup(file="full_backup.zip", full=True, force=True)

ds.hard_reset(files_only=True)

ds = Dataset.restore(file="full_backup.zip")

print(ds.runners[0].extra_files_recv[0].content)
```

```
test
```

Usage Details

There is a few things to note here, the first of which being that the filetype has to be `.zip`. If this is not the case, you'll get an error.

> **Note**
>
> This limitation is due to using the Python inbuilt ZipFile module.

Secondly, here we're using `force=True`. `backup` by default will not overwrite a file if it already exists, again raising an error. If you want to overwrite anyway, you can use `force=True` to overwrite the backup.

CHAPTER

SEVEN

FAILED RUNS

7.1 Dealing with Failures

In a large dataset, it's not impossible to imagine that some runners will fail to run for unforseen circumstances. Failures can occur at any point: In the shell, scheduler, or python, for example. Runners will still be marked as "satisfied", if that is the case, but the a summary of the error message will be available in `ds.errors`.

However it is not enough to know that your calculation *has* failed, so lets explore some tools that help you figure out *why* they failed.

We'll cover some common failure modes, so you can get a sense of what they look like:

- Function Error
- Argument Error
- Submission Error
- Walltime Issue

7.1.1 Function Error

There is a disconnect (however small) between writing and running the function. This can lead to small issues that ultimately cause the job to fail.

For this example, we'll simulate a broken function by attempting to access a variable that doesn't exist:

[2]:
```
import time
from remotemanager import Dataset

def multiply(a, b):

    foo

    return a * b

ds = Dataset(multiply, skip=False)

ds.append_run({'a': 2, 'b': 2})

ds.run()
ds.wait(1, 10)

ds.fetch_results()
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-e1c0c7cd-runner-0... running
```

(continues on next page)

(continued from previous page)

```
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

Run complete, lets see what happened:

[3]:
```
ds.results
```

```
Warning! Found 1 error(s), also check the `errors` property!
```

[3]:
```
[RunnerFailedError('NameError: name 'foo' is not defined')]
```

No results, and a warning saying that there is something in the `errors` property, lets check it.

[4]:
```
ds.errors
```

[4]:
```
["NameError: name 'foo' is not defined"]
```

Here's the error we were expecting.

Key indicators of failure are:

- An unexpected `None` result
- Content in the `errors` property

> **Note**
>
> It is possible to have a populated error file, but a sucessful run (some schedulers put warnings in stderr). This is why the message for this is only a warning. We will see this later in this tutorial.

7.1.2 Function Fixes

Since the identity of the dataset is tied heavily to the function, the only option for fixing the function is to create a new dataset.

If you already have submitted runs that you don't want to resubmit, however, you can copy them across to your new dataset, preserving their status. This is best done by `ds_new.copy_runners(ds)`.

Lets fix this function and rerun:

[5]:
```
def multiply(a, b):
    return a * b

ds_fixed = Dataset(multiply, skip=False)

ds_fixed.copy_runners(ds)

ds_fixed.runners
```

[5]:
```
[dataset-6fd64b82-runner-0]
```

Now we have our runner in our new dataset. This works because while the `Dataset` handles the function, a `Runner` only cares about the *arguments*. So as the function signatures match, this copy across will allow you to preserve your work.

> **Note**

> You can also select runners to insert, using `ds.insert_runner(runner)`. Internally `copy_runners` uses this function by looping over the `runners` property of the given dataset.

Note how since the runners are copied across unchanged, they retain their run state. So if we want to rerun, we must force:

[6]:
```
ds_fixed.run()
```

```
Running Dataset
assessing run for runner dataset-6fd64b82-runner-0... skipping already completed run
```

[7]:
```
ds_fixed.run(force=True)

ds_fixed.wait(1, 10)

ds_fixed.fetch_results()
```

```
Running Dataset
assessing run for runner dataset-6fd64b82-runner-0... force running
Transferring 4 Files... Done
Fetching results
Transferring 2 Files... Done
```

[8]:
```
ds_fixed.results
```

[8]:
```
[4]
```

7.2 Argument Error

When generating runs, sometimes the arguments themselves can be at fault. We can demonstrate this simply by adding a runner for the `multiply` function that has `None` as one of the args.

[10]:
```
ds = Dataset(multiply, skip=False)

ds.append_run({"a": 10, "b": 5})
ds.append_run({"a": 7, "b": None})

ds.run()
ds.wait(1, 10)

ds.fetch_results()

ds.results
```

```
appended run runner-0
appended run runner-1
Running Dataset
assessing run for runner dataset-6fd64b82-runner-0... running
assessing run for runner dataset-6fd64b82-runner-1... running
Transferring 6 Files... Done
Fetching results
Transferring 3 Files... Done
Warning! Found 1 error(s), also check the `errors` property!
```

[10]:
```
[50,
 RunnerFailedError('TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
↪')]
```

```
[11]: ds.errors
```

```
[11]: [None, "TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'"]
```

As expected, the 2nd runner failed.

7.2.1 Argument Fixes

Since the args themselves are at fault, and runners are responsible for holding the args, we should remove and replace this runner.

For this purpose, `Dataset` has a `remove_run` function:

```
[12]: ds.remove_run({"a": 7, "b": None})
```

```
removed runner dataset-6fd64b82-runner-1
```

```
[12]: True
```

For more information on running runners (and removing other bad data), see the Dataset Cleaning Tutorial

7.3 Submission Error - python

Supercomputers are often very specific about their environments and software. It's very easy to specify an incorrect module, python version or submitter. This is often solved within the `URL`, however the issue can arise from the `extra` in the dataset or runner. In any case, simply updating the incorrect line and resubmitting is often enough to resolve the issues.

Lets set `python` to something that doesn't exist to simulate this:

```
[14]: from remotemanager import URL

url = URL(python="foo")
ds = Dataset(multiply, url=url, skip=False)

ds.append_run({"a": 10, "b": 5}, extra="bar")
ds.append_run({"a": 7, "b": 15})

ds.run()
ds.wait(1, 10)

ds.fetch_results()

ds.results
```

```
appended run runner-0
appended run runner-1
Running Dataset
assessing run for runner dataset-6fd64b82-runner-0... running
assessing run for runner dataset-6fd64b82-runner-1... running
Transferring 6 Files... Done
Fetching results
Transferring 2 Files... Done
Warning! Found 2 error(s), also check the `errors` property!
```

```
[14]: [RunnerFailedError('dataset-6fd64b82-runner-0-jobscript.sh: line 6: foo: command not␣
↪found'),
 RunnerFailedError('dataset-6fd64b82-runner-1-jobscript.sh: line 5: foo: command not␣
↪found')]
```

```
[15]: ds.errors
```

```
[15]: ['dataset-6fd64b82-runner-0-jobscript.sh: line 6: foo: command not found',
       'dataset-6fd64b82-runner-1-jobscript.sh: line 5: foo: command not found']
```

7.3.1 Error Investigation

Now here we *know* that the `python` was set to an incorrect value, but this is not always the case, so the error would need more investigation.

First off, the `errors` property only shows us the last line of the error. While this can be enough, lets see if there's more to this particular error.

The `ds.failed` property returns a list of all runners that report `is_failed=True`. Runners also have a `full_error` property which will return the full contents of the error file for you:

```
[16]: print(ds.failed[0].full_error)
```

```
dataset-6fd64b82-runner-0-jobscript.sh: line 3: bar: command not found
dataset-6fd64b82-runner-0-jobscript.sh: line 6: foo: command not found
```

Hey look, here we can see the extra "bar" string that we set in the runner extra. But no extra information about our "foo" error.

Lets fix that by setting `python` to something sensible. Lets also leave the `bar` untouched for now, to see what happens:

```
[18]: ds.url.python = "python3"

      ds.run(force=True)
      ds.wait(1, 10)

      ds.fetch_results()

      ds.results
```

```
Running Dataset
assessing run for runner dataset-6fd64b82-runner-0... force running
assessing run for runner dataset-6fd64b82-runner-1... force running
Transferring 6 Files... Done
Fetching results
Transferring 4 Files... Done
Warning! Found 1 error(s), also check the `errors` property!
```

```
[18]: [50, 105]
```

Since we didn't update the `extra="bar"` line, we stil have an error there! This is important to display that just because there is an error, does not necessarily mean that the run has failed.

```
[19]: ds.errors
```

```
[19]: ['dataset-6fd64b82-runner-0-jobscript.sh: line 3: bar: command not found',
       None]
```

Extra Fixes

However this is something that can be removed, simply updating the `extra` to `None` will remove this error:

[21]:
```
ds.get_runner(0).extra = None

ds.run(force=True, force_ignores_success=True)
ds.wait(1, 10)

ds.fetch_results()

ds.results
```

```
Running Dataset
assessing run for runner dataset-6fd64b82-runner-0... force running
assessing run for runner dataset-6fd64b82-runner-1... force running
Transferring 6 Files... Done
Fetching results
Transferring 4 Files... Done
```

[21]:
```
[50, 105]
```

7.3.2 Submission Errors - `shell`

Submission of a run requires more than a simple `python` command, there are in fact two more similar arguments: `submitter`, which is put into the master script, and `shell`, which is used to launch the master script.

If you suspect that your `shell` might be broken, there is a very simple way to see what was submitted:

[23]:
```
url = URL(shell="foo")
ds = Dataset(multiply, url=url, skip=False)

ds.append_run({"a": 10, "b": 5}, extra="bar")
ds.append_run({"a": 7, "b": 15})

ds.run()

ds.wait(1, 5)

ds.fetch_results()

ds.results
```

```
appended run runner-0
appended run runner-1
Running Dataset
assessing run for runner dataset-6fd64b82-runner-0... running
assessing run for runner dataset-6fd64b82-runner-1... running
Transferring 6 Files... Done
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[23], line 9
      5 ds.append_run({"a": 7, "b": 15})
      7 ds.run()
----> 9 ds.wait(1, 5)
     11 ds.fetch_results()
     13 ds.results
```

(continues on next page)

(continued from previous page)

```
File ~/Work/Devel/remotemanager/remotemanager/dataset/dataset.py:2141, in Dataset.
↪wait(self, interval, timeout, watch, success_only, only_runner)
   2139 t0 = utcnow()
   2140 # check all non None states
-> 2141 while not wait_condition():
   2142     dt = utcnow() - t0
   2144     if watch:

File ~/Work/Devel/remotemanager/remotemanager/dataset/dataset.py:2113, in Dataset.
↪wait.<locals>.wait_condition()
   2112 def wait_condition():
-> 2113     states = self._is_finished()
   2115     if only_runner is not None:
   2116         return only_runner.is_finished

File ~/Work/Devel/remotemanager/remotemanager/dataset/dataset.py:2060, in Dataset._is_
↪finished(self, check_dependency, dependency_call, force)
   2058     self.run_cmd.communicate(ignore_errors=True)
   2059     if self.run_cmd.is_finished and not self.run_cmd.succeeded:
-> 2060         raise RuntimeError(f"Dataset encountered an issue:\n{self.run_cmd.
↪stderr}")
   2062 if check_dependency and not dependency_call and self.dependency is not None:
   2063     self.dependency.check_failure()

RuntimeError: Dataset encountered an issue:
/bin/bash: line 1: foo: command not found
```

Our wait timed out, which means no output files were produced, a surefire indicator of an error. If not even an error file was produced, it's very likely that the calculations were never submitted, something that's caused by a broken launch command. We can check this with the `run_cmd` attribute:

[24]:
```
ds.run_cmd.sent
```

[24]:
```
'cd temp_runner_remote && foo dataset-6fd64b82-master.sh'
```

There's a lot going on with this command, but all you really need to see here is the final section, where we can see our `foo`. This can be changed back to `bash` (or your preferred shell) via `url.shell`

7.4 Walltime Errors

Even if you make no mistakes on your end, it's still possible for a run to time out. Or run out of memory. Or any other scheduler related issue. The fixes in this case are similar to the previous example. Bump up the walltime request if needed and resubmit, done.

To demonstrate this we'll insert a string into the jobscripts that simulates a walltime issue, but also "hide" some "scheduler info" above.

[26]:
```
fake_walltime = '''
echo "{scheduler info}" >&2
echo out of walltime! >&2
exit 1'''

ds = Dataset(multiply, skip=False)

ds.append_run({"a": 10, "b": 5}, extra=fake_walltime)
ds.append_run({"a": 7, "b": 15})
```

(continues on next page)

(continued from previous page)

```
ds.run()
ds.wait(1, 10)

ds.fetch_results()

ds.results
```

```
appended run runner-0
appended run runner-1
Running Dataset
assessing run for runner dataset-6fd64b82-runner-0... running
assessing run for runner dataset-6fd64b82-runner-1... running
Transferring 6 Files... Done
Fetching results
Transferring 3 Files... Done
Warning! Found 1 error(s), also check the `errors` property!
```

[26]:
```
[RunnerFailedError('out of walltime!'), 105]
```

[27]:
```
ds.errors
```

[27]:
```
['out of walltime!', None]
```

There's our walltime line, perhaps the scheduler had more info for us?

[28]:
```
print(ds.failed[0].full_error)
```

```
{scheduler info}
out of walltime!
```

Seems it did, perhaps this content would give some advice useful for fixing your jobs (resource limits, etc).

Lets remove the walltime issue and resubmit. Here, we're just removing the `extra`, but in your case it may be on the URL side of things.

Since only one job actually failed, we really only want to rerun that one. You can use the `ds.failed` property to do this for you:

[29]:
```
for runner in ds.failed:
    runner.extra = None
    runner.run(force=True)
```

```
Running Dataset
assessing run for runner dataset-6fd64b82-runner-0... force running
Transferring 4 Files... Done
```

[30]:
```
ds.wait(1, 10)

ds.fetch_results()

ds.results
```

```
Fetching results
Transferring 2 Files... Done
```

[30]:
```
[50, 105]
```

7.5 Command Errors

In the background, `Dataset` is using the provided `URL` to issue commands on the remote machine. Sometimes, these can be the source of the failure.

The URL Tutorial has a section on error handling, but lets cover how to access these tools from the Dataset.

Each `Dataset` will have a `url` property, even if not set (one pointed at `localhost` will be created for you). This can be accessed at any time to change things or check for issues.

[31]:
```
ds.url.host
```

[31]:
```
'localhost'
```

Arguably the most useful debugging tool is the `cmd_history` property. This allows you to check the commands sent, up to the `cmd_history_depth` (defaults to 10).

We can write some quick debugging code to go through the history and find a specific command.

Lets say we think there was a problem with rsync, all we need to do is iterate back through the history and see what's there:

[32]:
```
transfer = None
for cmd in reversed(ds.url.cmd_history):
    if "rsync" in cmd.sent:
        transfer = cmd
        break

print(transfer.sent)
```

```
mkdir -p /home/ljbeal/Work/Devel/remotemanager/docs/source/tutorials/temp_runner_
↪local/ && rsync -auvh temp_runner_remote/{dataset-6fd64b82-runner-0-error.out,
↪dataset-6fd64b82-runner-0-result.json} /home/ljbeal/Work/Devel/remotemanager/docs/
↪source/tutorials/temp_runner_local/
```

This command was used to retrieve the log from the remote, you can also see what was returned by the command execution:

[34]:
```
print(transfer.stdout)
```

```
sending incremental file list
dataset-6fd64b82-runner-0-error.out
dataset-6fd64b82-runner-0-result.json

sent 202 bytes  received 54 bytes  512.00 bytes/sec
total size is 2  speedup is 0.01
```

And any stderr:

[35]:
```
print(transfer.stderr)
```

In this case we have none, but in the theoretical situation where the rsync has thrown errors, they will be printed in full here.

7.6 Combined Debugging

In many cases, your problem will require a mix of these tools and solutions. But with experience, hopefully you will find the data flow easy to follow. Some points to remember:

- An error in the output does not necessarily mean a failed run, it could just be a warning.
- Use the `failed` property in combination with the other runner-based tools to save having to search out the runners yourself.
- ‘Runner.full_error` is invaluable in finding hidden parts to your errors.
- Sometimes the `url` is at fault, check your `cmd_history`!
- Failing this, the `ds.run_cmd` will let you see if your run ever ran in the first place.

CHAPTER

EIGHT

SENDING AND RECEIVING FILES

Not all calculations can return their result via the remotemanager syntax. For this reason, `Dataset` also allows you to work with files, providing the `extra_files_send` and `extra_files_recv` hooks.

These perform as you would expect, a dataset which has some extra files to send will attempt to grab those and send them with each run. Likewise, if extra files are specified to be pulled, a `fetch_results()` call will attempt to fetch those files also.

Lets start with a function which merges two files to demonstrate this:

[2]:
```
from remotemanager import Dataset

def merge(fpath_a, fpath_b, fpath_c):
    with open(fpath_a, 'r') as o:
        data_a = o.read()

    with open(fpath_b, 'r') as o:
        data_b = o.read()

    with open(fpath_c, 'w+') as o:
        o.write(data_a + '\n' + data_b)

merge_files = Dataset(merge, skip=False)
```

Now we have our function, we need to create some files to send and merge:

[3]:
```
with open(f'temp_file_a.txt', 'w+') as o:
    o.write('hello, world!')

with open(f'temp_file_b.txt', 'w+') as o:
    o.write('add me to the output!')
```

[4]:
```
args = {'fpath_a': 'temp_file_a.txt',
        'fpath_b': 'temp_file_b.txt',
        'fpath_c': 'output.txt'}

merge_files.append_run(args = args,
                       extra_files_send = ['temp_file_a.txt', 'temp_file_b.txt'],
                       extra_files_recv = ['output.txt'])
```

```
appended run runner-0
```

Now run and collect our results:

[5]:
```
merge_files.run()
```

```
Running Dataset
assessing run for runner dataset-7b1e11ac-runner-0... running
Transferring 7 Files in 2 Transfers... Done
```

[6]:
```
merge_files.wait(1, timeout=10)
merge_files.fetch_results()
```

```
Fetching results
Transferring 2 Files... Done
```

Lets see what's in the results, and if the file has been returned as expected:

[7]:
```
print(merge_files.results)
```

```
[None]
```

[8]:
```
with open(f'{merge_files.local_dir}/output.txt', 'r') as o:
    print(o.read())
```

```
hello, world!
add me to the output!
```

Looks like it worked. Since the function itself does not return anything, we see `None` in the resutlts.

8.1 File Paths

When using this feature, it's important to pay attention to the locations of your files, as it's easy to get confused.

`extra_files_send` bases its locations on the current working directory from where the datset is run. When the `run()` command was issued for this example: `Dataset` will have looked within `os.getcwd()` for the files `temp_file_a.txt` and `temp_file_b.txt`. In short, it operates between `pwd` and the remote dir.

`extra_files_recv` is slightly different, operating between the `local_dir` and remote directory. This can be seen in the above example in that the `output.txt` is dropped into the `local_dir` rather than where the input files were sourced.

8.2 Fine Control

Added in version 0.12.3.

If the standard behaviour of files being sent between the working dir and remote dir aren't to your liking, there are other options.

While slightly more complex in terms of syntax, you also have the option of having fine control over your file locations.

8.2.1 Dict control

One way to do this is to specify your listings as dictionaries. This takes the form:

[9]:
```
extra_files_send = [{"local/path/to/file.txt": "path/to/target"}]
```

> **Note**
>
> It is assumed that the file name will be identical on the remote and local sides. If you need to change the name, you should do so within your Function.

In this case, it tells `remotemanager` that the extra file `file.txt` can be found in the directory `local/path/to/file/`, and that we want it to be sent to a directory named `path/to/target` relative to the dataset `remote_dir`.

Paths

Note that `remote_dir` is *relative* to the `Dataset.remote_dir` property, unless you specify an absolute path.

If we assume that we have a Dataset with the `remote_dir` set to `remote_run`, then we can send `file.txt` to `remote_run/inner_dir` using:

[10]:
```
extra_files_send = [{"file.txt": "inner_dir"}]
```

However:

[11]:
```
extra_files_send = [{"file.txt": "/home/user/run_data"}]
```

Would send `file.txt` to `/home/user/run_data`

8.2.2 Demonstration

We can demonstrate this with a simple function that reads the contents of a file.

[12]:
```
def read(file):
    with open(file) as o:
        return o.read()

def create_file(fname):
    with open(fname, "w+") as o:
        o.write("foo")
```

[13]:
```
ds = Dataset(read, name="read", skip=False)
```

The following setup is the same as the standard behaviour. We can print the intended remote directory of the extra file by accessing its `remote` property.

> **Note**
>
> Internally, your file specs are converted into a list of `TrackedFile` objects, so all the methods available to these can be used here.

[14]:
```
create_file("tmp_standard.txt")

ds.append_run({"file": "tmp_standard.txt"}, extra_files_send=[{"tmp_standard.txt": ""}
↪])
```

```
appended run runner-0
```

[15]:
```
print("Remote path for standard file:", ds.runners[0].extra_files_send[0].remote)  #␣
↪print the remote, for debugging
```

```
Remote path for standard file: temp_runner_remote/tmp_standard.txt
```

> **Note**
>
> To send to the remote_dir you can use the empty string "" or the "current dir" shortcut ".".

To send the file to a directory within the remote_dir, we can use this setup:

[16]:
```
create_file("tmp_dir.txt")

ds.append_run({"file": "inner_dir/tmp_dir.txt"}, extra_files_send=[{"tmp_dir.txt":
↪"inner_dir"}])
```

```
appended run runner-1
```

[17]:
```
print("Remote path for inner_dir file:", ds.runners[1].extra_files_send[0].remote)
```

```
Remote path for inner_dir file: temp_runner_remote/inner_dir/tmp_dir.txt
```

Otherwise, we can send the file to any arbitrary directory, if we know the abspath.

> **Note**
>
> Note that this will add a level of machine dependence to your run, as remotemanager expects that this path is valid and exsts.

[18]:
```
# create path using $HOME to allow testing
home = os.path.expandvars("$HOME")

path = os.path.join(home, "test")
file = os.path.join(path, "tmp_abs.txt")
```

[20]:
```
create_file("tmp_abs.txt")

ds.append_run({"file": file}, extra_files_send=[{"tmp_abs.txt": path}])
```

```
appended run runner-2
```

[21]:
```
print("Remote path for abspath file:", ds.runners[2].extra_files_send[0].remote)
```

```
Remote path for abspath file: /home/test/test/tmp_abs.txt
```

[22]:
```
ds.run()

ds.wait(1, 10)
```

```
Running Dataset
assessing run for runner read-cf7afc6e-runner-0... running
assessing run for runner read-cf7afc6e-runner-1... running
assessing run for runner read-cf7afc6e-runner-2... running
Transferring 12 Files in 4 Transfers... Done
```

If we collect the results we should see that all the files have been read in by the function.

[23]:
```
ds.fetch_results()
ds.results
```

```
Fetching results
Transferring 3 Files... Done
```

[23]:
```
['foo', 'foo', 'foo']
```

8.2.3 TrackedFile

Internally, all extra files are converted to `TrackedFile` instances. This allows the option for specifying these directly, if you prefer. Lets add an extra runner which displays this behaviour:

[24]:
```
from remotemanager.storage import TrackedFile

tfile = TrackedFile(".", ds.remote_dir, "trackedfile.txt")

# we can now use the write method of the TrackedFile class to add content to this file
tfile.write("foo, tracked")

ds.append_run({"file": tfile.name}, extra_files_send = [tfile])
```

```
appended run runner-3
```

Running this dataset again will run the new runner, showing the new file with its content:

[25]:
```
ds.run()
ds.wait(1, 10)
```

```
Running Dataset
assessing run for runner read-cf7afc6e-runner-0... ignoring run for successful runner
assessing run for runner read-cf7afc6e-runner-1... ignoring run for successful runner
assessing run for runner read-cf7afc6e-runner-2... ignoring run for successful runner
assessing run for runner read-cf7afc6e-runner-3... running
Transferring 6 Files in 2 Transfers... Done
```

[26]:
```
ds.fetch_results()
ds.results
```

```
Fetching results
Transferring 1 File... Done
```

[26]:
```
['foo', 'foo', 'foo', 'foo, tracked\n']
```

Important

The key points to setting up a `TrackedFile` are that the setup args are as follows: `TrackedFile(local_dir, remote_dir, filename)`. This sets up a file-like entity that provides the ability for remotemanager to "track" the behaviour between `local_dir` and `remote_dir`.

8.3 Retrieving Files

Using this methodology to collect files from your runs follows the same syntax. Keep in mind that the value of the dictionary is the *remote* specification, and the filename has to go in the *key*.

```
[27]: extra_files_recv = [{"local/path/to/file.txt": "remote_path"}]
```

This would fetch `"file.txt"` from `temp_runner_remote/remote_path/file.txt`, and move it to `local/path/to/file.txt`

> **Note**
>
> Just like with sending, you can also use abspaths here.

CHAPTER

NINE

SUBMITTING TO A SCHEDULER

9.1 Utilising a HPC

Until now we have been running things "directly", without using a scheduling system.

At best, this will run your calculations on the frontend/login nodes of the machine, without any large scale parallelisation.

9.2 Quick Links

In this section of the docs, we will cover how best to set up a Dataset to be able to launch jobs on the nodes of a machine.

The next tutorial covers the basics of jobscripts, and setting them "manually"

For information on templating, skip to the Template tutorial

For information on building a computer from a class, see the BaseComputer tutorial

If you already have a computer, and it was broken by an update, check here

Advanced URL usage can be found here

CHAPTER

TEN

TALKING TO THE SCHEDULER

10.1 Jobscripts

As mentioned before, we will need a jobscript if we are to access the scheduler system. This can be done in two ways:

1. Setting the `script` parameter of your `Dataset`
2. Defining your jobscript using a `BaseComputer` class

Lets start with the first option, setting a script manually.

10.2 Using a Direct Script

If you already know what should be in your jobscript, the "simplest" way to do this is to set the `script` attribute of a `Dataset` to a string.

Hardcoding the script this way is inflexible, but does work. Lets demonstrate with a basic jobscript:

[2]:
```
script = """
#SBATCH --ntasks-per-node=64
#SBATCH --cpus-per-task=4
#SBATCH --nodes=4
#SBATCH --queue=test
#SBATCH --account=myuser
#SBATCH --walltime=12:00:00
#SBATCH --exclusive

module load python
module load module/version"""
```

Script created, lets create a `Dataset` and `URL` for testing.

> **Note**
>
> You will need to manually set your `submitter` to your correct submitter. It should be listed in the documentation for your machine.

[3]:
```
from remotemanager import Dataset, URL

def f(inp):
    return inp

url = URL(submitter="sbatch")
```

(continues on next page)

(continued from previous page)

```
ds = Dataset(f, url=url, skip=False)

ds.script = script
```

[4]:
```
ds.append_run({"inp": True})

ds.run(dry_run=True)
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-9ebf1589-runner-0... running
launch command: cd temp_runner_remote && rm -f dataset-9ebf1589.manifest && sed -i -e
↪"s#{rootdir}#$(pwd)#" dataset-9ebf1589-repo.sh && source dataset-9ebf1589-repo.sh &&
↪ exec_and_log bash dataset-9ebf1589-master.sh
```

[5]:
```
print(ds.runners[0].jobscript.content)
```

```
#!/bin/bash

#SBATCH --ntasks-per-node=64
#SBATCH --cpus-per-task=4
#SBATCH --nodes=4
#SBATCH --queue=test
#SBATCH --account=myuser
#SBATCH --walltime=12:00:00
#SBATCH --exclusive

module load python
module load module/version

export DIR_3d536b34={run_rootdir}
source {run_rootdir}/dataset-9ebf1589-repo.sh

exec_and_log python dataset-9ebf1589-runner-0-run.py || write_to_log failed
```

The script is exactly what we set at the dataset level. This works as expected, assuming you want to submit your runners with the same script.

10.3 Using a Computer

A manual jobscript can be fine for quick testing purposes (or if it never changes). However for most cases, a more dynamic solution is required.

This is where the Computer comes in, the simplest way to describe these structures is as a "translation layer" between a common set of URL properties and whatever the scheduler is expecting.

The next tutorials will cover the usage of these more advanced objects.

CHAPTER

ELEVEN

PARAMETERISING JOBSCRIPTS

Now that we know how to set a script directly, we can at least run jobs on the nodes.

However this method is rather inflexible, and it would be nice to be able to parameterise these scripts in some way.

For that, we need to enable some level of parameterisation.

11.1 Templates

The simplest way of doing this is to use a template.

Assuming you have a jobscript already, you're most of the way there.

Lets take our script from earlier and create a template from it.

To do this, we just need to identify anything that we would want to change, and add a placeholder for a parameter.

11.1.1 Placeholders

These placeholders have a specific format. `remotemanager` will create value entries for you, by searching for anything of the form `#VALUE#`.

For example, for our `nodes` parameter, we should change the line

`#SBATCH --nodes=4`

to

`#SBATCH --nodes=#NODES#`

Lets do the whole script:

[2]:
```
jobscript_template = """#!/bin/bash

#SBATCH --ntasks-per-node=#TASKS_PER_NODE#
#SBATCH --cpus-per-task=#CPUS_PER_TASK#
#SBATCH --nodes=#NODES#
#SBATCH --queue=#QUEUE#
#SBATCH --account=#ACCOUNT#
#SBATCH --walltime=#TIME:format=time:default=3600#
#SBATCH --exclusive

#MODULES#"""
```

Now we have a parameterised jobscript, how do we use it?

This template can be used by a Computer (or Script) class to generate a script based on your inputs.

Lets go with Computer, since it's the more commonly used one:

[3]:
```
from remotemanager import Computer

conn = Computer(template=jobscript_template)

print(conn.script(tasks_per_node=16, cpus_per_task=4, nodes=12, queue="standard",␣
↪account="test"))
```

```
#!/bin/bash

#SBATCH --ntasks-per-node=16
#SBATCH --cpus-per-task=4
#SBATCH --nodes=12
#SBATCH --queue=standard
#SBATCH --account=test
#SBATCH --walltime=01:00:00
#SBATCH --exclusive
```

Note

Note that arguments are always lower case. Even if you specify uppercase in the actual parameterisation.

Note

Don't worry too much about this `script` function. This will be called for you when you're using a Dataset; you don't have to generate the scripts yourself.

Now that we have a script that uses parameters, we should cover the ways in which you can alter those params.

Generalisation

It's important to note that these parameters can be anything you want.

As an example of this, we're parameterising our module load with a `#MODULE#` parameter.

This means that we can dynamically change the modules by passing

`conn.modules = "module load ..."`

We will come back and use this template later in the tutorial. For now, we should cover the various ways you can control how these values behave.

11.2 Keyword Arguments

If we take a closer look at the `walltime` attribute, notice that `:format=time:default=3600` string? These are keyword args.

In this case, we are specifying that the output should be formatted as though it is a time string, with a default of 3600 (seconds).

Tip

Add keyword args just like you would in a normal python call, separated by a `:` character.

A complete list of kwargs is provided below:

11.2.1 default

This allows you to set the default value of a parameter. If not specified, and no value is given, then the `empty_treatment` behaviour applies (shown below)

[4]:
```
template = "a = #a:default=10#"

conn = Computer(template=template)

print(conn.script())
```

```
a = 10
```

value

You can *also* set the `value` directly. This is roughly equivalent to setting `default`, but has a higher priority. For example if you set `#PARAM:default="foo":value="bar"`, then "bar" will take priority, rendering the default essentially useless.

11.2.2 optional

Set to False to enforce that a value is present.

In the following example, we are allowed to give no value for `optional`, however we will get an error if we try to generate a script without specifying `required`

[5]:
```
template = """
optional = #OPT#
required = #REQ:optional=False#
"""

conn = Computer(template=template)

print(conn.script())
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[5], line 8
      1 template = """
      2 optional = #OPT#
      3 required = #REQ:optional=False#
      4 """
      6 conn = Computer(template=template)
----> 8 print(conn.script())

File ~/Work/Devel/remotemanager/remotemanager/script/script.py:319, in Script.
↪script(self, empty_treatment, **run_args)
    317 # check validity
    318 if not self.valid:
--> 319     raise ValueError(f"Missing values for parameters:\n{self.missing}")
    320 # generation section
    321 self._link_subs()  # ensure values are properly linked

ValueError: Missing values for parameters:
['req']
```

Optional Properties

You can see the required values at any moment by accessing the `required` property of your `Computer`.

Alternatively, the `missing` property shows what you still need to specify.

Finally, `valid` will be `True` only if there are no missing parameters.

[6]:
```
print("required:", conn.required)
print("missing:", conn.missing)
print("valid:", conn.valid)
```

```
required: ['req']
missing: ['req']
valid: False
```

11.2.3 requires

If a parameter requires another, you can specify that. In the following template, the `optional=True` is ignored, since `param` says that it requires it.

Added in version 0.13.3: You can now specify multiple requirements with a comma separated list: `requires=a,b,c`.

[7]:
```
template = """
param = #PARAM:default={foo}:requires=foo#

foo = #FOO:optional=True#
"""

conn = Computer(template=template)

print(conn.script())
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[7], line 9
      1 template = """
      2 param = #PARAM:default={foo}:requires=foo#
      3
      4 foo = #FOO:optional=True#
      5 """
      7 conn = Computer(template=template)
----> 9 print(conn.script())

File ~/Work/Devel/remotemanager/remotemanager/script/script.py:319, in Script.
↪script(self, empty_treatment, **run_args)
    317 # check validity
    318 if not self.valid:
--> 319     raise ValueError(f"Missing values for parameters:\n{self.missing}")
    320 # generation section
    321 self._link_subs()  # ensure values are properly linked

ValueError: Missing values for parameters:
['foo']
```

11.2.4 replaces

You can also mark a parameter as *replacing* another. This is less useful, but if you find a use case, it's there.

Here, you can specify only `param`, and `foo` will not complain.

Added in version 0.13.3: You can now specify multiple replacements with a comma separated list: `requires=a,b,c`.

[8]:
```
template = """
param = #PARAM:replaces=foo#

foo = #FOO:optional=False#
"""

conn = Computer(template=template)

print(conn.script(param="param"))
```

```
param = param
```

11.2.5 hidden

Hidden does what it says on the tin, allowing you to hide values but still use them elsewhere.

This is useful if you have configuration values or constants that are to be used in other parameters, but you don't want them printed in the script.

> **Warning**
>
> Hidden parameters will always use the `line` method of removal. They can (and will) clobber data that shares a line with them, so make sure that you space them out.

[9]:
```
template = """
val = #VAL:default={const*2}#

#CONST:value=10:hidden=True#
"""

print(Computer(template=template).script())
```

```
val = 20
```

11.2.6 min/max

Setting the `min` or `max` will raise an exception if the value steps out of these bounds (even if calculated)

Here, `b` returns 10x the value of `a`, so calling with `a=3` will result in a value of `b=30`.

As this is above our set limit, we will get an exception. The same applies to `min`, but in reverse.

[10]:
```
template = """
a = #a#
b = #b:max=20:default={a*10}#
"""

conn = Computer(template=template)

print(conn.script(a=3))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[10], line 8
      1 template = """
      2 a = #a#
      3 b = #b:max=20:default={a*10}#
      4 """
      6 conn = Computer(template=template)
----> 8 print(conn.script(a=3))

File ~/Work/Devel/remotemanager/remotemanager/script/script.py:327, in Script.
↪script(self, empty_treatment, **run_args)
    325     value = DELETION_FLAG_LINE
    326 else:
--> 327     value = sub.value  # get the value in string form
    328 if value is None or value == "None":
    329     # no value, triage this argument
    330     treatment = empty_treatment or sub.empty_treatment

File ~/Work/Devel/remotemanager/remotemanager/connection/computers/substitution.py:
↪172, in Substitution.value(self)
    166 @property
    167 def value(self) -> any:
    168     """
    169     Returns:
    170         value if present, else default
    171     """
--> 172     val = super().value
    174     if len(self.dependencies) == 0:
    175         return val

File ~/Work/Devel/remotemanager/remotemanager/connection/computers/dynamicvalue.py:
↪356, in DynamicMixin.value(self)
    352     pass
    354 val = self._format_value(val)
--> 356 self._validate(val)
    358 return val

File ~/Work/Devel/remotemanager/remotemanager/connection/computers/dynamicvalue.py:
↪441, in DynamicMixin._validate(self, value)
    437     raise ValueError(
    438         f"{value}{nameinsert} is less than minimum value {self.min}"
```

(continues on next page)

(continued from previous page)

```
    439     )
    440 if self.max is not None and value > self.max:
--> 441     raise ValueError(
    442         f"{value}{nameinsert} is more than maximum value {self.max}"
    443     )

ValueError: 30 for b is more than maximum value 20
```

11.2.7 format

Format allows you to enforce the format of the variable

The possible values are:

- `time`
- `float`

time

This will enforce a `HH:MM:SS` format for the input. By default it's expecting integer seconds, but also accepts a string input or "semantic time". For example:

`"01:00:00" == "1h" == 3600 -> 01:00:00`

`"24:00:00" == "1d" == 86400 -> 24:00:00`

Added in version 0.11.15.

Enabled the ability to set "semantic time" with `4h`, `1d`, etc.

float

Float will enforce a value to float.

By default all numerical inputs are passed through a `math.ceil` and converted to int.

This prevents two issues:

- Integers are generally preferred in jobscripts, requesting resources with floats may lead to unintended behaviour
- Small fractions will not resolve to `0`, requesting at least `1` of the resource in question

Lets say we have a calculation that requests nodes based on the total number of tasks and the number of cores that the nodes have. In this situation we can reasonably see that we could accidentally request 0 nodes, if we ask for less than a full node:

`TASKS/CORES_AVAILABLE = NODES`

Resolves to `64/128=0.5`

So instead of requesting `nodes=0.5` (or `nodes=0`), we convert this to `nodes=1`

However, if you *want* a float, you can set format=float, which will enforce that the value is printed as a float.

[11]:
```
template = """
time = #walltime:format=time#

num_flt = #num_flt:format=float#
num_int = #num_int#
```

(continues on next page)

(continued from previous page)

```
"""

conn = Computer(template=template)

conn.walltime = "24h"
conn.num_flt = 3.0
conn.num_int = 3.0

print(conn.script())
```

```
time = 24:00:00

num_flt = 3.0
num_int = 3
```

11.2.8 static

Sometimes you need to have {} in your output, however you may have found that this causes the contents to be evaluated. If that's the case, you can force that value to be "static" by passing that keyword.

[12]:
```
template = """
a = #a:default=10#

b = #b:default={a}#
c = #c:default={a}:static=True#
"""
print(Computer(template=template).script())
```

```
a = 10

b = 10
c = {a}
```

11.3 empty_treatment

This argument dictates how empty values are treated. It can be applied individually to the parameters, or globally to the `Computer`.

Since the default behaviour is to remove lines with empty values, you do not need to worry about *over* parameterising a jobscript. Provided you keep in mind the behaviours described below, we can envision a template that contains multiple times more content than any individual jobscript it may produce.

The possible values are:

- `line`
- `ignore`
- `local`

We can demonstrate this with a parameter `#NODES#`, and see what happens if we leave no value

11.3.1 line

This is the default behaviour, and removes the *whole line*.

[13]:
```
template = "#SBATCH nodes = #NODES:empty_treatment=line#"
print(Computer(template=template).script())
```

```

```

11.3.2 local

Local removes the actual arg itself, leaving the rest of the line.

Useful for when you have multiple parameters in the same line.

[14]:
```
template = "#SBATCH nodes = #NODES:empty_treatment=local#"
print(Computer(template=template).script())
```

```
#SBATCH nodes =
```

11.3.3 ignore

This behaviour intentionally does nothing, leaving everything untouched:

[15]:
```
template = "#SBATCH nodes = #NODES:empty_treatment=ignore#"
print(Computer(template=template).script())
```

```
#SBATCH nodes = #NODES:empty_treatment=ignore#
```

Here, we can see all the behaviours in one script:

[16]:
```
template = """
line = #line:empty_treatment=line#
local = #local:empty_treatment=local#
ignore = #ignore:empty_treatment=ignore#
"""

print(Computer(template=template).script())
```

```
local =
ignore = #ignore:empty_treatment=ignore#
```

11.4 Templating

Lets apply the initial script to a Computer and put it to use.

> **Note**
>
> Again, we need to set the `submitter` parameter.

[17]:
```
from remotemanager import Dataset

conn = Computer(template=jobscript_template, submitter="sbatch")
```

11.4.1 Arguments

Checking the `arguments` property, we can see a list of everything that the connection is expecting to be parameterised.

This is similar to the list you get when querying `required` and `missing`, and also has aliases at `args` and `subs`.

[18]:
```
conn.arguments
```

[18]:
```
['tasks_per_node',
 'cpus_per_task',
 'nodes',
 'queue',
 'account',
 'time',
 'modules']
```

Lets add this connection to a dataset and set the parameters that it's expecting.

> **Note**
>
> Parameters can be set at the URL (`Computer`), `Dataset`, `Runner`, or `run()` level.

[19]:
```
modules = """
module load python
module load module/version
"""

def f(inp):
    return inp

ds = Dataset(f,
             url=conn,
             account="myuser",
             modules=modules,
             skip=False)

ds.append_run({"inp": True}, mpi_per_node=64, omp=4, nodes=4)

ds.run(dry_run=True)
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-9ebf1589-runner-0... running
launch command: cd temp_runner_remote && bash dataset-9ebf1589-master.sh
```

[20]:
```
print(ds.runners[0].jobscript.content)
```

```
#!/bin/bash

#SBATCH --nodes=4
#SBATCH --account=myuser
#SBATCH --walltime=01:00:00
#SBATCH --exclusive


module load python
module load module/version
```

(continues on next page)

(continued from previous page)

```
export DIR_7f3744ae=7f3744ae_master
source $DIR_7f3744ae/dataset-9ebf1589-repo.sh
python dataset-9ebf1589-runner-0-run.py 2>> dataset-9ebf1589-runner-0-error.out
```

And there we have a sensible jobscript.

run_args

It was noted earlier, but it's worth covering in more detail. The actual arguments that are required are called "run args", usually acessible at the `run_args` property where relevant.

You can set these at multiple different levels, which each sucessive one overriding the previous.

Set location	Note
`Computer`	Top level setting, global, but overidden by everything.
`Dataset`	Global level defaults, will override any URL level params.
`Runner`	Specific to that runner, will override any `Dataset` level params.
`run()`	Global, but specific to that *run*. Overrides all parameters.

For more information on the specifics of run args and Datasets, see the Run Args Tutorial.

We can demonstrate this behaviour with a simple script:

[21]:
```
mini = Computer(template = "#TEST#")

mini.test = "conn level"

ds = Dataset(f, url=mini, skip=False, verbose=0)
ds.append_run({"inp": True})

ds.run(dry_run=True)

print(ds.runners[0].jobscript.content)
```

```
conn level
export DIR_3d536b34=3d536b34_master
source $DIR_3d536b34/dataset-9ebf1589-repo.sh
python dataset-9ebf1589-runner-0-run.py 2>> dataset-9ebf1589-runner-0-error.out
```

[22]:
```
ds.set_run_arg("test", "Dataset level")

ds.run(dry_run=True)

print(ds.runners[0].jobscript.content)
```

```
Dataset level
export DIR_3d536b34=3d536b34_master
source $DIR_3d536b34/dataset-9ebf1589-repo.sh
python dataset-9ebf1589-runner-0-run.py 2>> dataset-9ebf1589-runner-0-error.out
```

Here we see that the "Dataset level" setting takes priority.

Now lets demonstrate with multiple runners what happens there:

[23]:
```
ds.append_run({"inp": False}, test = "Runner level")

ds.run(dry_run=True)

print(ds.runners[0].jobscript.content)
```

```
Dataset level
export DIR_3d536b34=3d536b34_master
source $DIR_3d536b34/dataset-9ebf1589-repo.sh
python dataset-9ebf1589-runner-0-run.py 2>> dataset-9ebf1589-runner-0-error.out
```

[24]:
```
print(ds.runners[1].jobscript.content)
```

```
Runner level
export DIR_a61456f2=a61456f2_master
source $DIR_a61456f2/dataset-9ebf1589-repo.sh
python dataset-9ebf1589-runner-1-run.py 2>> dataset-9ebf1589-runner-1-error.out
```

Adding a second runner and setting a value for `test` there updates it in place, for that *Runner*.

However now setting it at the `run()` level will override even this:

[25]:
```
ds.run(dry_run=True, test="run() level")

print(ds.runners[1].jobscript.content)
```

```
run() level
export DIR_a61456f2=a61456f2_master
source $DIR_a61456f2/dataset-9ebf1589-repo.sh
python dataset-9ebf1589-runner-1-run.py 2>> dataset-9ebf1589-runner-1-error.out
```

11.5 Duplicate Arguments

The template extraction will function on the *first* argument found within a script, so any kwargs should be added there.

Added in version 0.11.15: BaseComputer will raise an exception if arguments are detected in any arguments but the first.

[26]:
```
template = """
#test:default=True#

#test:default="foo"#
"""

temp = Computer(template=template)

print(temp.test.value)  # note how the value is "True", not "foo"
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[26], line 7
      1 template = """
      2 #test:default=True#
      3
```

(continues on next page)

(continued from previous page)

```
      4 #test:default="foo"#
      5 """
----> 7 temp = Computer(template=template)
      9 print(temp.test.value)  # note how the value is "True", not "foo"

File ~/Work/Devel/remotemanager/remotemanager/connection/computers/computer.py:38, in␣
↪Computer.__init__(self, template, **kwargs)
     34 def __init__(self, template, **kwargs):
     35     # super() behaves strangely with multiple inheritance
     36     # explicitly call the __init__ with self
     37     URL.__init__(self, **kwargs)
---> 38     Script.__init__(self, template=template, **kwargs)

File ~/Work/Devel/remotemanager/remotemanager/script/script.py:80, in Script.__init__
↪(self, template, empty_treatment, **init_args)
     77 self._empty_treatment = None
     78 self.empty_treatment = empty_treatment
---> 80 self._extract_subs()

File ~/Work/Devel/remotemanager/remotemanager/script/script.py:148, in Script._
↪extract_subs(self)
    146 if name in self._subs:
    147     if kwargs is not None and kwargs != "":
--> 148         raise ValueError(
    149             f"Got more kwargs for already registered argument "
    150             f"{name}: {kwargs}"
    151         )
    152     logger.debug("\talready processed, continuing")
    153     continue

ValueError: Got more kwargs for already registered argument test: default="foo"
```

11.6 Escape Sequences

Added in version 0.13.5.

Templates reserve a small selection of characters as “control characters”. An example of this is splitting arguments using the `:` delimiter.

But what if we need to add one of these characters to our template? There are two ways to indicate that control sequences should be added directly.

11.6.1 Quotation

A simple method to do this is to simply quote the value. Strings inside quotations will not be parsed.

[27]:
```
template = """
ratio: #ratio:default="a:b"#
equal: #equal:default="a=b"#
"""

test = Computer(template=template)
print(test.script())
```

```
ratio: "a:b"
equal: "a=b"
```

11.7 Escaping with \

The backslash () character is a standard escape sequence, and the functionality has been extended to templates.

There exists two methods of adding escape sequences to templates:

- Specify the template as a "raw" string: `r"foo\:bar"`
- "Double-escape" the sequence: `"foo\\:bar`

Lets repeat the previous example, without quotes:

[28]:
```
template = r"""
ratio: #ratio:default=a\:b#
equal: #equal:default=a\=b#
"""

test = Computer(template=template)
print(test.script())
```

```
ratio: a:b
equal: a=b
```

[29]:
```
template = """
ratio: #ratio:default=a\\:b#
equal: #equal:default=a\\=b#
"""

test = Computer(template=template)
print(test.script())
```

```
ratio: a:b
equal: a=b
```

11.8 Further control

We have seen hints in this tutorial that values can be linked to one another. The next tutorial will cover this in greater detail, allowing you to get the full potential out of templating.

CHAPTER

TWELVE

DYNAMIC TEMPLATES

The previous tutorial covered how you can parameterise a template, allowing you to change variables on the fly.

While this works great for smaller scripts, there is more power to be leveraged here with "Dynamic" variables.

These allow you to "link" parameters together, so updating one value can make much larger changes to the script automatically.

12.1 Linked Parameters

Linking parameters together is simple to do.

When we covered `kwargs` in the previous tutorial, we also described `default` and `value`.

These allow you to specify python code within them, which will be evaluated.

The syntax used here is based on the python `f-string` syntax.

12.1.1 f-strings

f-strings are a feature in python that allows strings to be "evaluated" at runtime.

As a simple example, the string `f"value={foo}"` will evaluate using the value of `foo`. If we set `foo=True`, `print(f"value={foo}")` will return `value=True`.

> **Note**
>
> In short, anything within `{curly brackets}` will be evaluated as though it is python code.

> **Warning**
>
> Code within a dynamic value is executed via `eval` on your local machine, so you should be aware of the security implications of this. Make sure that you're aware of any code that may be running when you generate a jobscript!

Lets demonstrate this by setting up `nodes` to be dependent on `mpi`, `omp` and a new `cores_per_node` variable.

[2]:
```
from remotemanager import Computer
```

[3]:
```
template = """#!/bin/bash

#SBATCH --ntasks=#NTASKS#
#SBATCH --cpus-per-task=#CPUS_PER_TASK#
```

(continues on next page)

(continued from previous page)

```
#SBATCH --nodes=#NODES:default={ntasks*cpus_per_task/cores_per_node}#
#SBATCH --queue=#QUEUE#
#SBATCH --account=#ACCOUNT#
#SBATCH --walltime=#TIME:format=time:default=3600#
#SBATCH --exclusive

# using cores_per_node: #CORES_PER_NODE:default=128#

#MODULES#"""
```

[4]:
```
test = Computer(template=template)
```

[5]:
```
test.ntasks = 1024
test.cpus_per_task = 4

print(test.script())
```

```
#!/bin/bash

#SBATCH --ntasks=1024
#SBATCH --cpus-per-task=4
#SBATCH --nodes=32
#SBATCH --walltime=01:00:00
#SBATCH --exclusive

# using cores_per_node: 128
```

12.1.2 Extra Variables

All variables that are required within the script must be available within the Computer itself.

The easiest way of doing this is to add a “commented” line to the template.

For example, we had to add `cores_per_node` to the script, itself.

By doing this, you expose it as a value to the Computer, and add clarity for users.

12.1.3 Input Order

One thing to note with the previous example is that the order of your variables do not matter.

Since the values are evaluated when the script is generated, you can link values in any direction.

In this case, `nodes` is dependent on the `cores_per_node` value, which comes after it in the script.

12.1.4 Chaining Variables

Variables can also be "chained" into one another.

In this example, we are generating a default jobname that depends on the nodes. Nodes, in turn, depends on other variables.

[6]:
```
template = """#!/bin/bash

#SBATCH --ntasks=#NTASKS#
#SBATCH --cpus-per-task=#CPUS_PER_TASK#
#SBATCH --nodes=#NODES:default={ntasks*cpus_per_task/cores_per_node}#
#SBATCH --walltime=#TIME:format=time:default=3600#
#SBATCH --jobname=#JOBNAME:default=RUN_{ntasks}_{cpus_per_task}_{nodes}#
#SBATCH --exclusive

# using cores_per_node: #CORES_PER_NODE:default=128#

#MODULES#"""
```

[7]:
```
test = Computer(template=template)
```

[8]:
```
test.ntasks = 256
test.cpus_per_task = 4

print(test.script())
```

```
#!/bin/bash

#SBATCH --ntasks=256
#SBATCH --cpus-per-task=4
#SBATCH --nodes=8
#SBATCH --walltime=01:00:00
#SBATCH --jobname=RUN_256_4_8
#SBATCH --exclusive

# using cores_per_node: 128
```

12.2 Iterables

Added in version 0.13.5.

You are also able to specify limited iterables within your values, this can be useful for "selecting" arguments within jobscripts.

Similar to the quotation method of escaping control characters, values in `{evaluation}` blocks are also escaped. This allows the specification of dictionaries.

Lets set up a template that requests a larger `partition` if there are too many nodes.

[9]:
```
template = """#!/bin/bash

#SBATCH --ntasks=#NTASKS#
#SBATCH --cpus-per-task=#CPUS_PER_TASK#
#SBATCH --nodes=#NODES:default={ntasks*cpus_per_task/cores_per_node}#
#SBATCH --partition=#partition:default={partition_table[nodes<16]}#

# using cores_per_node: #CORES_PER_NODE:default=128#
#partition_table:default={{True: "small", False: "large"}}:hidden=True#
"""

test = Computer(template=template)
```

> **Note**
>
> We are making use of the `hidden` variable here to reduce clutter.

Now if we generate a script that only requests 2 nodes, we would expect the "small" partition to be requested:

[10]:
```
print(test.script(ntasks=64, cpus_per_task=4))
```

```
#!/bin/bash

#SBATCH --ntasks=64
#SBATCH --cpus-per-task=4
#SBATCH --nodes=2
#SBATCH --partition=small

# using cores_per_node: 128
```

Now, if we generate a much larger job, it should request the "large" partition automatically

[11]:
```
print(test.script(ntasks=1024, cpus_per_task=4))
```

```
#!/bin/bash

#SBATCH --ntasks=1024
#SBATCH --cpus-per-task=4
#SBATCH --nodes=32
#SBATCH --partition=large

# using cores_per_node: 128
```

> **Note**
>
> This functionality is not limited to dicts. (Tested to be working with lists and tuples).

12.3 Dynamics and Escape Sequences

As mentioned previously, control characters like `:`, `=` can be escaped either by quoting, or by using the escape charater `\`.

The resulting string will continue to function with dynamic values:

[12]:
```
template = """#!/bin/bash

#SBATCH --ntasks=#NTASKS#
#SBATCH --cpus-per-task=#CPUS_PER_TASK#
#SBATCH --nodes=#NODES:default={ntasks*cpus_per_task/cores_per_node}#

# using cores_per_node: #CORES_PER_NODE:default=128:hidden=True#

# tasks to nodes ratio: #ratio:default={ntasks}\\:{nodes}#
"""

test = Computer(template=template)
```

[13]:
```
print(test.script(ntasks=64, cpus_per_task=4))
```

```
#!/bin/bash

#SBATCH --ntasks=64
#SBATCH --cpus-per-task=4
#SBATCH --nodes=2


# tasks to nodes ratio: 64:2
```

12.3.1 Escaping Evaluation

Using the backslash escape technique is the only method for escaping the `{}` evaluation pattern within values.

[14]:
```
template = r"""srcdir = #src:default=test#
dir = #dir:default=$\{HOME\}/{src}#"""

test = Computer(template=template)
```

[15]:
```
print(test.script())
```

```
srcdir = test
dir = ${HOME}/test
```

Try this without the escape `\`, you'll notice that the script production will fail, as it's expecting a `home` parameter.

12.4 Summary

Taking into account the previous two tutorials, you should now have all the tools needed to generate suitable jobscripts for any machine.

To summarise the steps:

1. Obtain a valid jobscript. This can be either from a job that you know has run (either yours or another user's), the documentation, or the machine helpdesk.
2. Identify the parts of this script that you would like to have control over. `ntasks`, `nodes`, etc.
3. Convert the static parameters that the script has into sensible parameter names.
4. Load the template into a `Computer`
5. `Dataset` can now use this object to generate scripts based on the arguments that you give.

> **Tip**
>
> Remember that you can debug your script at any point manually by printing the output of the `script()` method.

CHAPTER

THIRTEEN

ADVANCED CONNECTION PARAMETERS

13.1 Prequisites

In the quickstart guide, we covered the basic utilities required to run a python function on a remote machine using a `URL`.

The previous tutorials covered machine definitions using `BaseComputer`.

As this is a subclass of `URL`, you can further enhance your workflows with the extra functionalities provided.

13.1.1 Creating a Connection

Assuming we have a machine which we can `ssh` into without issues, lets create a `URL`:

[2]:
```
from remotemanager import URL

# we will use a localhost connection to ensure compatibility for this tutorial
connection = URL(host='localhost')
```

We now have the concept of a "connection" to a remote machine. Though here we are using a simple "localhost" connection, remember that URL is able (and intended to) connect outside of your current workstation by specifying a connection to that machine.

A simple rule-of-thumb is to use `URL(<remote>)` where `<remote>` is whatever you would use for a `ssh <remote> ...` command.

See the relevant Quickstart section for more info.

> **Tip**
>
> You can quickly check your connection any time by issuing a command on the machine with `connection.cmd('...')`. `pwd` and/or `ls` would also likely alert you to if you're connected to the right machine or not.

13.1.2 Testing your Connection

Added in version 0.5.10.

`url` also provides a `test_connection()` method which will attempt to connect to the remote and run a test suite. The results of this test are a strong indicator of whether `remotemanager` can run jobs on your machine.

[3]:
```
connection.test_connection()
```

```
Checking for entry point... Success (/home/ljbeal/Work/Devel/remotemanager/docs/
↪source/tutorials)
Checking file creation in home... True
```

(continues on next page)

(continued from previous page)

```
Checking file creation in /tmp... True
Checking file creation in /scratch... False
Testing remotemanager.transport.rsync:
        send... Transferring 3 Files... Done
True
        pull... Transferring 1 File... Done
True
Testing remotemanager.transport.scp:
        send... Transferring 3 Files... Done
True
        pull... Transferring 1 File... Done
True
Cleaning up... Done
Done! Made 15 calls, taking 0.19s
Approximate latency, 0.01s
Tests passed successfully
```

The results are returned in dictionary format and can be queried here, or from the `URL` itself:

`test_connection` creates a `ConnectionTest` object and runs the contained tests. Within, we test the minimal required functionality for running jobs:

1. connection to the remote
2. creation of a file in at least the home dir
3. functional transport system

Provided these three conditions are true, the test will evaluate as passed.

[4]:
```
connection.connection_test.passed
```

[4]:
```
True
```

Some extra parameters are checked, and stored within the `data` (which stores useful info), and `extra` (which stores errors and minor details). You can query these if you are interested in their content.

The test also runs some basic timing checks and calculates a very rough latency:

[5]:
```
print(connection.connection_test.latency)

print(connection.latency)  # this is also available from the root URL object
```

```
0.012922207514444986
0.012922207514444986
```

Ping

Added in version 0.5.9.

`URL` also provides a `ping()` method, which will attempt to run the `ping` command on your system, targeting the remote. This takes the arguments `n`, waiting for `n` returns from the remote (defaults to 5), and `timeout`, which limits the total duration (defaults to 30s).

This method will return the delay in ms as a float.

13.2 URL.cmd

URL is a powerful interface between python and your remote system, the method that will likely be most used in your workflows is `URL.cmd()`.

It has been mentioned occasionally in earlier tutorials, so here we will go through some of the more specialised features.

```
[6]: connection.cmd('echo "this command is executed on the remote"')
```

```
[6]: this command is executed on the remote
```

Internally `URL` creates a `CMD` object, and then executes the command, adding the appropriate `ssh` in order to operate over the network. We will see later how this can be fine tuned.

13.2.1 Error handling

By default, cmd will raise any errors encountered. If cmd detects anything on `stderr` (that isn't an empty string), it will be raised as a RuntimeError:

```
[7]: connection.cmd('do a thing')
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[7], line 1
----> 1 connection.cmd('do a thing')

File ~/Work/Devel/remotemanager/remotemanager/connection/url.py:729, in URL.cmd(self,␣
↪cmd, asynchronous, local, stdout, stderr, timeout, max_timeouts, raise_errors, dry_
↪run, prepend, force_file, landing_dir, stream, verbose)
    726 if dry_run:
    727     return thiscmd
--> 729 thiscmd.exec()
    730 if not local:
    731     self._callcount += 1

File ~/Work/Devel/remotemanager/remotemanager/connection/cmd.py:369, in CMD.exec(self,
↪ verbose)
    366     return self._fexec(stdout, stderr, verbose)
    368 try:
--> 369     self._exec(stdout, stderr, verbose)
    370 except OSError as E:
    371     msg = "Encountered an OSError on exec, attempting file exec"

File ~/Work/Devel/remotemanager/remotemanager/connection/cmd.py:469, in CMD._
↪exec(self, stdout, stderr, verbose)
    467 if not self._async and not self.is_redirected:
    468     logger.debug("in-exec communication triggered")
--> 469     self.communicate(verbose=verbose)

File ~/Work/Devel/remotemanager/remotemanager/connection/cmd.py:600, in CMD.
↪communicate(self, use_cache, ignore_errors, verbose)
    598         logger.warning("locale error detected: %s", err)
    599     else:
--> 600         raise RuntimeError(f"received the following stderr: \n{err}")
    602 self._stdout = _clean_output(std)
    603 self._stderr = _clean_output(err)
```

(continues on next page)

(continued from previous page)

```
RuntimeError: received the following stderr:
/bin/bash: -c: line 1: syntax error near unexpected token `do'
/bin/bash: -c: line 1: `do a thing'
```

Spurious Errors

Some systems can place non-critical warnings onto `stderr`, which can cause otherwise perfectly functional workflows to think they have failed. If this is the case, you can use `raise_errors=False`.

> **Note**
>
> If you have a situation where a *machine* is raising non-fatal errors, `raise_errors=False` can be passed to the actual `URL`, which sets and `cmd` call to ignore errors by default.

13.2.2 The CMD Object

`URL.cmd` also returns a `CMD` *object*, which can be stored and queried.

The most useful properties are `stdout` and `stderr` which allow access to these attributes after a call.

> **Note**
>
> If a direct `CMD` call is important, it is advisable to capture it within a variable (such as below). `URL` does keep a history, but it is limited in size.

[8]:
```
output = connection.cmd('do a thing', raise_errors=False)
```

[9]:
```
print('cmd stdout:', output.stdout)
print('cmd stderr:', output.stderr)
```

```
cmd stdout:
cmd stderr: /bin/bash: -c: line 1: syntax error near unexpected token `do'
/bin/bash: -c: line 1: `do a thing'
```

There are other useful attributes attached to this object, which may assist in your workflows, or debugging:

[10]:
```
print('Shell process id is:', output.pid)
print('Working dir of the call is:', output.pwd)
print('The command that was sent is:', output.sent)
print('User that executed the cmd:', output.whoami)
print('You also have access to the returned code:', output.returncode)
```

```
Shell process id is: 42294
Working dir of the call is: /home/ljbeal/Work/Devel/remotemanager/docs/source/
↪tutorials
The command that was sent is: do a thing
User that executed the cmd: ljbeal
You also have access to the returned code: 2
```

You can also use the `output.kill()` method, which will attempt to terminate the process.

CMD History

Added in version 0.6.1.

URL captures your most recent `cmd` calls within a `cmd_history` property.

This has a fixed length set by `url.cmd_history_depth` (defaults to 10).

This is useful for debugging unexpected results from a call which was not captured within a variable.

[11]:
```
print(connection.cmd('pwd'))
print(connection.cmd_history[-1])
```

```
/home/ljbeal/Work/Devel/remotemanager/docs/source/tutorials
/home/ljbeal/Work/Devel/remotemanager/docs/source/tutorials
```

[12]:
```
print(type(connection.cmd_history[-1]))
```

```
<class 'remotemanager.connection.cmd.CMD'>
```

13.2.3 Async calls

Up until now we have been calling commands sequentially and waiting for the result, however it's possible to launch a command and proceed without waiting.

Below we have 2 structures that issue a command that waits for 3s, then returns the string "finished!"

We will time how long the execution takes, and how long it takes to get back the result:

[14]:
```
import time

t0 = time.time()
output1 = connection.cmd('sleep 3 && echo "finished!"')

t1 = time.time()
dt = int(round(t1 - t0))
print(f'call took ~{dt}s')

print(output1)
t2 = time.time()

dt = int(round(t2 - t1))
print(f'collecting the results took ~{dt}s')
```

```
call took ~3s
finished!
collecting the results took ~0s
```

[15]:
```
t0 = time.time()
output2 = connection.cmd('sleep 3 && echo "finished!"', asynchronous=True)

t1 = time.time()
dt = int(round(t1 - t0))
print(f'call took ~{dt}s')

print(output2)
t2 = time.time()

dt = int(round(t2 - t1))
print(f'collecting the results took ~{dt}s')
```

```
call took ~0s
finished!
collecting the results took ~3s
```

As we can see, the first call waits for completion, returning the result. The second call however skips this waiting phase, and we don't actually have to wait for the command to execute until *after* we request the result

13.3 Fine Tuning a `cmd` Call

URL has some further options available which may enhance your workflows in a remote setting. We have already seen the `asynchronous` argument, lets look at a few more.

To show these more in depth systems, we shall create a "dummy" connection:

```
[16]: dummy = URL(user='username', host='remote.connection.address')
```

13.3.1 Dry Run

If you're about to issue a command which could be potentially destructive (or time intensive), it is wise to check that it actually looks sensible.

`dry_run` does just this. Instead of executing the command on the remote, it will simply return what it *would* excecute as a string.

```
[17]: dummy.cmd('echo "this call will just be returned as a string"', dry_run=True)
[17]: ssh -p 22 -q username@remote.connection.address 'echo "this call will just be␣
      ↪returned as a string"'
```

13.3.2 Local

A useful flag which you may need to use, is `local`.

This allows you to run commands on your local machine, even using a `URL` that is pointed at a remote. See the change in command here:

```
[18]: dummy.cmd('echo "this call will just be returned as a string"', local=True, dry_
      ↪run=True)
[18]: echo "this call will just be returned as a string"
```

As this command skips over the remote portion, we don't actually need the `dry_run` here.

```
[19]: dummy.cmd('echo "this call will just be returned as a string"', local=True)
[19]: this call will just be returned as a string
```

13.3.3 Forcing a file-type execution

The internal CMD object has a special run-mode where it will first dump the cmd to a file, then execute that *file* with bash.

Normally this is used as a backup for a situation where the cmd can fail to execute. However you can force this behaviour by passing force_file=True.

Once this cmd is communicated with, the file will be cleared from the system, so we need to use asynchronous=True to prevent this.

[20]:
```
file_cmd = dummy.cmd('echo "this call will just be returned as a string"', local=True,
↪ force_file=True, asynchronous=True)
```

The filename is stored temporarily in the redirect attribute of the resulting CMD object.

[21]:
```
tempfile = file_cmd.redirect["execfile"]

print(tempfile)
```

```
221971b2.sh
```

[22]:
```
with open(tempfile) as o:
    print(o.read())
```

```
echo "this call will just be returned as a string"
```

Now if we access the result of the call, it will attempt to communicate with the process, removing the file as it does so.

[23]:
```
print(file_cmd.stdout)
```

```
this call will just be returned as a string
```

13.4 Global CMD Parameters

The remaining options can be set at the URL level (not just on the cmd() call), so we'll demonstrate them there.

Options set this way then apply to *all* cmd calls issued by that URL. (Though any args passed to the cmd() method will override them)

13.4.1 Timeout Parameters

Each call to the remote will attempt to gracefully handle a timeout. In the case of a slow connection, a timeout will occur after timeout seconds. The operation of this is as follows:

1. If a connection takes longer than timeout seconds to respond, it will issue an internal timeout error.
2. CMD will then wait for timeout seconds, then retry.
3. If the attempt fails again, CMD will wait for n*timeout and repeat, where n is the number of current failures + 1.
4. This continues until max_timeouts is reached, when a RuntimeError will be raised instead.

timeout defaults to 5s and max_timeouts defaults to 3 attempts

> **Note**
>
> This occurs on the `communicate` side of a `CMD` exec, so an `asynchronous` call will not see this until you try to access the output (or trigger `communicate` in another way).

[24]:
```
dummy = URL(user='username', host='remote.connection.address', timeout=10, max_
↪timeouts=5)
```

[25]:
```
print(dummy.timeout)
```
```
10
```

[26]:
```
print(dummy.max_timeouts)
```
```
5
```

Added in version 0.13.4.

> **Note**
>
> You can now disable the timeout function by setting `timeout` to `0`, a negative number or `False`.

Landing Directory

Added in version 0.9.19.

By default, a `URL.cmd` will "land" in the default directory that a standard `ssh` would. This can be configured via the `landing_dir` argument.

To demonstrate this, we will have to hop back over to a functional `URL`.

(It will also be helpful to create a directory to land in using our main `URL`, `connection`.

[27]:
```
connection.cmd("mkdir -p inner_directory")

print("initial landing dir:", connection.cmd('pwd'))
print("updated landing dir:", connection.cmd('pwd', landing_dir="inner_directory"))
```
```
initial landing dir: /home/ljbeal/Work/Devel/remotemanager/docs/source/tutorials
updated landing dir: /home/ljbeal/Work/Devel/remotemanager/docs/source/tutorials/
↪inner_directory
```

Now lets do this at the `URL` level.

[28]:
```
url = URL(landing_dir="inner_directory")

url.cmd("pwd")
```
[28]:
```
/home/ljbeal/Work/Devel/remotemanager/docs/source/tutorials/inner_directory
```

13.5 Editing the ssh string

URL has an `ssh` property which will return the string that allows interfacing with the remote. If this needs updating for whatever reason it can be overridden by simply setting the attribute. The example below is used to remove a locale error that can occur on some systems.

Added in version 0.6.0: This specific update is no longer needed, as the locale errors are ignored by default. However the functionality of modifying your `ssh` remains.

[29]:
```
print('initial ssh string:', dummy.ssh)

dummy.ssh = 'LANG=C ' + dummy.ssh

print('updated ssh string:', dummy.ssh)
```

```
initial ssh string: ssh -p 22 -q username@remote.connection.address
updated ssh string: LANG=C ssh -p 22 -q username@remote.connection.address
```

To undo this change, set `ssh` to `None`, or call `url.clear_ssh_override()`

[30]:
```
dummy.ssh = None
dummy.clear_ssh_override()
print('the reverted ssh string is', dummy.ssh)
```

```
the reverted ssh string is ssh -p 22 -q username@remote.connection.address
```

13.6 URL.utils

URL also provides a `utils` module which provides both commonly used functions, and more complex ones. First of these is the `mkdir` and `touch` methods, which will create a dir and file with the given path, respectively

[31]:
```
test_mtime = int(time.time())

connection.utils.mkdir('temp_utils_test')
connection.utils.touch('temp_utils_test/create_me')
connection.utils.touch('temp_utils_test/create_me_also')
```

[31]:

There is also `utils.ls`, which returns the files as a list by default

[32]:
```
connection.utils.ls('temp_utils_test')
```

[32]:
```
['create_me', 'create_me_also']
```

The more powerful functions granted by utils is the `search_folder`, `file_presence` and `file_mtime` methods.

These methods allow searching for a list of files, condensing the the query down to a single call in each case. This is useful for high latency remote systems, where an `ls` search for 100+ files could take a long time. These functions will do this in a single call.

`search_folder` takes a list of files and a folder, returning a {file: bool} "truth-dict" of whether those files are present

[33]:
```
connection.utils.search_folder(['create_me', 'not_present'], 'temp_utils_test')
```

[33]:
```
{'create_me': True, 'not_present': False}
```

Similarly, a more general form exists in `file_presence`, which will take a list of files and return a similar truth-dict of their `ls` presence

```
[34]: connection.utils.file_presence(['temp_utils_test/create_me',
                                      'missing_folder/file',
                                      'temp_utils_test/not_present'])
```

```
[34]: {'temp_utils_test/create_me': True,
       'missing_folder/file': False,
       'temp_utils_test/not_present': False}
```

If the file modification time is what you want, then `file_mtime` can be used in a similar way. This is the method called internally in `file_presence`, so incurrs no extra runtime

```
[35]: times = connection.utils.file_mtime(['temp_utils_test/create_me',
                                           'temp_utils_test/create_me_also',
                                           'temp_utils_test/not_present'])

      print(times)
```

```
{'temp_utils_test/create_me': 1738078661, 'temp_utils_test/create_me_also':␣
↪1738078661, 'temp_utils_test/not_present': None}
```

13.7 Tunnels

ssh tunnels allow a peristent connection to a machine. You could create a tunnel to a machine hosting a jupyter instance to access it locally, for example.

Lets demonstrate what that looks like.

```
[37]: remote_url = URL("remote.host")

      remote_url.cmd("jupyter lab --ip=0.0.0.0", dry_run=True)
```

```
[37]: ssh -p 22 -q remote.host 'jupyter lab --ip=0.0.0.0'
```

> **Note**
>
> The `--ip=0.0.0.0` modifier is required to allow external connections.

This would start a jupyter lab session with the base python. Note that this is a simplified example, in your case you will most likely need to follow a different procedure to start jupyter.

In any case, lets assume that the server is running on port `8888` (the default) on `remote.host`.

Now we can create a tunnel to it. If we have (or want) any locally run servers, we now cannot reuse port 8888. So lets redirect to 9999.

```
[38]: tunnel = remote_url.tunnel(local_port=9999, remote_port=8888, background=True, dry_
      ↪run=True)

      print(tunnel.cmd)
```

```
ssh -p 22 -q remote.host -N -L :9999:remote.host:8888 remote.host
```

This command creates a tunnel between your machine and the remote. The jupyter server will now be available at `127.0.0.1:9999`

The local ip address can be changed by setting `local_address`.

Important

`remotemanager` will attempt to avoid leaving "dangling" tunnels open. To help with this, the PID of a non `dry_run` tunnel will be reported. However assigning the tunnel to a variable is advised, allowing you to call the `kill()` method.

Note

The `with` context (below) is the preferred method of handling tunnels, if it is possible for your use case. This will handle the safe closure of the tunnel on your behalf, even in the case of an exception.

13.8 The `with` context

It is possible to execute commands using the python `with` context. This ensures that the process is properly killed if an exception occurs.

Note

This can be used to ensure that your tunnels are closed if you're using them within a script.

We can demonstrate this by generating a long running async command and then causing a failure within the context.

[39]:
```
t0 = time.time()

with url.cmd("sleep 300 && echo 'foo'", asynchronous=True) as c:
    print(f"my PID is {c.pid}")
    raise RuntimeError("Raise an exception here to force the with(...) to exit")
```

```
my PID is 42323
Exiting context, killing pid 42323
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[39], line 5
      3 with url.cmd("sleep 300 && echo 'foo'", asynchronous=True) as c:
      4     print(f"my PID is {c.pid}")
----> 5     raise RuntimeError("Raise an exception here to force the with(...) to exit
↪")

RuntimeError: Raise an exception here to force the with(...) to exit
```

[40]:
```
print(f"dt: {time.time() - t0:.2f}s")
```

```
dt: 0.02s
```

Note that the time to execute is significantly shorter than the sleep. We can check that the process no longer exists by querying the pid:

[41]:
```
import psutil
psutil.pid_exists(c.pid)
```

[41]:
```
False
```

13.9 ProxyJump

If you can connect with an `ssh ...` command, you can do it using URL. An extension to this is `ProxyJump`, which allows you to "hop" between hosts to get to your destination. If you have this set up, you will have an ssh config file that looks somewhat like this:

```
Host remote-endpoint
    User username
    Hostname remote.endpoint.address
    ProxyJump remote-middleman
Host remote-middleman
    User username
    Hostname remote.middleman.address
```

The following URL is an example that would connect using these parameters:

[42]:
```
proxyurl = URL(host='remote-endpoint')

print(proxyurl.userhost)
```

```
remote-endpoint
```

[43]:
```
print(proxyurl.cmd('echo "test"', dry_run=True))
```

```
ssh -p 22 -q remote-endpoint 'echo "test"'
```

13.10 Setting a Default URL

The standard default `URL` is one pointed at `localhost`, this is in reality a safety measure to ensure that the `Dataset` at least *has* a `URL`. However this is not ideal for a remote workflow.

The `default_url` is a property of `Dataset`, and can be set at the object level after importing.

[44]:
```
from remotemanager import Dataset

def func(inp):
    return inp

ds = Dataset(func, skip=False)

print(ds.url.userhost)
```

```
localhost
```

[45]:
```
url = URL("user@host")

Dataset.default_url = url
```

[46]:
```
ds = Dataset(func, skip=False)

print(ds.url.userhost)
```

```
user@host
```

13.11 I'm seeing errors that look like system messages

Added in version 0.10.15.

It is normal for machines to output information on connection. Usage, documentation, disk quotas, etc. Sometimes, however sometimes these messages can be emitted on stderr, rather than stdout. `remotemanager` will see this and assume that something has gone wrong. To prevent this, by default all ssh calls use the `-q` flag.

13.11.1 ssh `-q` flag

This flag suppresses most errors and warnings, and should allow for smoother control of your machine. If you're seeing strange behaviour with errors not being properly collected, you can either set

`url.quiet_ssh = False`, or initialise URL with `URL(..., quiet_ssh = False)`

CHAPTER

FOURTEEN

REMOTE COMMANDS

URL has a set of default commands that are used to issue actions on the remote. While these work for *most* situations, they can be changed in the event that they don't.

14.1 Python, Submitter and Shell

URL has 3 main running commands:

1. `python`: Used to launch a python script, defaults to `python`
2. `submitter`: Used to submit the jobscript, defaults to `bash` (direct submission)
3. `shell`: Used to change the command which is used to run the master script, defaults to `bash`

Of these, python and submitter are the most important.

Shell exists for an edge case where you need a specific command to run scripts, and can usually be left at its default.

> **Note**
>
> If `submitter` is left default, yet `shell` has been changed, `url.submitter` will return the value of `shell`. For example, setting `url.shell = "sh"` with a default `submitter` will cause `url.submitter` to return `"sh"`

Lets set up a quick example that shows what changes are made when all three of these are set:

[1]:
```
from remotemanager import Dataset, URL

url = URL(python='python3', submitter='sbatch', shell = 'my_shell')

def f():
    return

ds = Dataset(f, url=url, skip=False)

ds.append_run()
```

```
appended run runner-0
```

The run command is where the `shell` takes effect, it is what is used to execute the master script.

[2]:
```
print("run command:")
ds.run(dry_run=True)
```

```
run command:
Running Dataset
assessing run for runner dataset-06a84b6d-runner-0... running
launch command: cd temp_runner_remote && my_shell dataset-06a84b6d-master.sh
```

The contents of the master script are somewhat confusing to read.

Since the master script deals with the actual job submission, there is extra content in there to make sure the jobs run properly.

The important thing for us here is the `sbatch dataset_...` section in the middle of the long line.

[3]:
```
print('master script:')
print(ds.master_script.content)
```

```
master script:
sourcedir=$PWD
rm -f dataset-06a84b6d.manifest
echo $(date +'%d/%m/%Y %H:%M:%S') 66c0efed submitted >> dataset-06a84b6d.manifest &&␣
↪sed -i -e "s#66c0efed_master#$sourcedir#" dataset-06a84b6d-runner-0-jobscript.sh &&␣
↪ sbatch dataset-06a84b6d-runner-0-jobscript.sh 2> dataset-06a84b6d-runner-0-error.
↪out
```

The jobscript is what deals with the actual job execution. Here it is nice and simple, the final line is using our `python3` specification.

[4]:
```
print('jobscript:')
print(ds.runners[0].jobscript.content)
```

```
jobscript:
#!/bin/bash


export DIR_66c0efed=66c0efed_master
python3 dataset-06a84b6d-runner-0-run.py 2>> dataset-06a84b6d-runner-0-error.out
```

CHAPTER

FIFTEEN

CHAINING DATASETS

> **Warning**
>
> Dependencies are still under development, the current implementation has been added owing to its usefulness, though expect changes in subsequent releases. Every effort will be made to ensure the stability of current methods, however you should always check release notes before upgrading versions if you have an important workflow running with dependencies.

Imagine a scenario where you want to run a heavy calculation which produces a very large output. If you wanted to postprocess those results, you could collect them with `fetch_results` and process them locally. This, however, could take time to transfer, and uses up disk space locally. Datasets have a better way of doing this, called "dependencies".

15.1 Dependencies

Dependencies allow you to chain jobs together on the remote machine, submitting with different parameters. For example, we can submit a calculation on 50 nodes, and then postprocess the result using only one cpu core.

Currently, dependencies are limited to a linear chain, and only on a one-to-one basis. This means jobs can be joined together in an A -> B -> C sense, and their runners will have a continuous line between them. While expansion is planned, lets look into the current implentation and how it can help our hypothetical scenario.

Lets begin by defining three functions this time:

[2]:
```
from remotemanager import Dataset, URL

def init(offset):
    return offset

def mult(x, y):
    offset = loaded
    return offset + (x * y)

def post():
    return f'The final result is {loaded}'
```

This workflow does three things:

1. An "offset" is specified. Think of it as the `c` in a `y = mx + c` equation.
2. Two numbers are multiplied together and added to the offset (your `mx`).
3. The result is formatted into a string and returned.

15.1.1 An aside on `loaded`

`loaded` is a property that is added by the dependency network, and it simply allows a function to access the returned value of the function immediately before it in the chain.

Think of it as exactly what is given by the `return` of a function:

[3]:
```
def multi_return():
    return 1, 2, 3

def multi_process():
    a, b, c = loaded

    return a + b + c
```

The output of the above functions would always be 6.

15.2 Creating your chained runners

Going back to our original workflow, lets create the datasets. This is done as normal:

[4]:
```
url = URL()  # again, a "local" url for testing purposes

dataset_init = Dataset(function = init,
                       name = 'init',
                       url = url,
                       remote_dir = 'temp_remote',
                       local_dir = 'temp_local',
                       mpi = 1,
                       omp = 1,
                       nodes = 1,
                       skip = False)

dataset_calc = Dataset(function = mult,
                       name = 'calc',
                       url = url,
                       remote_dir = 'temp_remote',
                       local_dir = 'temp_local',
                       mpi = 64,
                       omp = 4,
                       nodes = 50,
                       skip = False)

dataset_post = Dataset(function = post,
                       name = 'post',
                       url = url,
                       remote_dir = 'temp_remote',
                       local_dir = 'temp_local',
                       mpi = 1,
                       omp = 1,
                       nodes = 1,
                       skip = False)

dataset_init.set_downstream(dataset_calc)  # new option!
dataset_calc.set_downstream(dataset_post)
```

The new option here to pay attention to is `set_downstream`, though there is also the mirror of this in `set_upstream`. These dictate the order in which the datasets will be run.

Doing this creates a global `Dependency` object that can be accessed from any of the Datasets:

```
[5]: dataset_post.dependency.network
```

```
[5]: [(dataset-init-40221af0, dataset-calc-dce4a953),
      (dataset-calc-dce4a953, dataset-post-794d5d4f)]
```

```
[6]: dataset_init.dependency is dataset_calc.dependency is dataset_post.dependency
```

```
[6]: True
```

Here we can see two "edges" to this network, first a connection between `init` and `calc`, then a second one between `calc` and `init`

There also exists some extra methods to check whether a dataset has parents or children

```
[7]: print('init:')
     print('Is parent?', dataset_init.is_parent)
     print('Is child?', dataset_init.is_child)

     print('\ncalc:')
     print('Is parent?', dataset_calc.is_parent)
     print('Is child?', dataset_calc.is_child)
```

```
init:
Is parent? True
Is child? False

calc:
Is parent? True
Is child? True
```

You may append runs to any dataset within this chain, and a runner will be created in *every* dataset. When appending runs, you must include all arguments. Best practice here is to select a Dataset within the chain to append your runs to, and stick to it.

.. note::
Runners will be created with the *same arguments* in all datasets. This allows specification of args in more than just one dataset, however it does mean that you must be aware of this when defining functions.

Important

You can avoid "chaining" `run_args` through to the other datasets by passing `chain_run_args=False` to the run append. That way, any args such as `mpi`, `omp`, etc. will be passed *only* to the dataset to which the append was called.

```
[8]: dataset_post.append_run(args={'x': 5, 'y': 2, 'offset': 10})
     dataset_post.append_run(args={'x': 3, 'y': 7, 'offset': 0})
```

```
appended run runner-0
appended run runner-0
appended run runner-0
appended run runner-1
```

(continues on next page)

(continued from previous page)

```
appended run runner-1
appended run runner-1
```

Now we have appended runs to the main dataset, we can see that runners have been added to both:

[9]:
```
print(f'initial dataset has {len(dataset_init.runners)} runners')
print(f'calculation dataset has {len(dataset_calc.runners)} runners')
print(f'postprocess dataset has {len(dataset_post.runners)} runners')
```

```
initial dataset has 2 runners
calculation dataset has 2 runners
postprocess dataset has 2 runners
```

15.3 Running

Datasets can be run as normal, submit from any in the chain to initiate:

[10]:
```
dataset_calc.run()
```

```
Running Dataset
assessing run for runner init-40221af0-runner-0... running
assessing run for runner init-40221af0-runner-1... running
assessing run for runner calc-dce4a953-runner-0... running
assessing run for runner calc-dce4a953-runner-1... running
assessing run for runner post-794d5d4f-runner-0... running
assessing run for runner post-794d5d4f-runner-1... running
Transferring 15 Files... Done
```

[11]:
```
dataset_post.wait(1, timeout=10)
```

[12]:
```
dataset_post.fetch_results()
```

```
Fetching results
Transferring 2 Files... Done
```

[13]:
```
print(dataset_post.results)
```

```
['The final result is 20', 'The final result is 21']
```

15.4 Checking interim results

Here, our output looks as we expect, but what do we do if it doesn't? Well we could first off start by investigating the calculation dataset, as in a real case, it's likely to be the cause of the problems.

Internally, these datasets are no different to any other, so all their methods work exactly as you'd expect. The only limitations being the ones mentioned previously about appending runners and running parents.

[14]:
```
dataset_init.fetch_results()
print(dataset_init.results)
```

```
Fetching results
Transferring 2 Files... Done
[10, 0]
```

[15]:
```
print(dataset_init.get_all_runner_states())
```

```
[RunnerState("satisfied"), RunnerState("satisfied")]
```

> **Note**
>
> A final note on branching jobs: At present, jobs having multiple *children* is supported in a limited sense. So your job tree can expand as you go down the chain, however the opposite is not true. You can not "merge" returned values into a single child.

15.5 Envionment Variables

You can set environment variables with the `extra` keyword arg in datasets (or runners). With dependencies, there's some more info to note.

Any `extra` set at the Dataset level will be applied for that dataset.

So if you add `extra="export KEY='VAR'"` to a parent, it will be available for the children, but not in reverse.

[16]:
```
from remotemanager import RemoteFunction

@RemoteFunction
def search_env(nvars):
    import os

    found = {}
    for i in range(nvars):
        varname = f"var{i+1}"
        found[varname] = os.environ.get(varname, None)

    return found


def parent(nvars):
    return search_env(nvars)

def child(nvars):

    child_vars = search_env(nvars)

    output = {}
    for key in child_vars:
        output[key] = [loaded[key], child_vars[key]]

    return output

ds_1 = Dataset(parent, skip=False, extra="export var1='parentvar'")
ds_2 = Dataset(child, skip=False, extra="export var2='childvar'")

ds_1.set_downstream(ds_2)

ds_1.append_run({"nvars": 4}, extra="export var3='appendvar'")

ds_2.run(extra="export var4='runvar'")
```

(continues on next page)

(continued from previous page)

```
ds_2.wait(1, 10)

ds_2.fetch_results()

ds_2.results
```

```
appended run runner-0
appended run runner-0
Running Dataset
assessing run for runner dataset-b0832d53-runner-0... running
assessing run for runner dataset-1eaa68f9-runner-0... running
Transferring 7 Files... Done
Fetching results
Transferring 1 File... Done
```

[16]:
```
[{'var1': ['parentvar', 'parentvar'],
  'var2': [None, 'childvar'],
  'var3': ['appendvar', 'appendvar'],
  'var4': ['runvar', 'runvar']}]
```

Lets analyse these results.

`var1` was added at the level of the parent runner, so is exported in the first jobscript.

`var2` was addeda at the level of the child runner, so is exported in the *second* jobscript.

Because of this, `var1` is available in both the parent and child, whereas `var2` is only available in the child.

`var3` was added at the append level and is propagated to both appended runners (one for each dataset).

`var4` is similar, since it's set at the run level, it's temporary, but added to both datasets.

CHAPTER

SIXTEEN

JUPYTER MAGIC

IPython magic is a feature natively integrated into jupyter wherein functions can be turned into simple commands. For example, setting matplotlib inline mode by adding a cell with:

```
%matplotlib inline
```

`remotemanager` also has support for magic, providing `%%sanzu`. A cell headed with this magic will have its contents run on a remote machine via a Dataset. Firstly, we need to load the extension into this notebook:

[1]:
```
%load_ext remotemanager
```

16.1 Usage

At present this is mostly limited to doing simple tasks on a machine, though you can return results if you follow one simple rule. We'll cover that later, for now lets start with a basic task, creating a folder:

[2]:
```
from remotemanager import URL

connection = URL()
dirname = 'test'
```

Basic args set, now we can create our cell, which runs when we execute it.

The syntax is as follows: Always start with `%%sanzu`, as this calls the function that does the work. Then, you can follow up with arguments for your Dataset. Note that we don't need to specify `function=...` here, as the function *is* the cell.

If you need arguments for the cell (such as `dirname` here), prefix with `%%sargs`, and then continue with anything you need for the cell.

It may look strange to have to specify the local and remote dirs for this runner, though remember that this is still running a Dataset behind the scenes. Everything you can specify there also works here.

> **Note**
>
> If you are familiar with scheduler jobscripts, you can consider `%%sanzu` as something like a pragma such as `#SBATCH` or `#PBS`.

[3]:
```
%%sanzu url = connection
%%sanzu local_dir = "temp_local"
%%sanzu remote_dir = "temp_remote"
%%sargs dirname = dirname
%%sargs dirname2 = "testme"

import os
```

(continues on next page)

(continued from previous page)

```
os.mkdir(dirname)
os.mkdir(dirname2)
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-f771fa7a-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

Now lets see if our directory was created:

[4]:
```
'test' in connection.cmd('ls temp_remote').stdout
```

[4]:
```
True
```

16.2 Run Behaviour

Much like a standard Dataset run, `skip` is enabled by default. This means that a cell will only execute once, storing the result. This saves on resource usage, however can be undesirable in some situations. Lets demonstrate the skipping first:

[5]:
```
%%sanzu url = connection

7 * 7
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-b463218f-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

[5]:
```
49
```

[6]:
```
%%sanzu url = connection

7 * 7
```

```
runner runner-0 already exists
Running Dataset
assessing run for runner dataset-b463218f-runner-0... ignoring run for successful␣
↪runner
Fetching results
No Transfer Required
```

[6]:
```
49
```

Note the cell output warns us that the run was skipped.

We can disable this much like a normal Dataset, by setting `skip=False`, or `force=True`:

[7]:
```
%%sanzu url = connection
%%sanzu skip = False

7 * 7
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-b463218f-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

[7]:
```
49
```

We can see a "test" dir in there, which means it has. Everything else you see is functional files for the Runner that was created. Including a results file, which means…

16.3 Accessing results

By default the jupyter cell will return the result as though you had run it normally. But what if we want to use this in a later cell?

This is doable via accessing the dataset after the run. The function inserts a `magic_dataset` attribute into the jupyter runtime which can be accessed later on.

[8]:
```
%%sanzu url=connection

"this string has come from the cell!"
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-33535f10-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

[8]:
```
'this string has come from the cell!'
```

[9]:
```
print(magic_dataset.results)
```

```
['this string has come from the cell!']
```

Note that we follow the convention of Jupyter in that the last line is returned only if it is at the top level of indentation.

[10]:
```
%%sanzu url=connection, local_dir="temp_local", remote_dir="temp_remote"
if True:
    "nothing comes back"
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-9bba769e-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

[11]:
```
print(magic_dataset.results)
```

```
[None]
```

The use case for magic is if you need to run some failrly quick jobs on a remote machine. It could be a run on the front end which grabs some data from a calculation. It could be that the remote machine has access to programs you didn't install locally. The syntax of the magic is aimed at making it as clear as possible to the readers of your notebook what science you are doing, while somewhat transpararently giving you access to powerful machines.

16.3.1 Pulling extra results

Aside from the base `sanzu` (which enables the tool), and `sargs` (which enables arg passthrough), there is a third option: `spull`.

This option will flag objects within a cell for "pulling", inserting it into the general notebook stream.

[12]:
```
%%sanzu url = connection
%%spull output

output = []
for i in range(5):
    output.append(i)
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-70a96e99-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

[12]:
```
{'output': [0, 1, 2, 3, 4]}
```

This also skips the requirement to access the `magic_dataset`:

[13]:
```
output
```

[13]:
```
[0, 1, 2, 3, 4]
```

You are also not limited to a single output variable, and can have as many targets as you need. Ensuring that the targets are available is also not something that's needed, as they will simply return `None`:

[14]:
```
%%sanzu url = connection
%%spull output
%%spull val
%%spull foo

val = 10

output = []
for i in range(val):
    output.append(i)
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-f4c67926-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

[14]:
```
{'output': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 'val': 10, 'foo': None}
```

[15]:
```
print(output)
print(val)
print(foo)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
10
None
```

16.4 Errors

The goal of sanzu is to run a cell on a remote machine as though it is running locally. Ideally it should be transparent as though only the executor of the cell is changed. For this, any errors that are raised on the remote are emitted as a `RuntimeError` on the local side.

[16]:
```
%%sanzu url = connection

prin("test")
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-0bf81d0b-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

```
/home/test/remotemanager/remotemanager/decorators/magic.py:95: UserWarning: Sanzu␣
↪encountered an exception, see below, or access magic_dataset.errors
  warnings.warn(
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[16], line 1
----> 1 get_ipython().run_cell_magic('sanzu', 'url = connection', '\nprin("test")\n')

File ~/envs/main/lib/python3.11/site-packages/IPython/core/interactiveshell.py:2541,␣
↪in InteractiveShell.run_cell_magic(self, magic_name, line, cell)
   2539 with self.builtin_trap:
   2540     args = (magic_arg_s, cell)
-> 2541     result = fn(*args, **kwargs)
   2543 # The code below prevents the output from being displayed
   2544 # when using magics with decorator @output_can_be_silenced
   2545 # when the last Python token in the expression is a ';'.
   2546 if getattr(fn, magic.MAGIC_OUTPUT_CAN_BE_SILENCED, False):

File ~/remotemanager/remotemanager/decorators/magic.py:99, in RCell.sanzu(self, line,␣
↪cell, local_ns)
     94 if ds.runners[0].is_failed:
     95     warnings.warn(
     96         "Sanzu encountered an exception, see below, "
     97         "or access magic_dataset.errors"
     98     )
---> 99     raise RuntimeError(ds.runners[0].error)
    101 for name in spull:
    102     logger.debug("looking for pull target %s", name)

RuntimeError: NameError: name 'prin' is not defined. Did you mean: 'print'?
```

CHAPTER

SEVENTEEN

DECORATORS

17.1 Intro to Decorators

Decorators are a feature of python that allows extra functionality to be added to objects. A simple example of this is a decorator which formats the output for you

[1]:
```
# actual decorator name, takes a function as an argument
def format_to_string(func):

    # internal decorator parser, this does the "work"
    def formatter(*args, **kwargs):
        # create this as a simple parser, returning the result in a string
        return f"The result is: '{func(*args, **kwargs)}'"

    return formatter

# the @ syntax calls our function as a decorator
@format_to_string
def multiply(x, y):
    return x * y

# lets see what it does to the output
multiply(4, 7)
```

[1]:
```
"The result is: '28'"
```

This example shows a basic use of a decorator. Python has internal decorators, a good example is the `@classmethod` which is used to modify a class method to return an instance of that `class`.

Likewise, `remotemanager` also provides some decorators for usage:

- `RemoteFunction` allows users to "tag" extra functions to be brought along with a main Dataset
- `SanzuFunction` operates similar to sanzu, but can be used directly within scripts

17.2 RemoteFunction

It is a fundamental idea of programming to convert repetitive sections of code into functions, turning whole blocks into single lines.

Lets imagine a two stage workflow for processing a number:

1. input is processed according to the rules: if num > 100 -> x0.5, else x2
2. format the output into a string format for easy reading

> **Note**
>
> This is an extremely basic example, designed to show the limitations of a base `Dataset`.

[2]:
```
num = 200
if num > 100:
    num = num / 2
else:
    num = num * 2

num = float(num)
print('The result is', num)
```

```
The result is 100.0
```

17.2.1 Using Functions

While this *works*, it does not scale well to large sets of inputs. A standard way of increasing the scalability is to define functions which can be called on any inputs:

[3]:
```
# define the formatter
def formatted(temp):
    return f'The result is {float(temp)}'

# and now the processing function
def process(number):
    if number > 100:
        return formatted(number / 2)
    return formatted(number * 2)

print(process(74))
```

```
The result is 148.0
```

17.2.2 Single Function Limits

It is here where we run into an issue with remotemanager as we can only define a single function for the Dataset to hold.

There are two paths we can take here using native python:

1. Refactor the workflow to be contained within a single function
2. Use an inner function

Obviously refactoring here is trivial to do, however that approach can get cumbersome very quickly with even small increases in complexity. Lets start with option 2, inner functions:

[4]:
```
def process(number):
    """
    This function halves any number above 100,
    doubling otherwise.
    """

    def formatted(temp):
        return f'The result is {float(temp)}!'
```

(continues on next page)

(continued from previous page)

```
    if number > 100:
        return formatted(number / 2)
    return formatted(number * 2)

print(process(200))
print(process(42))
```

```
The result is 100.0!
The result is 84.0!
```

17.2.3 The Third Option

Obviously this tutorial wouldn't exist if there wasn't some way around this limitation so `remotemanager` takes this a step further and gives you a third option: Allowing you to mark extra functions for sending. These functions are added in addition to the one placed within `Dataset`. For this we use the `RemoteFunction` decorator.

This is useful in a situation where you have multiple datasets holding different functions, but want a single formatting function for all jobs, for example.

For this workflow, we would be adding `process` to the `Dataset`, which means we should also indicate that we need `formatted` in this workflow:

[5]:
```
from remotemanager import Dataset, URL, RemoteFunction

@RemoteFunction
def formatted(temp):
        return f'The result is {float(temp)}!'

def process(number):
    if number > 100:
        return formatted(number / 2)
    return formatted(number * 2)

print(process(200))
print(process(42))
```

```
The result is 100.0!
The result is 84.0!
```

[6]:
```
url = URL()

ds = Dataset(process,
             url = url,
             skip = False)

ds.append_run({'number': 200})
ds.append_run({'number': 42})
```

```
appended run runner-0
appended run runner-1
```

[7]:
```
ds.run()
ds.wait(2)
ds.fetch_results()
print(ds.results)
```

```
Running Dataset
assessing run for runner dataset-7d7936c0-runner-0... running
assessing run for runner dataset-7d7936c0-runner-1... running
Transferring 6 Files... Done
Fetching results
Transferring 2 Files... Done
['The result is 100.0!', 'The result is 84.0!']
```

17.2.4 Expandability

You are not limited to a single extra function, so go wild! All stored functions are also available to *all* Datasets within the notebook in which they are defined, so you can further reduce boilerplate code in complex workflows where function sharing would be beneficial.

> **Warning**
>
> The functions that are cached are not stored within the databases themselves, so must always be defined within your notebook.

17.3 SanzuFunction

Added in version 0.10.9.

Similar to sanzu, however instead of executing a cell, you can designate a function to be remote callable.

> **Note**
>
> You can pass any `run_args` you may need directly to the decorator, and they will be attributed to the `Dataset` that is created.

[8]:
```
from remotemanager import URL, SanzuFunction

url = URL("localhost")

@SanzuFunction(url=url)
def execute_remotely(x, y):
    return x * y

print(execute_remotely(x=10, y=9))
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-776c1365-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
90
```

17.3.1 Choosing a Remote

Passing a `URL` at the decorator level will "bake in" that url to the function, causing it to always be called on that remote. Without passing a url, however, the Dataset will use its `default_url` property if set.

> **Note**
>
> To set the `default_url` property you can import `Dataset` then update with `Dataset.default_url = URL(...)`

Added in version 0.10.10: Now SanzuFunctions can be called with non keyword args.

[9]:
```
print(execute_remotely(7, 3))
```

```
appended run runner-1
Running Dataset
assessing run for runner dataset-776c1365-runner-1... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
21
```

CHAPTER

EIGHTEEN

SERIALISING COMPLEX OBJECTS

18.1 Pass the `class`

`Dataset` uses `JSON` for serialisation by default. This works fine for simple calcuations, however sometimes it is inconvenient (or impossible) to rely purely on JSON serialisable objects.

This can be easily demonstrated by creating a function that creates a `class`:

[1]:
```
def classcreate(name):
    class tmp:

        def __init__(self, name):
            self.name = name

    return tmp(name)

test = classcreate("test")

print(test.name)
```

```
test
```

This works just fine for us locally, however if we tried to wrap this into a `Dataset`, we'd get a `JSON` error.

18.1.1 Swapping out the serialiser

There are currently 4 inbuilt serialisers within `remotemanager.serialisation` you can swap to:

- `serialjson` (default)
- `serialyaml`
- `serialdill*`
- `serialjsonpickle*`

Those marked with a * are able to serialise complex objects.

To swap out the serialiser, simply import them and pass them at dataset creation, as you would a `URL`:

[2]:
```
from remotemanager import Dataset
from remotemanager.serialisation import serialdill

ser = serialdill()

ds = Dataset(function=classcreate,
                 serialiser=ser,
                 skip=False)
```

(continues on next page)

(continued from previous page)

```
ds.append_run({"name": "dilltest"})

ds.run()

ds.wait(1, 10)

ds.fetch_results()

ds.results[0].name
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-dbcaead1-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

```
[2]: 'dilltest'
```

18.1.2 Requirements

For these serialisers to work, you must have their prerequisite package installed:

`serialdill` requires `pip install dill` `serialjsonpickle` requires `pip install jsonpickle`

18.2 Creating your own serialiser

If none of these options suit your use case, there remains the possibility of creating your own class. To do so, you should subclass `../remotemanager.serialisation.serial` docs link here, implementing the methods as seen in the various subclasses of `serial` yaml, for example

CHAPTER

NINETEEN

SCRIPT DATASETS

Dataset focuses on running python scripts on the remote machine, however it is not locked into this run method.

19.1 Running with Script

Instead of setting the Dataset function to a python function, you are also able to use a `Script` object.

To enable this, you should first create a template.

Then, instead of using `append_run` with the arguments of a function, you can use the parameters that exist within your template.

> **Note**
>
> It is also possible to specify `None` as a function, and the arguments will be instead substituted into the `Computer.script()` method in addition to the run_args.

19.1.1 Output

Running in this manner means that this script will be executed and the `stdout` captured as the "result". `stderr` is captured as normal.

To demonstrate this behaviour lets set up a run that exports an environment variable in the submission script, and then uses that parameter within the actual run.

To do this, we will use the Dataset, Script and Computer modules.

[2]:
```
from remotemanager import Dataset, Script, Computer
```

First, lets create a submission script that exports a variable to be used:

[3]:
```
template = 'export sub_value=#sub_value#'

url = Computer(template=template)
```

Now, lets use `sub_value`.

Here, we're using the bash arithmetic wrapper to multiply two numbers together.

> **Note**
>
> Note that since our "result" has to be present on stdout, we should echo this value. As a result, it will always be a string.

[4]:
```
template = "echo $(( sub_value * #inner_value# ))"

script = Script(template=template)
```

Looks good, lets set up the Dataset.

Passing the script as the function allows us to also pass the url explictly.

Additionally, we will set the `sub_value` parameter here.

[5]:
```
ds = Dataset(script, url=url, skip=False, sub_value=10)
```

You should set up your dataset as normal, but explicitly pass `None` to the `function` argument

From here on out, it's as usual:

[6]:
```
for i in range(5):
    ds.append_run({"inner_value": i+1})

ds.run()

ds.wait(1, 10)

ds.fetch_results()

ds.results
```

```
appended run runner-0
appended run runner-1
appended run runner-2
appended run runner-3
appended run runner-4
Running Dataset
assessing run for runner dataset-9983aa87-runner-0... running
assessing run for runner dataset-9983aa87-runner-1... running
assessing run for runner dataset-9983aa87-runner-2... running
assessing run for runner dataset-9983aa87-runner-3... running
assessing run for runner dataset-9983aa87-runner-4... running
Transferring 12 Files... Done
Fetching results
Transferring 10 Files... Done
```

[6]:
```
['10', '20', '30', '40', '50']
```

This output is exactly what we'd expect!

Again, note that the output is in string format, so must be converted if you require it in some other format.

19.2 Other Output Methods

Should you require other output formats, you are able to output them to a file. This file should then be registered as an `extra_file_recv`.

CHAPTER

TWENTY

DATASET CLEANING

20.1 Starting Anew

By default, a `Dataset` will attempt to reinitialise at launch. In short, this means that it looks for a file that looks like itself. If it finds such a file, it will recreate itself from it.

For the user, this brings the benefits that your workflow is robust against restarts and data loss. However this does mean that datasets act like "accumulators", constantly gaining attributes and runners as you test.

There can come a point where you realise that something you set earlier could be causing problems (or simply isn't needed), this tutorial will run through some methods of dealing with these situations.

Lets start with the most basic case, `skip`:

[1]:
```
from remotemanager import Dataset

def f(inp):
    return inp

ds = Dataset(f,
             skip=False  # new option!
            )
```

When set to `False`, the `skip` argument will force the Dataset to start anew, and thus any variables that were stored are lost. This can also be done by deleting the database file (and in fact, this is what is done here internally, though then a new one is created). The filename is a combination of `name-dataset-uuid.yaml`. However if no `name` is set, it is omitted.

Deleting this file has the same effect as `skip`, though only once. It will be created along with the dataset.

The database filename can be seen using `Dataset.dbfile`

[2]:
```
ds.dbfile
```

[2]:
```
'dataset-9ebf1589.yaml'
```

You can also force this value to be whatever you want, but only at the dataset initialisation:

[3]:
```
ds_unique = Dataset(f,
                    dbfile = 'set_a_name_here',  # new option!
                    )

print(f'the database file for this dataset is now {ds_unique.dbfile}')
```

```
the database file for this dataset is now set_a_name_here.yaml
```

Now whenever you initialise a dataset with this filename, it will attempt to connect with that file.

20.2 Finer options

This is all well and good, but what if you don't want to blow up your dataset and start again? For example, you know that one of your runners is causing issues and needs to be removed. Well there are options for this too.

Lets append some runs, and experiment with removing them.

Lets also create a function to show us some information about our runners.

[4]:
```
for run in range(7):
    ds.append_run(args={'inp': run})

def print_runs():
    for r_id, runner in ds.runner_dict.items():
        print(f'{r_id}: {runner.short_uuid} | {runner.args}')
```

```
appended run runner-0
appended run runner-1
appended run runner-2
appended run runner-3
appended run runner-4
appended run runner-5
appended run runner-6
```

[5]:
```
print_runs()
```

```
runner-0: 655dd314 | {'inp': 0}
runner-1: 1360fc34 | {'inp': 1}
runner-2: e008c421 | {'inp': 2}
runner-3: eb3e73e3 | {'inp': 3}
runner-4: 2e945aa8 | {'inp': 4}
runner-5: e5e355b4 | {'inp': 5}
runner-6: 7082ed01 | {'inp': 6}
```

Now, we can look at all the ways of removing a run. We do this with `ds.remove_run(id)`. Here, `id` is a "smart" value, and can be `int`, `str` or `dict`, the function will perform slightly differently based on the input type:

- An `int` will be treated like a list index, and the runner at that id will be removed.
- A `dict` will be treated like arguments, and the runner with those args will be searched for.
- `str` is first checked against the runner names, and and is then checked against the `uuid` of each runner.
  - short and long uuids can be used (8 or 64 chars)

20.3 Runner Removal

Firstly, if you know the index of the runner within `ds.runners`, you can pass that id:

[6]:
```
ds.remove_run(0)
print_runs()
```

```
removed runner dataset-9ebf1589-runner-0
runner-1: 1360fc34 | {'inp': 1}
runner-2: e008c421 | {'inp': 2}
runner-3: eb3e73e3 | {'inp': 3}
runner-4: 2e945aa8 | {'inp': 4}
runner-5: e5e355b4 | {'inp': 5}
runner-6: 7082ed01 | {'inp': 6}
```

Runner 0 has dissappeared!

> **Note**
>
> This function always removes the runner at that index. So in this case if we call again with index 0, runner-1 would be removed, as it is the first.

Next, is the uuid. This can be found by printing the uuid of a runner you have access to:

[7]:
```
r_uuid = ds.runners[2].uuid

print(r_uuid)

r_short_uuid = ds.runners[3].short_uuid

print(r_short_uuid)
```

```
eb3e73e3dfaa213b1f58893de0340220cd39876c36ff7684a280e22bf51b85b6
2e945aa8
```

[8]:
```
ds.remove_run(r_uuid)
ds.remove_run(r_short_uuid)
print_runs()
```

```
removed runner dataset-9ebf1589-runner-3
removed runner dataset-9ebf1589-runner-4
runner-1: 1360fc34 | {'inp': 1}
runner-2: e008c421 | {'inp': 2}
runner-5: e5e355b4 | {'inp': 5}
runner-6: 7082ed01 | {'inp': 6}
```

We grabbed the uuids of runners at id 2 and 3, which in the runner list would be runner numbers 3 and 4 (as we removed 0). These two have also dissappeared.

If you don't know the id of the runner and don't have their uuids stored, you can remove by args. This attempts to match passed args with those that the runners have stored and will attempt to remove them. This is arguably the most flexible (and useful) method, though is less efficient than other approaches.

For example, if you append run with `{'inp': 6}`, you may remove that runner by calling:

[9]:
```
ds.remove_run({'inp': 6})
print_runs()
```

```
removed runner dataset-9ebf1589-runner-6
runner-1: 1360fc34 | {'inp': 1}
runner-2: e008c421 | {'inp': 2}
runner-5: e5e355b4 | {'inp': 5}
```

Looks like runner 6 (who had `inp: 6`) has also gone.

Finally, removing via `id` may be confusing if runs have already been removed, (i.e. you don't have a continuous, zero-indexed list). Thus, you can remove by the actual id by passing `remove_run("runner-{n}")`

[10]:
```
ds.remove_run('runner-5')
print_runs()
```

```
removed runner dataset-9ebf1589-runner-5
runner-1: 1360fc34 | {'inp': 1}
runner-2: e008c421 | {'inp': 2}
```

Leaving us with just runners 1 and 2.

This function also returns True or False depending on if it removed a runner or not:

[11]:
```
print('removed runner-2?:', ds.remove_run(1))
print('removed runner-3?:', ds.remove_run(2))
print('\nfinal runner list:')
print_runs()
```

```
removed runner dataset-9ebf1589-runner-2
removed runner-2?: True
removed runner-3?: False

final runner list:
runner-1: 1360fc34 | {'inp': 1}
```

20.3.1 Clearing Runners

There is one additional option for removing runners, and that's `wipe_runs`. This removes *all* runs from the dataset:

[12]:
```
ds.wipe_runs()
print(ds.runners)
```

```
[]
```

Persistence

All these changes are of course, saved to the database when performed, so be careful when using them. If we simulate restarting this notebook (or a different notebook that also uses this dataset), we will see no runners:

[13]:
```
ds = Dataset(f)
print(ds.runners)

ds.append_run({'inp': 2})
print(ds.runners)
```

```
[]
appended run runner-0
[dataset-9ebf1589-runner-0]
```

re-adding runners

Adding runners back to a dataset that has "holes" within its runner storage will cause no harm. Runners will be added to fill any missing spaces then continue as normal after that:

[14]:
```
for run in range(10):
    ds.append_run({'inp': run})
```

```
appended run runner-1
appended run runner-2
runner runner-0 already exists
appended run runner-3
appended run runner-4
appended run runner-5
appended run runner-6
appended run runner-7
appended run runner-8
appended run runner-9
```

[15]:
```
print_runs()
```

```
runner-0: e008c421 | {'inp': 2}
runner-1: 655dd314 | {'inp': 0}
runner-2: 1360fc34 | {'inp': 1}
runner-3: eb3e73e3 | {'inp': 3}
runner-4: 2e945aa8 | {'inp': 4}
runner-5: e5e355b4 | {'inp': 5}
runner-6: 7082ed01 | {'inp': 6}
runner-7: 863094b9 | {'inp': 7}
runner-8: 4bfe2419 | {'inp': 8}
runner-9: 64d5a225 | {'inp': 9}
```

Note here how we now have 10 runners as expected. Runner with `inp: 2` has been skipped, as it already exists.

20.4 Run Args

Now you know how to remove runners, what about run args? If you want to update a value, in most cases you can simply overwrite the value. Though if a run argument is causing issues, you can also delete it with the usual python syntax.

> **Note**
>
> Starting in version `0.10.0`, run_args are no longer accessible at the dataset level (i.e. `ds.mpi`), and must be accessed via the `run_args` property.

[16]:
```
ds.set_run_arg("mpi", 16)

print(ds.run_args["mpi"])
```

```
16
```

[17]:
```
print(ds.global_run_args)

del ds.run_args["mpi"]

print(ds.global_run_args)
```

```
{'skip': True, 'force': False, 'asynchronous': True, 'local_dir': 'temp_runner_local',
↪ 'remote_dir': 'temp_runner_remote', 'mpi': 16}
{'skip': True, 'force': False, 'asynchronous': True, 'local_dir': 'temp_runner_local',
↪ 'remote_dir': 'temp_runner_remote'}
```

20.5 Cleaning Directories

Added in version 0.5.9.

Too much clutter from testing? Dataset has some functions which help with deleting unwanted data:

- `dataset.wipe_local()` will attempt to delete any local directories
- `dataset.wipe_remote()` will attempt to delete any remote and run directories

If you really want to reset, dataset also provdes a `dataset.hard_reset()` function, which will do all of the above, delete the database file and then clear any runners. This essentially gives you a like-new dataset.

CHAPTER

TWENTYONE

TRANSPORT USAGE

In order to facilitate running on remote machines, remotemanager uses a file sending system internally referred to as `Transport`. For most use cases, you will not need to interface with these structures, however you may find their functions helpful for controlling files.

21.1 Transport types

`Transport` itself is not a useful structure, and you won't get very far by using it in its raw form. It exists to give a common set of methods to all subclasses. The primary subclass is `transport.rsync`

[1]:
```
# dev note: be careful editing this tutorial,
# it's _very_ sensitive to files and folders already existing,
# they must be cleared prior to any run, else it will cause the CI to fail

from remotemanager.transport import rsync
```

Transport functions with a queue system, and holds a concept of `push` and `pull`. First, initialise your transport class with the arguments that you would like rsync to use

[2]:
```
tr = rsync(flags='auv')
```

To actually transfer files, you must first queue them. Transport entities consider your current machine as the local or origin point, and the destination as the remote or target point. First, lets create some folders and files for demonstration:

[3]:
```
from remotemanager import URL

url = URL()

url.cmd('rm -r temp_trn_local', raise_errors=False)
url.cmd('rm -r temp_trn_remote', raise_errors=False)
url.utils.mkdir('temp_trn_local')
url.utils.mkdir('temp_trn_remote')

url.utils.touch('temp_trn_local/send_me')
url.utils.touch('temp_trn_local/send_me_also')

url.utils.touch('temp_trn_remote/fetch_me')
url.utils.touch('temp_trn_remote/fetch_me_too')
url.utils.touch('temp_trn_remote/fetch_me_differently')
```

[3]:
```

```

[4]:
```
print(url.utils.ls('temp_trn_local'))
```

```
['send_me', 'send_me_also']
```

[5]:
```
print(url.utils.ls('temp_trn_remote'))
```

```
['fetch_me', 'fetch_me_differently', 'fetch_me_too']
```

Now we have 2 files on our "local" machine we want to send, and also 3 files on our "remote" machine that we need to fetch. Lets start with pushing. To do this, we need to use the method `queue_for_push`

This takes the format of `files`, `local`, `remote`:

[6]:
```
tr.queue_for_push(['send_me', 'send_me_also'], 'temp_trn_local', 'temp_trn_remote')
```

With this done, we can see the transferrs that are ready to occur. Either by accessing the `transfers` property, or using the `print_transfers` method, which formats it for you

[7]:
```
tr.transfers
```

[7]:
```
{'/home/test/remotemanager/docs/source/tutorials/temp_trn_local/>temp_trn_remote/': [
↪'send_me',
  'send_me_also']}
```

[8]:
```
tr.print_transfers()
```

```
transfer 1:
origin: /home/test/remotemanager/docs/source/tutorials/temp_trn_local/
target: temp_trn_remote/
        (1/2) send_me
        (2/2) send_me_also
```

Here we can see a single transfer that is ready to occur, which represents *one* rsync call. Before executing, we can see the commands to be executed by calling the `transfer` method with `dry_run=True`

[9]:
```
tr.transfer(dry_run=True)
```

[9]:
```
[rsync -auv /home/test/remotemanager/docs/source/tutorials/temp_trn_local/{send_me,
↪send_me_also} temp_trn_remote/]
```

This looks good, lets go:

[10]:
```
tr.transfer()
```

```
Transferring 2 Files... Done
```

Now check the "remote" folder to see what it looks like:

[11]:
```
url.utils.ls('temp_trn_remote')
```

[11]:
```
['fetch_me', 'fetch_me_differently', 'fetch_me_too', 'send_me', 'send_me_also']
```

Seems that the files have been sent as expected

21.2 More complex movement

You may be aware that rsync cannot handle a many-to-many situation. This is the greatest strength of the `Transport` systems. The queuing necessity means that prior to a command execution, logic can be applied and the *minimum* amount of calls can be made.

In the following example we have 3 files to fetch from the "remote". Lets assume that we want one to go to a different folder, `Transport` handles this for you:

[12]:
```
tr.queue_for_pull(['fetch_me', 'fetch_me_too'], 'temp_trn_local', 'temp_trn_remote')

url.utils.mkdir('temp_trn_local_different')  # create a different target dir for this␣
↪file
tr.queue_for_pull('fetch_me_differently', 'temp_trn_local_different', 'temp_trn_remote
↪')
```

> **Note**
>
> Pay close attention to the folder ordering. While we are *pulling* from the remote, `Transport` itself is still a connection from the "local" to the "remote". Hence, the folder order **does not change**.

Lets look at our transfers:

[13]:
```
tr.print_transfers()
```

```
transfer 1:
origin: temp_trn_remote/
target: /home/test/remotemanager/docs/source/tutorials/temp_trn_local/
        (1/2) fetch_me
        (2/2) fetch_me_too
transfer 2:
origin: temp_trn_remote/
target: /home/test/remotemanager/docs/source/tutorials/temp_trn_local_different/
        (1/1) fetch_me_differently
```

and commands

[14]:
```
tr.transfer(dry_run=True)
```

[14]:
```
[rsync -auv temp_trn_remote/{fetch_me,fetch_me_too} /home/test/remotemanager/docs/
↪source/tutorials/temp_trn_local/,
 rsync -auv temp_trn_remote/fetch_me_differently /home/test/remotemanager/docs/source/
↪tutorials/temp_trn_local_different/]
```

Now execute, and look into the folders

[15]:
```
tr.transfer()
```

```
Transferring 3 Files in 2 Transfers... Done
```

[16]:
```
print(url.utils.ls('temp_trn_local'))
```

```
['fetch_me', 'fetch_me_too', 'send_me', 'send_me_also']
```

[17]:
```
print(url.utils.ls('temp_trn_local_different'))
```

```
['fetch_me_differently']
```

Looks like all our files have been brought back to the correct place!

21.3 Bash Compatibility Mode

All Transports can be provided with an argument `dir_mode`, either at init (when calling `rsync(..., dir_mode=True)`, or on the `transfer(dir_mode=True)`. In most cases, you will not have access to the actual `transfer` call, so it is best to set it at init, or update it via the `dir_mode` property if needed.

If `True`, any transfer will have an extra step added where the target files are first copied to a temporary directory, then transferred via *. This avoids using bash brace expansion to generate the command, who's behaviour can change on some machines.

21.4 Progress

If you have used rsync before, you may be aware that there is a `--progress` option. This prints a continuous update stream as the files are transferred.

When creating an `rsync` object, you can enable this for your terminal by setting `progress=True` on the initial call.

We can demonstrate this here using a `Dataset`

[18]:
```
from remotemanager import Dataset
from remotemanager.transport import rsync

def f(i):
    return i

ds = Dataset(f, transport=rsync(progress=True), skip=False)

ds.append_run({"i": 1})
ds.append_run({"i": 2})
```

```
appended run runner-0
appended run runner-1
```

[19]:
```
ds.run()
```

```
Running Dataset
assessing run for runner dataset-a6e26708-runner-0... running
assessing run for runner dataset-a6e26708-runner-1... running
Transferring 7 Files
sending incremental file list
dataset-a6e26708-master.sh

            198 100%    0.00kB/s    0:00:00 (xfr#1, to-chk=6/7)
dataset-a6e26708-repo.py

          5.34K 100%    5.09MB/s    0:00:00 (xfr#2, to-chk=5/7)
dataset-a6e26708-repo.sh

            850 100%  830.08kB/s    0:00:00 (xfr#3, to-chk=4/7)
dataset-a6e26708-runner-0-jobscript.sh

            161 100%  157.23kB/s    0:00:00 (xfr#4, to-chk=3/7)
dataset-a6e26708-runner-0-run.py

          1.03K 100% 1008.79kB/s    0:00:00 (xfr#5, to-chk=2/7)
dataset-a6e26708-runner-1-jobscript.sh
```

(continues on next page)

(continued from previous page)

```
            161 100%  157.23kB/s    0:00:00 (xfr#6, to-chk=1/7)
dataset-a6e26708-runner-1-run.py

          1.03K 100% 1008.79kB/s    0:00:00 (xfr#7, to-chk=0/7)

sent 9.28K bytes  received 149 bytes  18.86K bytes/sec
total size is 8.77K  speedup is 0.93
Done
```

You can, of course, override this behaviour with `verbose=False`

[20]:
```
ds.run(verbose=False, force=True)
```

21.5 Advanced Usage

Contrary to the note regarding the folder order, there exists one further method which inverts the behaviour of the folder ordering. In fact both queueing methods internally call this method, acting as formatters for its arguments.

This method is not intended to be called by the user, but is left as a non-private function for those who prefer its behaviour.

Instead of passing `files`, `local`, `remote`, you must pass `files`, `origin`, `target`, `mode`. This takes a file-centric view, and thus for a pull, the `origin` is the remote dir. The `mode` simply tells `Transport` where to put the structures for connecting to the remote, and can either be "push" or "pull":

[21]:
```
tr.add_transfer('fetch_me', 'temp_trn_remote', 'temp_trn_local', 'pull')
```

[22]:
```
tr.transfer(dry_run=True)
```

[22]:
```
[rsync -auv temp_trn_remote/fetch_me /home/test/remotemanager/docs/source/tutorials/
↪temp_trn_local/]
```

As you can see, the transfer is created in the intended way, despite the "swapped" folders. You may deem this to be a more sensible use case, and prefer to use it. As the queue functions exist soley to call this function, this should remain a safe method of use for those that wish to use it.

21.5.1 Naming Conventions

For reference, the below table sums up the naming convention within the source, for those who want to do further reading:

name	meaning
local	"local" folder, regardless of mode of use
remote	"remote" folder, regardless of mode of use
origin	starting folder for the files; the **first** folder in an rsync command
target	destination folder for the files; the **second** folder in an rsync command

Note

Be aware of the argument expansion limitation that exists with rsync versions below version 3. If you get errors during transfer, be sure to check `rsync --version` >= 3.

CHAPTER

TWENTYTWO

ADVANCED LOGGING TECHNIQUES

remotemanager comes with an inbuilt logging system who's Handler is accessible via the `remotemanager.Logger` attribute

these functions are limited, however useful for debugging and it may be helpful to have a block at the top of any notebooks in testing that sets some variables

A common use case is to place a block at the start of a notebook, dictating the intended log path and the parameters, as follows:

[1]:
```
import remotemanager

remotemanager.Logger.level = 'debug'
remotemanager.Logger.path = 'notebook_to_test_feature'
remotemanager.Logger.overwrite = True
```

22.1 Properties to set

Below is a quick summary of the available properties. See the Logging documentation for full info

As these are properties, they can be set or queried. For example, set and read the logging level with

[2]:
```
remotemanager.Logger.level = 'warning'
remotemanager.Logger.level
```

[2]:
```
'WARNING'
```

22.1.1 `Logger.level`

This allows you to set the logging level which will be used. Defaults to WARNING. Possible options (in order of increasing verbosity) are:

Mode	Adds to the logfile
Critical	Fatal errors only
Error	Damaging errors
Warning	All errors and warnings
Important	Noteworthy non-errors
Info	Important runtime info
Runtime	Runtime information
Debug	All runtime information

22.1.2 Logger.path

Setting the path will firstly point the logger at the intended filepath, and also attempt to delete the previous log, if it is empty. If called before any other actions are taken, essentially moves the log to a new location.

Note

Logs are enforced to be in yaml format, in keeping with the package theme of using yaml files where possible. This has the added benefit of yaml formatting and features

Warning

If the default log path `log.yaml` already exists and has content in it, it will not be deleted and a warning will be raised. This does not affect downstream runtime, but exists as a safety measure to not delete any logs you may wish to keep with a default name

22.1.3 Logger.overwrite

Setting this to True will set the write mode for the logs to `w`, overwriting any previous log

22.1.4 Logger.write_mode

Similarly to the previous `overwrite` property, only allowing you to directly set the write mode of the logging Handler

22.2 Useful functions

22.2.1 Logger.refresh()

Calling this function will force delete and re-create the logfile

22.2.2 Logger.debug()

This function, along with its companions, allows direct access to the log. Calling this will place a log message in the log under the caller `EXTERNAL`. For example:

[3]:
```
remotemanager.Logger.info('this will a log message at the level "info"')
remotemanager.Logger.warning('this will a log message at the level "warning"')
```

These exist for all log levels as listed above

CHAPTER

TWENTYTHREE

VERBOSE LEVELS

Many operations within `remotemanager` have a concept of `Verbose`.

At the base level, this can be True or False, but you may also specify higher levels with integer values.

`verbose` can be specified at the topelevel Dataset, or for the individual calls.

It functions much like a `run_arg`, in that there is a hierarchy to it.

Basically, the more specific the call, the higher priority it has.

For example, if we set `verbose=False` at the `Dataset` level, it will disable output for that Dataset.

However if we then call `run(verbose=True)`, that verbose option local to the run call will take priority over the global setting.

23.1 Default Level

By default, verbose is set to `True` or `1`. This prints the basic runtime information you have already seen in other tutprials.

Here, we shall give a brief overview of what the different levels look like.

> **Note**
>
> In short, higher verbose number = more output in the terminal.

23.2 Muting Output

The most useful feature of verbose is being able to disable the output.

In some situations a lot of information can be printed all at once and it can be beneficial to disable the output for that operation.

[1]:
```
from remotemanager import Dataset

def f(inp):
    return f'inp was {inp}'

ds = Dataset(f, verbose=False, skip=False)
```

[2]:
```
ds.append_run(args={'inp': 'first input'})
ds.append_run(args={'inp': 'second input'})
```

```
[3]: ds.run()
```

```
[4]: ds.run()
```

```
[5]: ds.fetch_results()
```

```
[6]: ds.results
[6]: ['inp was first input', 'inp was second input']
```

23.3 Increasing Verbosity

Verbose levels go the other way, you can increase the output by specifying a higher integer level.

> **Note**
>
> For debugging, the logging functionalities will be more beneficial.

```
[7]: ds = Dataset(f, verbose=3, skip=False)
```

```
Dataset initialised
new url created with url details:
   host: localhost
   port: None
   user: None
Creating a fresh Dataset w/ database at dataset-0ae54639.yaml
```

```
[8]: ds.append_run(args={'inp': 'first input'})
ds.append_run(args={'inp': 'second input'})
```

```
appended run runner-0
appended run runner-1
```

```
[9]: ds.run()
```

```
Running Dataset
assessing run for runner dataset-0ae54639-runner-0... running
assessing run for runner dataset-0ae54639-runner-1... running
Transferring 6 Files... Done
```

```
[10]: ds.run()
```

```
Running Dataset
assessing run for runner dataset-0ae54639-runner-0... skipping already submitted run
assessing run for runner dataset-0ae54639-runner-1... skipping already submitted run
```

```
[11]: ds.fetch_results()
```

```
Fetching results
Checking Runner states
        dataset-0ae54639-runner-0... Completed, pulling result, pulling error,␣
↪pulling extras
        dataset-0ae54639-runner-1... Completed, pulling result, pulling error,␣
↪pulling extras
Transferring 4 Files... Done
```

[12]:
```
ds.fetch_results()
```

```
Fetching results
Checking Runner states
        dataset-0ae54639-runner-0... Already marked Satisfied, no work needed
        dataset-0ae54639-runner-1... Already marked Satisfied, no work needed
No Transfer Required
```

[13]:
```
ds.results
```

[13]:
```
['inp was first input', 'inp was second input']
```

23.4 Setting for individual objects

As was hinted at earlier, we can also set specific levels for different objects.

Dataset, URL and Runners all store their own verbose levels.

Dataset and URL can be set on creation.

URL will inherit from Dataset if not set.

Runner is set at `append_run`.

[14]:
```
from remotemanager import URL

no_verbose_url = URL(verbose=2)

ds = Dataset(f, verbose=1, url = no_verbose_url, skip=False)
```

```
new url created with url details:
   host: localhost
   port: None
   user: None
```

[15]:
```
ds.append_run(args={'inp': 'first input'}, verbose = 0)
ds.append_run(args={'inp': 'second input'}, verbose = 0)
```

```
appended run runner-0
appended run runner-1
```

23.4.1 Quiet Mode

Since Runners inherit their verbose level from the `append_run` call, there is a special `quiet` argument here.

This is for the case where you want to silently append runners, but not have the runners themselves be silent.

[16]:
```
ds = Dataset(f, skip=False)
```

[17]:
```
ds.append_run(args={'inp': 'first input'}, quiet=True)
ds.append_run(args={'inp': 'second input'}, quiet=True)
```

[18]:
```
ds.run()
```

```
Running Dataset
assessing run for runner dataset-0ae54639-runner-0... running
assessing run for runner dataset-0ae54639-runner-1... running
Transferring 6 Files... Done
```

```
[19]: ds.fetch_results()
```

```
Fetching results
Transferring 4 Files... Done
```

```
[20]: ds.fetch_results()
```

```
Fetching results
No Transfer Required
```

```
[21]: ds.results
```

```
[21]: ['inp was first input', 'inp was second input']
```

23.5 Distinction between Logger.level, verbose and quiet

`Logger.level`, `verbose` and `quiet` are all separate entities. For example, if you have `Logger.level` set to `debug`, and `verbose=0`. The logfile will still contain messages at `debug` level, but nothing will be printed to the terminal.

`Quiet` simply mutes any printing (not logging) for the duration of that call.

CHAPTER

TWENTYFOUR

THE FLAGS MODULE

24.1 Customising a Transfer

`remotemanager` comes with a `Flags` module which is used internally to handle command line flags. It is used in the `Transport` modules, and can help you adjust your file transfers as needed.

Lets set up a dataset and query its transport class to see this in a real world situation:

[1]:
```
from remotemanager import Dataset

def f():
    return

dataset = Dataset(function = f,
                  local_dir = 'temp_local',
                  remote_dir = 'temp_remote',
                  skip=False)
```

[2]:
```
print(dataset.transport.__class__)
print('Transport commandline flags are:', dataset.transport.flags)
```

```
<class 'remotemanager.transport.rsync.rsync'>
Transport commandline flags are: -auvh
```

Here we can see that by default, the dataset creates an `rsync` transport with the default flags `auv`. Lets say that we want to change the flags to make rsync operate quietly. We need to swap the `v` with a `q`. If the flags are simple, or want to set them statically, you can set them:

[3]:
```
dataset.transport.flags = 'auq'
print('Transport commandline flags are:', dataset.transport.flags)
```

```
Transport commandline flags are: -auq
```

Though if you're in a situation where they could differ, you can also modify them in place:

[4]:
```
dataset.transport.flags = 'auv'  # reset the flags for this test

dataset.transport.flags -= 'v'
dataset.transport.flags += 'q'
print('Transport commandline flags are:', dataset.transport.flags)
```

```
Transport commandline flags are: -auq
```

Flags also handles verbose arguments (ones preceeded by `--`), and prints them as expected:

[5]:
```
dataset.transport.flags += '--checksum --progress --test'
print('Transport commandline flags are:', dataset.transport.flags)
```

```
Transport commandline flags are: -auq --checksum --progress --test
```

We can also do the same inplace modification here, just be sure to prefix with `--`:

[6]:
```
dataset.transport.flags -= '--test'
print('Transport commandline flags are:', dataset.transport.flags)
```

```
Transport commandline flags are: -auq --checksum --progress
```

In fact, you can also prefix any singular arguments with `-`, as this is treated as a special case by the module: When assessing a string, it will count the amount of `-`. If none are found, it assumes the single case.

[7]:
```
dataset.transport.flags += '-v'
print('Transport commandline flags are:', dataset.transport.flags)
```

```
Transport commandline flags are: -auqv --checksum --progress
```

24.2 Direct Setting

It is also possible to set the flags directly when importing. You can see more information on this at the Changing Transport tutorial

CHAPTER

TWENTYFIVE

FUNCTION TOOLS

There are some additional tools that are available within a runner.

The first of these is `loaded`, which is available for dependencies. The second of which is `manifest`, which allows access to the `manifest` file that's used to track runner states.

25.1 loaded

Loaded was covered in the dependency tutorials, but as a quick refresher:

Whenever you create two functions that are intended to be linked together, you can access the parent's output from within the children using the `loaded` object. Basically, `loaded` is as if you had called the function right there.

This means that if your parent function returns more than one item, you should be careful to collect the correct one in the child. Lets say we have a function that returns a result, and a status:

[1]:
```
from remotemanager import Dataset

def f1(x):
    return x, True

def f2(y):
    return loaded[0] + y
```

[2]:
```
ds1 = Dataset(f1, name="a", skip=False, local_dir="temp_dep_local", remote_dir="temp_
↪dep_remote")
ds2 = Dataset(f2, name="b", skip=False, local_dir="temp_dep_local", remote_dir="temp_
↪dep_remote")

ds1.set_downstream(ds2)

ds2.append_run({"x": 1, "y": 10})

ds2.run()

ds2.wait(1, 5)

ds2.fetch_results()

ds2.results
```

```
appended run runner-0
appended run runner-0
Running Dataset
assessing run for runner a-526bebb1-runner-0... running
assessing run for runner b-233f50b5-runner-0... running
```

(continues on next page)

(continued from previous page)

```
Transferring 6 Files... Done
Fetching results
Transferring 1 File... Done
```

[2]:
```
[11]
```

25.2 manifest

The `manifest` is a file that allows the runners to store their status, for later collection. As it acts somewhat like a logfile, you also have the ability to log to it.

This is done by accessing the `write` method:

[3]:
```
def function_with_log(inp):
    manifest.write(f"function was called with input: {inp}")
```

[4]:
```
ds = Dataset(function_with_log, skip=False)

ds.append_run({"inp": True})

ds.run()

ds.wait(1, 10)

ds.fetch_results()

ds.results
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-cd130c74-runner-0... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

[4]:
```
[None]
```

Here, our function returns nothing, but we can still access what was logged at the `history` attribute of the runner:

[5]:
```
for time, log in ds.runners[0].history.items():
    print(time, log)
```

```
2024-06-24 10:50:54/0 created
2024-06-24 10:50:54/1 staged
2024-06-24 10:50:54/2 submit pending
2024-06-24 10:50:54/3 submitted
2024-06-24 10:50:54/4 started
2024-06-24 10:50:54/5 function was called with input: True
2024-06-24 10:50:54/6 completed
2024-06-24 10:50:55/0 completed
2024-06-24 10:50:55/1 satisfied
```

Any newlines will also be respected:

[6]:
```
ds.append_run({"inp": "\nthis demonstrates a newline"})

ds.run()
```

(continues on next page)

(continued from previous page)

```
ds.wait(1, 10)

ds.fetch_results()

ds.results
```

```
appended run runner-1
Running Dataset
assessing run for runner dataset-cd130c74-runner-0... ignoring run for successful␣
↪runner
assessing run for runner dataset-cd130c74-runner-1... running
Transferring 4 Files... Done
Fetching results
Transferring 1 File... Done
```

[6]:
```
[None, None]
```

[7]:
```
for time, log in ds.runners[1].history.items():
    print(time, log)
```

```
2024-06-24 10:50:55/0 created
2024-06-24 10:50:55/1 staged
2024-06-24 10:50:55/2 submit pending
2024-06-24 10:50:55/3 submitted
2024-06-24 10:50:55/4 started
2024-06-24 10:50:55/5 function was called with input:
this demonstrates a newline
2024-06-24 10:50:55/6 completed
2024-06-24 10:50:56/0 completed
2024-06-24 10:50:56/1 satisfied
```

CHAPTER

TWENTYSIX

CHANGING THE TRANSPORT

Similar to how you are able to change the serialiser by importing a new one and passing it to Dataset, you are also able to do this with Transport.

26.1 Use Cases

The main reason to swap transport, is if the default `rsync` does not work for your system. This can either be related to the remote machine, the connection, or an outdated version.

> **Important**
>
> `remotemanager` requires `rsync --version >= 3.0.0`. MacOS devices may run an outdated version. To fix this, you can either update your install (slower, but permanent fix), or swap to `scp` (fast, but is required for each `Dataset`).

Even if you have no issues, it is possible to customise the transport further by setting Flags directly. This is an alternative method to that shown in the flags tutorial.

26.2 Importing

Just like with `serialdill`, `serialjson`, etc., you may import from the available Transport methods:

- `rsync`
- `scp`
- `cp`

Of these, `cp` is less useful as it is unable to connect to external machines. It is provided for the edge case where you require no remote connection and the machine has no rsync or scp. And to provide a very simple template for creating your own Transport.

To start, we can set up a run just as normal. The transport is a **drop in** replacement, having no effect on the Dataset other than the command that actually gets used to send/retrieve data.

[1]:
```
from remotemanager import Dataset
```

[2]:
```
def function(x, y):
    return x * y
```

Since `rsync` is default, lets swap to `scp`

[3]:
```
from remotemanager.transport import scp
```

[4]:
```
ds = Dataset(
    function,
    skip = False,
    transport = scp(),  # new option!
)
```

> **Note**
>
> Like the serialiser, and URL, the transport object must be instantised ("called") at some point post-import.

[5]:
```
ds.append_run({"x": 21, "y": 2})
ds.run()
ds.wait(1, 10)
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-991e1c92-runner-0... running
Transferring 5 Files... Done
```

[6]:
```
ds.fetch_results()
ds.results
```

```
Fetching results
Transferring 2 Files... Done
```

[6]:
```
[42]
```

26.2.1 Verification

Right now, it looks like nothing has changed, we have to do some digging to see if it worked.

A quick way is to check the `transport` property

[7]:
```
ds.transport
```

[7]:
```
<remotemanager.transport.scp.scp at 0x706a7e176790>
```

That reads, `scp`, so it's the right module at least. But we want to see some *commands*. Lets search the cmd_history for commands containing `scp`:

[8]:
```
for cmd in ds.url.cmd_history:
    if "scp" in cmd.sent:
        print(cmd.sent)
        break
```

```
scp -r /home/ljbeal/Work/Devel/remotemanager/docs/source/tutorials/temp_runner_local/
↪{dataset-991e1c92-master.sh,dataset-991e1c92-repo.py,dataset-991e1c92-repo.sh,
↪dataset-991e1c92-runner-0-jobscript.sh,dataset-991e1c92-runner-0-run.py} temp_
↪runner_remote/
```

And there we have our first scp call, sending data from local to remtote dirs.

26.3 Flags

As mentioned at the top, it is possible to directly set the flags of the transport at the initialisation, using the `flags` keyword:

[9]:
```
ds = Dataset(
    function,
    skip = False,
    transport = scp(flags="-v"),  # new option!
)
```

[10]:
```
ds.transport.flags
```

[10]:
```
-v
```

26.4 Custom Transport

Just like with Serialiser, it is possible to create your own transport class.

This can be done by subclassing the transport module and adding the necessary overrides (usually just the `cmd` method).

26.4.1 `cmd`

When overriding the `cmd` method of `Transport`, there is a pattern to follow.

The docstring of the base level method explains this in detail. Found here.

But in short, the function should return a valid command in string form, and accept two arguments `primary` and `secondary`. These are both strings.

26.4.2 `primary`

This argument will come "preformatted" in bash-syntax. For example `directory_name/{file1,file2,file3,...,fileN}`

CHAPTER

TWENTYSEVEN

JUBE INTEROPERABILITY

JUBE is a benchmark automation tool developed by the Julich Supercomputing Centre.

You can find the github repo here: https://github.com/FZJ-JSC/JUBE

And the website here: https://apps.fz-juelich.de/jsc/jube/docu/index.html

27.1 JUBETemplates

JUBE defines the machines on which it operates via "platform.xml" files and job templates. `remotemanager` provides iteroperability with these definitions via the `JUBETemplate` module, which is avaiable at `remotemanager.JUBEInterop`

This object provides a modified `from_repo` which is able to pull these files automatically.

> **Note**
>
> By default `from_repo` is poined at the JUBE4MaX repository, however you can change this via the `repo` argument. (Point it at the root of the repo).

You should then specify a `path` to the directory containing the platform and template files.

The target for these files can be changed by updating `platform_name` and `template_name`

Since the file names are likely to clash when pulling multiple machines, they are stored by default in a directory extracted from the name. You can also set this parameter with `local_dir`.

[2]:
```
from remotemanager.JUBEInterop import JUBETemplate

template = JUBETemplate.from_repo(path="max-inputs/platforms/cineca/leonardo/booster",
↪ local_dir="temp_platform_store")
```

```
searching for platform.xml & submit.job at https://gitlab.com/max-centre/JUBE4MaX/-/
↪raw/develop/max-inputs/platforms/cineca/leonardo/booster
Grabbed file 'temp_platform_store/platform.xml'
Grabbed file 'temp_platform_store/submit.job'
```

After a successful file collection, you will now be able to generate jobscripts using this computer. Lets set some basic arguments and print a sample.

[3]:
```
template.accountno = "test_acc"
template.nodes = 24
template.ncpus = 128
template.ncores = 128
template.taskspernode = 32

template.executable = "bigdft"
```

[4]:
```
script = template.script()
print(script)
```

```
#!/bin/bash
#SBATCH --nodes=24
#SBATCH --ntasks-per-node=32
#SBATCH --cpus-per-task=1
#SBATCH --gres=gpu:4
#SBATCH --time=24:00:00
#SBATCH --exclusive
#SBATCH --account=#ACCOUNT_NO#
#SBATCH --partition=boost_usr_prod
#SBATCH --qos=normal

module purge
export OMP_NUM_THREADS=1

scontrol show jobid -dd $SLURM_JOB_ID > scontrol.out




sacct -j $SLURM_JOB_ID --long > sacct.out

touch #READY#
```

27.1.1 Parameterisation

Since these platforms are intended to be used within the JUBE infrastructure, you will need to be careful to set the correct parameters. If you're not sure which parameters to set, you can check the downloaded files, or print the `arguments` property.

[5]:
```
print(template.arguments)
```

```
['jube_benchmark_name', 'queue', 'timelimit', 'starter', 'args_starter', 'measurement
↪', 'outlogfile', 'errlogfile', 'executable', 'args_executable', 'touch $ready_file',
↪ 'nodes', 'threadspertask', 'tasks pernode', 'tasks persocket', 'cpuspertask', 'pe',
↪'gres', 'accountno', 'qos', 'modules', 'preprocess', 'postprocess', 'wrapname',
↪'wrappre', 'wrappost', 'ready_file', 'make', 'cc', 'cflags', 'mpi_cc', 'mpi_cxx',
↪'mpi_f90', 'mpi_f77', 'load_module', 'mapping', 'submit', 'submit_script', 'shared_
↪folder', 'shared_job_info', 'nsocket', 'nodecpus', 'ncores', 'threads', 'env',
↪'ngpus', 'tasks', 'memnodemachine', 'minmem']
```

27.2 Missing Parameters

By default, when encountering a missing argument for a substitution, `BaseComputer` will delete the whole line. This is based on the rationale that a jobscript `#PRAGMA flag=#argument#` is best deleted if empty, as it will cause the job to fail.

`JUBETemplate` uses the "local" empty behaviour for substitutions by default. This means that any missing parameters are removed "locally", not globally.

If you look at the earlier example, and compare it with the template, you can see many examples of this. Especially on the `mpirun` call, arguments are missing, however the line is still present.

27.3 Temporary Values

Just like BaseComputer is able to accept "temporary" values within the `script()` method, so is `JUBETemplate`

[6]:
```
print(template.script(nodes=128))
```

```
#!/bin/bash
#SBATCH --nodes=128
#SBATCH --ntasks-per-node=32
#SBATCH --cpus-per-task=1
#SBATCH --gres=gpu:4
#SBATCH --time=24:00:00
#SBATCH --exclusive
#SBATCH --account=#ACCOUNT_NO#
#SBATCH --partition=boost_usr_prod
#SBATCH --qos=boost_qos_bprod

module purge
export OMP_NUM_THREADS=1

scontrol show jobid -dd $SLURM_JOB_ID > scontrol.out




sacct -j $SLURM_JOB_ID --long > sacct.out

touch #READY#
```

Just like the `BaseComputer` temporary values, these exist for only a single run.

[7]:
```
nodes = template.script().split("\n")[1]

print(nodes)
```

```
#SBATCH --nodes=24
```

The following page contains a compilation of patch notes:

CHAPTER

TWENTYEIGHT

VERSION HISTORY

The following is a list of patch notes for the previous three minor updates.

Patch notes for updates not on this list can be found on the releases page on GitLab.

28.1 0.13.x

> **Note**
>
> Despite the jump in versions, 0.13.x is a direct descendant of 0.11.x. It is a migration of the functional 0.11.20 engine onto the 0.12.5 codebase. In theory, this ensures non regression of features and functionality from 0.12, along with some unreleased 0.12.5 fixes.

28.1.1 0.13.7

[fixes]

- Fixed an issue related to `run_dir` and `Dependency`

[refactor]

- Refactored out the `DIR_{UUID}` syntax from jobscripts

28.1.2 0.13.6

[docs]

- Added a page dedicated to changing the `Transport` module
- Minor cleanup and readability improvements

[feature]

- `rsync` now checks that the locally installed version is high enough
- Added `URL.reset_cmd_history`
- `Computer.from_yaml` can now override settings
- `TrackedFile` now provides the `chmod` method, applies the given permissions to the local copy

[fixes]

- `URL` now resets the cmd history prior to a `deepcopy` to prevent a crash
- Fixed a bug where multi line functions would be incorrectly stored if they also had inner functions
- Fixed a bug where children would search in an invalid location for parent results

[refactor]

- Transferred internal script files are now made executable by chmod
- `Dataset` no longer logs when a `Database` is deleted, or cannot be deleted because it does not exist

[CI]

- Removed the migrated `storage` and `decorator` notebook test suites

28.1.3 0.13.5

[features]

- It is now possible to specify (and access) iterables such as dicts and lists in Templates
- Templates can now handle quoted `:` characters
- Templates now allow escaping characters with ``

[fixes]

- Fixed a bug caused by concatenating ints to a string using _ as the separator
- Fixed an instance where querying *Substitution.shortform_op* would cause a crash

[refactor]

- `Script` now recursively links values

28.1.4 0.13.4

[features]

- You can now disable the timeout of a cmd call by setting `timeout` to 0, `False` or a negative number
- `url.cmd(...)` now supports the `with` context. Calling `cmd` in this manner ensures that a process is killed in the event of an exception. Useful in combination with `asynchronous`
- `CMD` objects can now kill other PIDs, use with care
- `URL` can now create a tunnel to a host using `url.tunnel(local_port, remote_port)`

[refactor]

- `URL` is no longer serialised by Dataset
- `Dataset.from_file` now requires a url, and will print a warning if not provided
- `Unloaded` has been removed, objects are now expected to function without placeholders

[fixes]

- `URL.ping` and `URL.test_connection` now uses a randomised files and dirs
- `URL.gethome()` no longer returns `None`
- `ssh_prepend` can now be set in `URL` initialisation

[CI]

- Standard testing suite now uses the `logical` flag, making use of threading
- The `FunctionCache` and `Connection` notebook suites have now been migrated into the standard suite

28.1.5 0.13.3

[refactor]

- Increased scope of `replaces` and `requires` in templates
- `Computer.pack` now uses a more readable format
- `Dataset` now handles file dumping, rather than relying on `SendableMixin`
- `JUBEInterop` module updated to use the `Computer` module

[feature]

- `replaces` and `requires` will now parse a comma separated string into a list of values
- Added `Computer.to_yaml` and `from_yaml` which uses the new readable `pack`
- `Computer` now also accepts a path to a template file

[tests]

- `Computer` template functionality is now checked for regression
- Refactored tests that tested `BaseComputer` to now test `Computer`
- Added tests for `JUBETemplate`
- Serialisation tests migrated into standard
- Transport tests migrated into standard

28.1.6 0.13.2

[fixes]

- fixed a bug that could occur when setting `extra` on `Dataset` while using a `Computer`

28.1.7 0.13.1

[refactor]

- `remotemanager.logging` has been renamed to `remotemanager.logging_utils`. This should prevent some potential name clashes with the inbuilt `logging` module
- Templates are now handled by a dedicated `Script` class
- Deprecate `BaseComputer`

[feature]

- `Dataset` now accepts a `Script` entity as the `function` argument, generating a script (executed by `url.shell`) using the arguments as input
- Added a `Computer` class that handles Templates more elegantly than `BaseComputer`
- `Substitution` objects now have a `hidden` parameter that prevents them from being added to the jobscript
- `Substitution` objects now have an `empty_treatment` parameter that dictates how they are treated when empty
- `Substitution` objects now have a `static` parameter that forces non-evaluation

[tests]

- The standard test suite is now also tested with Python 3.7

[docs]

- Updated documentation surrounding templates and jobscripts

- Updated documentation surrounding non-function runs
- Moved deprecated documentation to a "legacy" directory. It's still available, but is untested and will be removed eventually

28.1.8 0.13.0

[features]

- `Version` can now "imply" the patch (and minor) versions if not provided (defaults to 0)
- add a `match` functionality to `Version`, checking against a generic version string like `0.13.x`
- `Database` now warns using more fine-grain version incompatibilities
- `Function` now has a `return_annotation` property

[fixes]

- `Version` now has a rigid output format for properties
- Templates can now properly handle boolean operations (extends to `min`, `max`, etc.)
- You can now specify `#extra#` within templates
- `Dependencies` no longer uses a legacy run method
- `Function.args` is now much more reliable
- `Function` no longer ignores the return type annotation

[refactor]

- Migrated some dependency functionality from 0.12.5

[tests]

- Standard tests now check that randomly generated directories do not already exist before executing

[clean]

- Database write error is now more clear

28.1.9 0.11.20

[fixes]

- Removes breaking issues with the 0.12.0 engine by replacing it with the 0.11.19 engine

[refactor]

- Backport features from 0.12.4 onto the 0.11.19 engine

28.2 v0.12.x

> **Warning**
>
> All 0.12.x versions should be considered unstable. This version changes the engine in such a way that is incompatible with some machines. 0.11.20 is a feature backport of these versions. If you experience missing outputs using these versions, you should back up your data and update to 0.11.20.

28.2.1 0.12.4

[feature]

- Added the `force` keyword to `fetch_results()`, this will ignore any remote Dataset errors and continue with the fetch

28.2.2 0.12.3

[feature]

- extra_files_send/recv can now be passed via a {local:remote} dict pair. This allows for fine control of where files will be transferred

[fix]

- `TrackedFile` objects passed to extra files no longer have their dirs ignored
- `Dataset.run_path` now defaults to `remote_dir` if nonexistent
- `ensure_list` no longer drops the values of a passed `dict`, instead encasing it within a `list`

[refactor]

- `Runner` and `Dataset` now handle extra files via the `ExtraFilesMixin` class
- `TrackedFile` objects are now represented via `"{file}"` shorthand
- `rsync` no longer wastes a command on creating a local dir

[tests]

- Added testing sanitisation for file paths
- pytest-xdist testing now uses a standardised testing class

28.2.3 0.12.2

[fix]

- `Dataset.copy_runners` now creates deep copies of runners
- Fixed some instances of `Function` incorrectly serialising a function

[refactor]

- `DynamicMixin` now also validates on value access
- Copied Runner objects now update their name
- `RunnerState` is now less eager to claim that it is in a failed state
- `Database` now waits for temporary file until a timeout, rather than a flat delay
- `ensure_list` now uses a try/catch methodology
- `ensure_filetype` now returns the input string if target type is None
- `get_version` is now lazy

[docs]

- Fix missing clarification regarding use on Windows

[CI]

- Notebook setup now installs `[dev]` extras
- Use pytest-xdist in standard test suite

28.2.4 0.12.1

[refactor]

- `Dataset.run_args` is now a modifiable dictionary containing the globally set parameters and defaults
- `Runner.run_args` is now a modifiable dictionary containing the `Runner` level _overrides_ (Empty by default)
- The `run_args` used by a runner are now available at `Runner.derived_run_args`

[fixes]

- Fixed a crash involving `ensure_list` and integers

[docs]

- Improved the docs page on `run_args`

28.2.5 0.12.0

[refactor]

- The manifest file is now the absolute source of truth for Runners. It contains status updates, `stdout` and `stderr`. All remote run behaviour is now based on the contents of this file
- Error files no longer exist, reducing the amount of files present
- `Dataset.wait` will raise an exception if an error is encountered in a dependency situation. This behaviour differs slightly from a non dependent situation, where `wait` simply exits
- `Runner` job submission commands are now generated as a list by `Runner.generate_runline()`
- `check_runner_outputs` has been removed
- Internal timestamps are now in UTC format
- Dependencies use more `Dataset` and `Runner` code for staging
- `Function` now uses regex to extract the name and signature from string inputs

[fixes]

- The initial run_cmd is no longer always asynchronous
- `Runner` no longer accepts state changes from before the current run
- Fixed several instances of `Function` not properly inserting `*args` and `**kwargs`
- Fixed `Function` discrepancies between python object and string based initialisation

[features]

- **Added `Runner.stdout` and `Runner.stderr`**
  - You can now use the `print()` function within your jobs
  - Access to `manifest.write()` still remains
- `TrackedFile` now has a `local_time_utc` property, which calls `os.path.getmtime()` and converts to UTC

[tests]

- Moved several tests to the standard directory, to be tested outside of nbval

28.3 v0.11.x

28.3.1 0.11.19

[features]

- `wait` can now await a single runner, pass a `Runner` object to `wait` to await only that runner
- `BaseComputer` now raises an exception when attempting to set a value that already exists
- `Substitution` objects now return a clearer error when attempting to create a dynamic link

[fixes]

- Generalised more code relating to `Resource`/`Substitution`
- Temporary arguments now set their values properly

[refactor]

- `Function` now has more common code when storing callable or string inputs

[clean]

- The first submission function now has a docstring

28.3.2 0.11.18

[refactor]

- Jobs are now submitted via bash function, instead of a single line command
- `Dataset.reset_runs()` now defaults to `wipe=False`
- `URL.cmd` now returns a `CMD` object on `dry_run`

[fix]

- Fixed a crash where `RunnerState` would error on comparison to a non `RunnerState` object
- `RunnerState` <= and >= operators now function correctly

28.3.3 0.11.17

[fixes]

- Fixed an issue where using more than one = symbol in a template would cause a failure

[features]

- `DynamicValue` objects can now handle boolean operations (<, <=, =>, >, ==, !=)

[refactor]

- Added `try_value` function, which must now be used internally when comparing `DynamicMixin` and `DynamicValue` objects

28.3.4 0.11.16

[feature]

- Runners now use a `manifest` file for updating their status and checking run times
- Added `ds.update_runners()`, which updates the runner history with the manifest content

[fixes]

- Runner history is now sorted by time
- Runner history insertion now checks that the state does not already exist in the whole history, rather than just against the current state
- Added `finished` property to runner states
- Added `sub()` function to `TrackedFile`, allowing for inplace substitutions

[docs]

- Added a page detailing the `manifest.write` function, and `loaded`

[refactor]

- Repo and Manifest depend on an internal uuid, which can be overridden by dependencies
- Dependencies now use the `Dataset._write_to_repo` function, rather than implementing their own

[tests]

- Added a CI test that ensures that a `Dataset` will be able to skip after an update

0.11.15

[features]

- Added the `JUBEInterop` module. Currently only supports `JUBETemplate`
- Computers can now accept "semantic" time formats for arguments with `format=time`. For example: "24h" will be converted to "24:00:00"
- Template based Computers can now use the "wipe" or "local" methods when handling missing arguments
- `DynamicValue` will now attempt to be more clear when warning about an overwrite

[fixes]

- An error is now raised if templates have multiple substitutions with differing kwargs
- `BaseComputer.required` now checks all arguments, not just `Resource`
- Substitution now also checks availablity using regex
- Computer Arguments now accept the json-style "true" and "false"

[refactor]

- `BaseComputer` file downloading is now done within a staticmethod
- Added a special case for `entry_format` when format is "time"

28.3.5 0.11.14

[fixes]

- Fixed an issue where `Substitution` objects would not be applied if their output evaluated to `False`

28.3.6 0.11.13

[fixes]

- Fixed an issue where `Substitution` objects would fail to apply to more than just the first instance

28.3.7 0.11.12

[docs]

- Added initial tutorial page on running a `Dataset` without a function

[features]

- Added `Script`, available at `remotemanager.connection`. Subclass of `BaseComputer`, but provides a `run()` method.
- `Dataset` can now take `None` as a function. `Runner` arguments will then be passed directly to the `script()` method of the `url`, and the stdout captured as the result.

[fixes]

- Setting temporary values in `run()` no longer permanently updates arguments
- Fixed a bug that caused default values to always be treated as `str` in evaluations

[refactor]

- `Transport` is now a property of the `URL`, rather than `Dataset`
- `URL` now has a `uuid`, based off connection parameters
- `BaseComputer` now has a `uuid`, based off `URL.uuid` and arguments

28.3.8 0.11.11

[features]

- Templates can now accept python f-string style values, creating Dynamic links where possible
- Added `computers.concat_basic`, allowing concatenation of strings with Dynamics
- Added `target_kwargs` to `Substitution`, provides a dict of the originally specified kwargs

[fixes]

- Dynamics now cast to str when attempting to mix strings with non strings

[docs]

- Added docs for dynamic templates

[refactor]

- `default` property of Dynamic arguments now uses the correct formatting
- `Substitution` objects are now created using a new `from_string` classmethod
- Added `utils.Tokenizer`, which aids in digesting arbitrary code

[performance]

- Optimised the order that `DynamicValue` calculates operations

[tests]

- Added some non `nbval` testing

28.3.9 0.11.10

[fixes]

- Enables the `optional` keyword for substitutions

28.3.10 0.11.9

[features]

- Added `stream` option to `CMD`, which will attempt to stream stdout to the terminal
- Added `progress` option to `rsync`, which adds the `--progress` flag to transfers and streams the output

[refactor]

- Added output for running a dataset
- Verbose output for transferring files now a part of `Transport`
- `Verbose` properties are now granted through the `@make_verbose` class decorator
- Added `h` flag to default `rsync` flags

[fixes]

- Fixed instances of spurious `)` characters being added to logging calls

[CI]

- Update CI scripts for new ubuntu container

28.3.11 0.11.8

[features]

- **`Resource` and `Substitution` objects can now set (and store) their `value`**
  - Use `collect_values=False` when storing a `BaseComputer` to avoid this behaviour

[refactor]

- Updated how `verbose` works, now decoupled from logging
- Logs are no longer `yaml` format
- Removed `LoggingMixin`
- Removed `Quiet`
- Updated f-string logging calls with preferered lazy method
- `Logger` now warns only when force deleting a logfile with content

[fixes]

- `DynamicMixin` no longer checks for only `Resource` when linking
- `Logger` now only logs when required, no longer creating an empty file

[tests]

- Replaced nbval cell comments with jupyter cell tags

28.3.12 0.11.7

[fixes]

- Fixed a data loss issue related to `yaml.dump`

[clean]

- Update license dates
- Update pypi information

[docs]

- Add quickstart info to README

28.3.13 0.11.6

[features]

- Computers now save the current remotemanager version when storing to dict (disable with `include_version=False`)
- `BaseComputer` subclasses that fail to call the `super().__init__` now raise a more helpful error when generating a script

[docs]

- Minor clarity improvements
- Add some missing docstrings and typehints

[refactor]

- Common code of `Resource` and `Substitution` moved to their mixin class
- `RemoteFunction` is now located in the `decorators` module

[fixes]

- Defaults for computer arguments can now be quoted with " or '
- Fixed strange behaviour when dealing with strings addition in argument defaults
- Fixed an issue where `DynamicValue` would ignore the `value` property

[clean]

- Reduced argument objects no longer have redundant brackets surrounding a calculation

[tests]

- Testing suite does a better job of cleaning up after itself
- Fixed argument default tests that were testing the wrong thing
- Legacy computer unpack testing now actually uses the legacy method

28.3.14 0.11.5

[refactor]

- Remove unserialisation protections from `SendableMixin`. They were too easy to bypass, and causing bugs
- Runners now intialise and connect to the database at init

[fixes]

- `Resource` objects no longer drop their `tag` and `separator` attributes after a serial loop
- Fixed an instance where `CMD.latency` would over estimate the delay
- `landing_dir` no longer defaults to local $HOME

[docs]

- Updated installation info
- Added warning to intro regarding running with untrusted data
- Added the `move_docs.py` script, allowing for moving documentation pages while maintaining links
- Rearrange documentation layout
- Fix broken repository links
- Overhaul Computer documentation
- Minor clarity improvements

[clean]

- Added `docs` installation optional
- Update `dev` installation optional
- Several cleanup passes on various source files

[tests]

- add seveal tests to increase coverage

[CI]

- PyPi push is no longer allowed to fail

28.3.15 0.11.4

[features]

- `BaseComputer` can now accept a `template`, extracting and creating `Substitution` objects for any `#VALUE#` strings
- `Substitution` can now accept `format`
- Source code for a `parser` is now available at the `parser_source` property

[fixes]

- temporary runtime args no longer update only `Resource` objects
- fixed an issue where `format_time` would not bother to convert the input

[refactor]

- `format` is now stored in the `DynamicMixin`

28.3.16 0.11.3

[refactor]

- `Substitution.entrypoint` renamed to `name`

[features]

- Default args can now be quoted to prevent evaluation

28.3.17 0.11.2

[docs]

- Add a section to Dependency tutorial describing the nuances of environment variables

[feature]

- Added `Substitution` objects, a companion to `Resource` that replaces a target string with a variable
- Added `arguments`, `argument_objects` and `argument_dict` properties to `BaseComputer`
- Added `substitutions`, `substitution_objects` and `substitution_dict` properties to `BaseComputer`
- `Resource` objects can now specify their tag, defaults to `--`
- `Resource` objects can now specify their separator, defaults to `=`
- Added `DynamicValue.static`, which returns `True` if the value has no chain
- `format_iterable` can now also collect the object type with `print_type=True`

[refactor]

- `BaseComputer` `resources`, `resource_objects` and `resource_dict` return only `Resource` derived objects
- `DynamicValues` are now based on a `DynamicMixin` class
- Argument objects no longer rely on `SendableMixin`, using their `reduced` property
- `Resource` objects now use `__slots__`
- `URL.python` and `URL.shebang` are no longer property based attributes
- `Transport` no longer stores a copy of the `URL`, relying on `Dataset` to ensure it is properly populated
- Added `Dataset.prepare_for_transfer()`, which updates the `Transport` `URL`. Called by `avoid_runtime()`
- `extra` insertion is deferred entirely to the `URL`, allowing substitutions to work on temporary args
- Submission section can now be moved in the jobscript by adding `"#SUBMISSION_SUBSTITUTION#"`
- Toplevel imports are no longer relative

[fixes]

- Fixed an issue where `Resource` objects could be set improperly
- `BaseComputer` should now properly serialise connection parameters
- `run_args` in `run()` should no longer permanently update a `BaseComputer`
- `BaseComputer.extra` is now properly serialised
- Overwriting a `DynamicValue` that has a chain now properly prints a warning
- `URL.port` is no longer permanent

28.3.18 0.11.1

[fixes]

- Fixed an issue where a parser could be ignored by `to_dict`

[perf]

- Tweaked the method of Resource collection in `to_dict`

28.3.19 0.11.0

[docs]

- Add FAQ info on potential `rsync` workarounds
- Add FAQ info on updating a `Computer` to `0.11.x`
- Overhaul `Computer` tutorials

[refactor]

- **Overhaul `Computer` definitions**
  - `optional` and `required` objects are removed, and replaced with a `Resource` object. Use `optional=False/True` args to enforce either
  - Replaced `required_or` spec field with `requires` and `replaces` keywords of `Resource`
  - Replaced `optional_defaults` spec field with `default` keyword of `Resource`
  - `Resource` outputs are cast to `int` if possible (or not blocked by `format` keyword)
  - `parser` can now be defined directly on the class, in a pythonic way
  - Added a limited ability for custom `BaseComputer` derived classes to define user customised functionality
  - `BaseComputer` now defines a default parser which should work reasonably well, or serve as a basis for your own
- `Example_Slurm` and `Example_Torque` no longer have an underscore in their names (`ExampleSlurm` & `ExampleTorque`)
- `format_time` function is now available at toplevel `remotemanager.connection.computers`
- `Function` will no longer remove `self` if it is found. (Can be forced added with `force_self=True`)

[features]

- Added `DynamicValue`, an internal value store for `Resource` that allows for deferred calculation. This enables setting dependent variables. For example: `url.nodes = url.mpi * url.omp / url.cores_per_node`
- `parser` now has access to *self* within the function
- **`Resource` has a `format` keyword that accepts the "float" and "time" options**
  - "float" allows output to be formatted as a float
  - "time" will convert integer seconds to HH:MM:SS format
- `run_args` are available within the parser at `Resources["run_args"]`
- Added `resource_tag` to `BaseComputer` and `tag` to `Resource`. This allows setting of the pre `--flag` "tag"
- Added `resource_separator` to `BaseComputer` and `separator` to `Resource`. This allows setting of the inter `flag=value` "separator"

[CI]

- Increase repeatability of some tests

28.4 v0.10.X

28.4.1 0.10.19

[fixes]

- `CMD` now warns if the stderr is empty, but returncode is nonzero
- Fixed an issue where `CMD._file_communicate` would be too agressive when reading files

[refactor]

- `CMD` now internally tracks the returncode
- `Dataset.results` and `errors` now uses `avoid_runtime`, prevents some race conditions

[docs]

- Add basic LaTeX config

28.4.2 0.10.18

[features]

- `Dataset` tries to replace any missing runner files on `fetch_results()`
- Added `Dataset.retry_failed`, reruns only runners marked with `is_failed = True`
- Added `force_ignores_success` as an argument for `run()`. Needs to be `True` for `force` to run Runners that are marked as `is_success = True`
- Added `TrackedFile.exists_local` property

[refactor]

- Runners will now not run if `is_success` (use `force` AND `force_ignores_success` to run)
- Transfer CMDs are available at `Transfer.cmds`

[docs]

- Add FAQ section on partially failed datasets

[performance]

- Minor optimisations on Transfers

28.4.3 0.10.17

[refactor]

- Added a `shebang` argument to URL. This is also accessible at the `Dataset` level.

28.4.4 0.10.16

[features]

- Runners now check that the remote python version is at least 3.x.x

[fixes]

- `Function` now ignores any content before `def ...`, allowing for decorators and comments

28.4.5 0.10.15

[features]

- Added `BaseComputer.generate_cell()` which will produce a string which can be copied into a Jupyter cell, producing an editable reproduction of the current `Computer`
- Added `-q` flag to *URL.ssh* by default, this should reduce the number of issues caused when messages are broadcast on *stderr*
- `Dataset` `run_args` are now available in `BaseComputer.parser` functions at `resources["run_args"]`

[fixes]

- Fixed an issue where functions containing `@` characters would become mangled if stored as a `Function`

28.4.6 0.10.14

[features]

- `BaseComputer.to_yaml()` now prints the Parser in a more legible formats
- Added `base.time_to_s`, converting a `HH:MM:SS` string to integer seconds

[docs]

- Tweaked some docstrings and signatures

28.4.7 0.10.13

[fixes]

- `BaseComputer.from_repo()` now functions as expected

[refactor]

- Updating `Dataset.dbfile` is now more streamlined

[clean]

- clean instances of `" f"` autoformatting
- apply a pass of pycharm inspections

28.4.8 0.10.12

[docs]

- add FAQ

[feature]

- `BaseComputer` now dumps parser as a direct source string, rather than a `Function` object

[fixes]

- `SanzuFunction` can now handle functions with type hinting

28.4.9 0.10.11

[docs]

- Added missing complex serialisation tutorial

[fixes]

- Fixed a serialisation bug related to numpy (issue #8)
- Fixed an edge case where an object would pass json serialisation but not yaml

28.4.10 0.10.10

[features]

- `append_run` and `insert_runner` now have the option `return_runner`, which returns the appended runner
- `Function` now exposes a list of args at `args`
- `SanzuFunction` can now be called using non keyword args

[fixes]

- **Fixed a bug where `Function` was unable to regenerate a python object.**
  - This also fixes a related bug with `BaseComputer`

28.4.11 0.10.9

[features]

- Added `SanzuFunction`, allowing tagging of functions to run remotely

[refactor]

- `Jupyter` module renamed to `Decorators`

[docs]

- `Extra Functions` renamed internally to `Decorators`

28.4.12 0.10.8

[features]

- `Dataset.wait` now has the option `success_only`, which ignores failed runs
- `Dataset` now has a `default_url` property, allowing it to be set for all created Datasets

[fixes]

- `Dataset.all_success` now polls remote as expected
- `sanzu` function is now named `__sanzu_fn`, fixing an ambiguous name clash

[clean]

- remove `Numpy` dependency, only required for tests
- add functionality to clean documentation of pytest-nbval tags at `docs/source/clean_docs`

[docs]

- now cleaned of all pytest-nbval tags

[CI]

- Add retries to relevant tests

28.4.13 0.10.7

[docs]

- Update existing `sanzu` docs
- Add note on `spull` to `sanzu` docs
- Add note on exceptions to `sanzu` docs

[features]

- `sanzu` now returns the cell output
- `sanzu` will emit any warnings or errors it encounters

[refactor]

- sanzu now raises an exception if an empty cell is passed

28.4.14 0.10.6

[fixes]

- Changing the `URL` associated with a `Dataset` now also changes the corresponding `Transport` `URL`
- large run summary now correctly prints forced runs
- fixed an issue which prevented a `url` from being serialised after `test_connection` was issued

28.4.15 0.10.5

[features]

- `Runners` now have the `set_run_arg` suite of functions
- `Dataset` now has an `all_success` property, returning True if all runners have completed successfully

[fixes]

- `RunnerStates` no longer report as `failed` for incomplete runs
- `Function` no longer mangles signatures with renamed `*args` and `**kwargs`

[docs]

- Add a section on `run_args`

28.4.16 0.10.4

[features]

- `ds.backup` now has a `full` option, which will also collect `extra_files_send`

[fixes]

- `url.passfile` and `keyfile` are now settable properties
- `Dataset.restore` no longer overwrites an existing `Dataset` by default

28.4.17 0.10.3

[docs]

- Tweaked intro page on requirements

[refactor]

- Added a workaround for `ast.unparse` that allows the local machine to run python 3.7

28.4.18 0.10.2

[docs]

- Added Version History page
- Update index page
- Update note on version limtations

[features]

- Added `sshpass_override`, allowing overriding of the `sshpass` string
- `URL` now splits a `"user@host"` string

[fixes]

- Remove usages of | syntax from signatures
- Bump minimum python version to 3.9

[CI]

- Minimum python version (3.9) is now tested within the CI

28.4.19 0.10.1

[docs]

- Minor improvements to Dataset tutorial

[features]

- Added `url.ssh_insert`, allowing a custom string to be placed just after any ssh call

[refactor]

- `BaseComputer.argument_dict` is now much more clear
- `host` is now the first kwarg of `URL`

28.4.20 0.10.0

[fixes]

- Fixed an instance where a runner which raises a warning but ultimately completes would never be marked as such

[refactor]

- Runners now inherit their base `run_args` from the parent
- `Dataset.run_args` must now be set by the `set_run_arg` family of functions

[fixes]

- Runner `run_args` are now always valid for the current state

28.5 v0.9.X

28.5.1 0.9.24

[docs]

- Update introduction package overview image
- Added a section to intro on requirements

[fixes]

- Fixed an issue where a non-explicit remote directory would break result recovery

[features]

- Added `url.expandvars(string)` which will pass the string through the remote, expanding any variables

[refactor]

- Remote, local and run dirs are now sanitised to PurePosixPath

28.5.2 0.9.23

[features]

- Added boolean `results`, `errors` and `extras` args to `fetch_results()`
- Added `ds.fetch_errors()` which only fetches error files

[fixes]

- `wait()` now properly awaits single runs
- `lazy_append` context no longer errors if nothing is appended
- Fixed an issue where a directory sent with extra files could not be wiped
- Fixed an issue where sending a file named "*" could cause dangerous interactions with file wipes

[refactor]

- `Dataset(skip=...)` is no longer solely dependent on database presence
- Runners that have not been run now return `None` for their `is_finished`

28.5.3 0.9.21

[docs]

- API documentation is now much cleaner, modules are only documented within their module page
- Added dedicated section on complex serialisation
- Added a note on `run_cmd` to failure tutorial
- Made Quickstart quicker, and more relevant

[fixes]

- Fixed some instances where `Function` would improperly parse the signature

[clean]

- Removed some more instances of `__all__` reliant imports

28.5.4 0.9.22

[refactor]

- Runners that fail now have a `RunnerFailedError` as their result

[fixes]

- Newly created runners no longer report as `is_failed`

28.5.5 0.9.20

[feature]

- Added `ds.insert_runner(runner)`, allowing insertion of an unmodifed runner to a Dataset
- Added `ds.copy_runners(dataset)`, copying runners from `dataset` into `ds`

[fixes]

- Fixed an instance where a `Transport` could hold a its own `URL`, separate to `Dataset`

[refactor]

- Runners will no longer run if their error file already exists (usually populated by a parent)

- Child jobs that have a pre-populated error file will pass it along the chain
- Runners now clear their `result` and `error` properties at run (files remain untouched)

[tests]

- Increased reproducibility of some unstable tests

[docs]

- Overhaul tutorial on handling failures

28.5.6 0.9.19

[features]

- Added the `dir_mode` option to `Transport`. On a multi-file transfer, files are copied into a temporary dir and copied using "*", instead of using bash brace expansion
- Added `URL.landing_dir`, allowing the user to set the "landing" directory that commands are executed in by default. Functions by prepending a `cd {landing_dir} &&` to each cmd

[fixes]

- Fixed an issue where a `Dataset` could repopulate its database incorrectly

[docs]

- Add docs sections for new features

28.5.7 0.9.18

[fixes]

- `scp` no longer tries to create the local target dir on the remote machine on pull

[refactor]

- `scp` now formats commands similarly to `rsync`

28.5.8 0.9.17

[fixes]

- Error files are no longer ignored if there is a result present
- Single runner runs no longer wipe all error files

[docs]

- Fix a dependency issue preventing docs compilation

28.5.9 0.9.16

[docs]

- Add documentation for the new Backup and Restore systems
- Trim Dataset tutorial

[features]

- Added `Dataset.backup` and `Dataset.restore`, allowing backing up, of a dataset and its results to a zip file for later recovery

- Added `Database.backup`, which will backup the current database file to `file`. Appends `.bk` to the current name by default

[fixes]

- Dataset not longer contaminates the `extra_files_recv` with send when doing a remote searches
- Fixed an issue where objects within lists would not be deserialised

28.5.10 0.9.15

[fixes]

- Runners that have not been run can no longer be considered for completion
- Dataset extra can now be set after initialisation

[refactor]

- BaseComputer is now importable from `remotemanager`
- SendableMixin now uses the same method for objects that use `__slots__` and `__dict__`

[tests]

- Fix wrongly named test

28.5.11 0.9.14

[features]

- Added `runner.full_error`, returning the full content of the error file as a string

[fixes]

- Using a function called dump or load no longer silently fails
- `ds.results` now checks if there are errors and sends a warning if there are
- `TrackedFile.content` now returns `None` if the file does not exist

28.5.12 0.9.13

[fixes]

- Appending a runner that already exists now warns using the existing runner name
- Database no longer sorts runners alphabetically
- `Database.find()` now returns an empty dict if the value was not found
- Fixed an issue where an old Dataset could be updated into a new one
- Fixed an issue where `sanzu` would raise an error after a skipped run
- Update sed command to perform identically on *nix and BSD based OSes (@wddawson)

28.5.13 0.9.12

[features]

- `avoid_nodes` is now a `run_arg`, meaning you can use it as part of a dependency

[refactor]

- Improve `TrackedFile.write`, now attempts to convert args to string type
- `TrackedFile.write` now has a add_newline arg (defaults `True`)
- Moved the base `Transport._cmd` method into the `cmd` method

[clean]

- Replace instances of explicit `TrackedFile` writing with `TrackedFile.write`

28.5.14 0.9.11

[features]

- Lazy appends now only print a summary of the process after completion, rather than for each runner
- If the number of runners is more than `Dataset.run_summary_limit` (defaults to 25, settable), `run()` will print a summary instead of a per-runner update

28.5.15 0.9.10

[refactor]

- Runner now wipes and reinitialises any TrackedFile instances at run, making them "static"
- TrackedFile no longer has Logging access, allows full conversion to `__slots__`
- extra files are now handled via `TrackedFile`
- `is_finished` is now checked against the Runner files' `last_seen_remote`

28.5.16 0.9.9

[fixes]

- `SendableMixin` now correctly handles objects that use `__slots__`

[refactor]

- `check_runner_outputs` now also checks for extra_files_recv
- Replaced an instance of old run path generation with `runner.run_path`

28.5.17 0.9.8

[refactor]

- Removes wildcard import from toplevel `__init__`

[fixes]

- `Database.find()` now searches more thoroughly. Use `greedy=True` for old behaviour

[clean]

- Update pre-commit hooks

28.5.18 0.9.7

[fixes]

- Master dir location passing is now done via a sed update of the jobscripts
- `wipe_local` no longer deletes local files from extra_files_send

28.5.19 0.9.6

[refactor]

- `wipe_local` now uses the same file collection methodology as wipe_remote
- `hard_reset`, `wipe_local` and `wipe_remote` now default to `file_only=True`

28.5.20 0.9.5

[features]

- Added `runner.is_failed` and `runner.is_success`, returning `True/False` if the runner has completed successfully or not
- Added `runner.extra`, which allows extra lines to be inserted into the jobscript of a runner. Use just like a run_arg, lines are inserted prior to submission.

[docs]

- Add a section regarding handling of failed runners

[CI]

- Tests are now split into sections, forcing some level of parallelisation

28.5.21 0.9.4

[fixes]

- Dataset master dir is now given to the runners via command line instead of environment vars

[features]

- Added `success` property of `RunnerState`, allows for extra context
- Added `ds.failed`, a list of runners reporting to have failed

28.5.22 0.9.3

[features]

- Added `runner.set_state()`, which allows force setting of the state (see fixes)

[fixes]

- Fixed an instance where an older run could masquerade as the results of a more recent one
- Runner state update now blocks the change if it would duplicate the state

28.5.23 0.9.2

[features]

- Adds `ds.set_run_args`, allowing a quick way of setting a run arg for each runner in the dataset

[fixes]

- Satisfied runners no longer re-update their status to "satisfied"

28.5.24 0.9.1

[fixes]

- Lazy append is now O(N) when appending to a dataset that has a linked dependency

28.5.25 0.9.0

[refactor]

- Drastically changes how runners are handled within a run
- TrackedFiles are now much more prevalent, and have increased power
- Dependencies can now be run from anywhere in the chain

[features]

- Runner states are now handled in a RunnerState class, allowing for more precise state checking and tracking
- TrackedFile allows for obtaining a `relative_remote_dir(source)`, which gives a callpath from `source` to the file
- TrackedFile now has a `write` method
- Transport now has direct support for a TrackedFile
- Added `utils.dir_delta`, which gives the relative directory level difference between two directories

Important

BaseComputer should be considered deprecated, and will be removed as soon as it is possible to do so. With recent changes to templates, there should be no reason to use `BaseComputer` over `Computer`. However, if you find such a situation, please open an issue.

CHAPTER

TWENTYNINE

CREATING A "TRUE" COMPUTER

Important

If your computer exists from before `0.11.x`, it is likely that your computers need updating. See the dedicated section for info about this.

29.1 Unscripted

Previous tutorials covered generating scripts from already existing jobscripts.

In this tutorial we will be covering how to generate scripts completely from scratch.

An understanding of your target scheduler will help here, however you may be able to get something functional by using the examples and tweaking them until you don't get errors anymore. You can import from `from remotemanager.connection.computers import Resource`, where you will find `ExampleSlurm` and `ExampleTorque`. These should get you most of the way there, and using the advice later in this tutorial you can use these as a starting point.

29.2 Machine Agnosticism

The end goal of a `Computer` object is to provide a machine agnostic interface between what you put in your notebook/dataset, and what the machine on the other side is expecting. Most often, HPC systems require a request for resources, which is then granted by a scheduler.

Since the syntax of these requests and the scheduler commands themselves can differ wildly between machines, we need a "middleman" to ensure that workflows are uniform across machines.

That is to say, if you have a dataset specifed with `mpi=64, omp=4, nodes=1`, for example, this will *always* request those resources regardless of the machine it is run on.

Hence "machine agnostic". Ideally a `Dataset` that runs on one machine, should be able to run on a completely different one by swapping out the connection.

29.2.1 Resources

We'll start by creating a Computer that requests some common resources. This is done using a `Resource` object, which represents one attribute of a machine.

This should make more sense as we progress.

[2]:
```
from remotemanager import BaseComputer
```

Just as before, we start from `BaseComputer` as does a lot of the internal work for you.

With templates we were only accessing a fraction of what it can do. By *subclassing* this object we can access the full suite of features.

To request resources we'll need our `Resource` object, so lets import and explore that first.

[3]:
```
from remotemanager.connection.computers import Resource
```

The Resource class

`Resource` is what does all of the conversion work, it specifies how to convert arguments into jobscript strings. As a simple example, lets create a basic `Resource` for `nodes`.

[4]:
```
nodes = Resource(name="nodes", flag="nodes")
```

Sure looks like a lot of repetition there, right? It does, but these inputs are all required. Lets run though them

- `nodes = ...` Is the initial assignment. It lets us access this resource again. In a computer, you'll be assigning this to `self.nodes`, but we'll get there later.
- `name = "nodes"`. We need to let the `Resource` know its own name, this is important for some of the helper methods available in `BaseComputer`. A simple rule of thumb is to just set this to whatever you assign the object to. So `mpi = Resource(name = "mpi", ...)`, etc.

Important

When specifying resources this way, `name` is required.

- `flag = "nodes"`. This is the actual *translation*. The `flag` argument dictates what's put in the jobscript. So for a slurm scheduler, it's expecting `#SBATCH --nodes=n`. In the `mpi` case above, it would be expecting something similar to `flag = ntasks` for `#SBATCH --ntasks=t`.

With that explained, lets look at some of the other args of `Resource`.

29.2.2 Optional

By default, `Resource` is marked as `optional`. This means that if it is not specified, the `Computer` will just ignore it when making the script. If you *must* have this resource specified, you can flag it as `optional=False` when creating it.

This will make the `Computer` raise an error when generating the script, and will also allow it to be queried within the `required` and `missing` properties. But more on that later.

[5]:
```
print(f"Is nodes optional? {nodes.optional}")

nodes = Resource(name="nodes", flag="nodes", optional=False)
print(f"Is nodes optional? {nodes.optional}")
```

```
Is nodes optional? True
Is nodes optional? False
```

29.2.3 Defaults

Resources can also specify a default, which will be presented in the case that nothing has been set.

[6]:
```
nodes = Resource(name="nodes", flag="nodes")

print(nodes.value)
```

```
None
```

[7]:
```
nodes = Resource(name="nodes", flag="nodes", default=1)

print(nodes.value)
```

```
1
```

29.2.4 `min` and `max`

Numerical resources can have a specified `min` and `max`. When setting the `value`, a validation step will be done, raising an error if the limits are exceeded.

[8]:
```
nodes = Resource(name = "nodes", flag = "nodes", max=64)

nodes.value = 16
```

[9]:
```
nodes.value = 128
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[9], line 1
----> 1 nodes.value = 128

File ~/Work/Devel/remotemanager/remotemanager/connection/computers/dynamicvalue.py:
↪348, in DynamicMixin.value(self, value)
    346 @value.setter
    347 def value(self, value):
--> 348     self.set_value(value)

File ~/Work/Devel/remotemanager/remotemanager/connection/computers/dynamicvalue.py:
↪379, in DynamicMixin.set_value(self, value)
    362 def set_value(self, value):
    363     """
    364     Sets the value, separating out the function allows for property␣
↪overloading
    365
   (...)
    377         to be a combination of other resources (DV)
    378     """
--> 379     self._validate(value)
    381     if isinstance(self._value, DynamicValue):
    382         # we're setting on an Argument _value
    383         if isinstance(value, DynamicValue):
    384             # if _value has any extra properties,
    385             # need to be careful not to drop them

File ~/Work/Devel/remotemanager/remotemanager/connection/computers/dynamicvalue.py:
↪423, in DynamicMixin._validate(self, value)
```

(continues on next page)

(continued from previous page)

```
    419     raise ValueError(
    420         f"{value}{nameinsert} is less than minimum value {self.min}"
    421     )
    422 if self.max is not None and value > self.max:
--> 423     raise ValueError(
    424         f"{value}{nameinsert} is more than maximum value {self.max}"
    425     )

ValueError: 128 for nodes is more than maximum value 64
```

29.2.5 Formatting

The `format` keyword allows some formatting to be done before outputting. The available formats are:

- `float`: By default, any value that evaluates to a float will be cast to an `int` type (to prevent issues like nodes=16.0). Set `format="float"` to avoid this.
- `time`: This format allows automatic conversion of `time={nSeconds}` to a `hh:mm:ss` format.

[10]:
```
int_resource = Resource(name="int", flag="int")
flt_resource = Resource(name="flt", flag="flt", format="float")

int_resource.value = 20/5  # This will be conveted to an int
flt_resource.value = 20/5  # Specifying format="float" will return this as-is
```

[11]:
```
print(int_resource.value)
```

```
4
```

[12]:
```
print(flt_resource.value)
```

```
4.0
```

[13]:
```
time_resource = Resource(name="time", flag="walltime", format="time")

time_resource.value = 86400  # 24h in seconds
```

[14]:
```
print(time_resource.value)
```

```
24:00:00
```

29.3 Computer Creation

Now we have an understanding of what the `Resource` class does, we can put it into a computer.

Lets create a very simple `slurm` scheduler interface. Start by creating your class, and subclassing `BaseComputer`. You should always make the `*args` and `**kwargs` available, and pass them to `super().__init__(*args, **kwargs)`.

This is what grants your `Computer` all of the functionality of both `BaseComputer` and `URL`.

[15]:
```
class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

Important

All computers should start this way, missing the `super()` call will cause strange behaviour.

This class will "function" as a URL at the very least. You can give it a user and hostname and it will behave as expected. But this isn't very useful for us, since we need to access the scheduler. Lets tell it how to do that:

[16]:
```
class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.submitter = "sbatch"
        self.shebang = "#!/bin/bash"
        self.pragma = "#SBATCH"
```

29.3.1 submitter

Previously, we had to specify the `submitter` when generating our connection.

However when generating a class version, we can specify it internally, eliminating the need to set it each time you access this machine.

We can additionally set the shebang, if that needs to be specifies. It defaults to `#!/bin/bash`, though we have set it here for clarity.

Pragma is a special string that defines the beginning of a resource line. `#SBATCH` in slurm, for example.

29.3.2 Defining Resource Requests

Now the base `URL` knows how to submit a job, what the shebang should be, and the prefix to the resource requests.

Lets add some resources.

[17]:
```
class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.submitter = "sbatch"
        self.shebang = "#!/bin/bash"
        self.pragma = "#SBATCH"

        self.mpi = Resource(name="mpi", flag="ntasks", min=1)
        self.omp = Resource(name="omp", flag="cpus-per-task", min=1, max=64)
        self.nodes = Resource(name="nodes", flag="nodes", optional=False)
        self.time = Resource(name="time", flag="walltime", optional=False, format=
↪"time", default=3600)
        # Note this addition, it will be optional, and we won't specify it.
        # This will stop it from appearing in the output
        self.test = Resource(name="test", flag="test")
```

Note

Remember that these objects are arbitrary, and unlimited. A good practice is to go through some jobscripts and create a `Resource` object for what you see. Noting that `flag` is whatever you see in the jobscript, and `name` can be whatever you want.

Lets see what sort of script this generates:

[18]:
```
test = Computer()

print(test.script())
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[18], line 3
      1 test = Computer()
----> 3 print(test.script())

File ~/Work/Devel/remotemanager/remotemanager/connection/computers/base.py:893, in␣
↪BaseComputer.script(self, **kwargs)
    890         self.argument_dict[key].temporary_value = val
    892 if not self.valid:
--> 893     raise RuntimeError(f"missing required arguments: {self.missing}")
    895 if self.template is None:
    896     logger.debug("Creating script from Resources")

RuntimeError: missing required arguments: ['nodes']
```

Great, an error.

If we look at the definition again we can see what happened.

We defined nodes as non-optional, but then didn't specify it. So when the Computer went to generate the script it instead raised an error.

We can see from the error message a list of missing arguments to fill in. If we include those (and a few others), we'll see a better output.

Note

We also specified `time` as non optional, but then also provided a default. Since it can fall back to its default value, it was not mentioned in the error output.

[19]:
```
test.mpi = 4
test.omp = 4
test.nodes = 1

print(test.script())
```

```
#!/bin/bash
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=4
#SBATCH --nodes=1
#SBATCH --walltime=01:00:00
```

Et voila! We specifed some resources, and they were converted into resource request lines as specified in the `Computer` and `Resource`.

When using a `Computer` in a workflow, you are not limited to setting the values at the `Computer` level.

Important

A `Resource` named "test" was added for this example, but is missing from the output. Just like with Templates, resources without arguments will not be added. This means that it's preferable to *over* provision with `Resource` objects, rather than exactly meet your targets.

Note

We covered the specifics of this in the template tutorial.

[20]:
```
from remotemanager import Dataset

def f():
    return

url = Computer()

ds = Dataset(f, url=url, skip=False)

ds.set_run_arg("omp", 4)
ds.set_run_arg("mpi", 64)
ds.set_run_arg("nodes", 8)

print(ds.script)
```

```
#!/bin/bash
#SBATCH --ntasks=64
#SBATCH --cpus-per-task=4
#SBATCH --nodes=8
#SBATCH --walltime=01:00:00

#SUBMISSION_SUBSTITUTION#
```

We can also query what is available (and required):

[21]:
```
test.arguments
```

[21]:
```
['mpi', 'nodes', 'omp', 'test', 'time']
```

[22]:
```
test.required
```

[22]:
```
['nodes']
```

[23]:
```
test.missing
```

[23]:
```
[]
```

Note

The `missing` parameter here is exactly what is output in the error message we saw above.

29.3.3 valid

Instead of checking `missing`, or waiting for an error, you can also check the `valid` property of a `Computer`.

This returns `True` if the `Computer` thinks that it will produce a working jobscript, and `False` if not.

> **Note**
>
> This is the same condition checked when raising the `RuntimeError` for missing values.

[24]:
```
test = Computer()
test.valid
```

[24]:
```
False
```

[25]:
```
test.missing
```

[25]:
```
['nodes']
```

[26]:
```
test.nodes = 4
test.valid
```

[26]:
```
True
```

29.4 Dynamic Values

Added in version 0.11.0.

Values can also depend on other values. This allows you to calculate based on other inputs. An obvious use case for this is calculating the nodes request.

Lets assume we have a machine that has 128 cores per node, we can specify the value this way:

[27]:
```
test = Computer()

test.mpi = 128
test.omp = 4

test.nodes = test.mpi * test.omp / 128

test.time = 3600
```

Since each node can handle 128 possible tasks, we need to calculate the total number of tasks (ntasks * omp), then divide by 128 to get the number of nodes needed to handle this amount of tasks.

[28]:
```
print(test.script())
```

```
#!/bin/bash
#SBATCH --ntasks=128
#SBATCH --cpus-per-task=4
#SBATCH --nodes=4
#SBATCH --walltime=01:00:00
```

Here the value comes out to 4, as we'd expect. Since we did not specify `format="float"`, the value is rounded up and cast to Int type.

Rounded up for the case where we request a fractional (by nodes) amount of tasks. We should still request enough nodes to cover the request.

Lets change mpi to 112. `Since 112 * 4 / 128 = 3.5` and we can't have 3.5 nodes, we need to request 4 nodes:

[29]:
```
test.mpi = 112

print(test.script())
```

```
#!/bin/bash
#SBATCH --ntasks=112
#SBATCH --cpus-per-task=4
#SBATCH --nodes=4
#SBATCH --walltime=01:00:00
```

Note

What happens to this mismatched number of cores and nodes is up to the scheduler.

29.5 DynamicValues as Defaults

Added in version 0.11.2.

Setting values to be dynamic can be very useful for automated parameterisation, but you can go one step further and set them as defaults.

Warning

This is a complex feature, and it's likely that there are still some edge cases yet to be found. If your Computer definition exhibits strange behaviour please file a bug report. Even if it's *not* a bug, it's likely an issue with the documentation.

Lets generate a simple computer to exhibit the automatic nodes behaviour by default:

[30]:
```
class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.cores_per_node = 128

        self.submitter = "sbatch"
        self.shebang = "#!/bin/bash"
        self.pragma = "#SBATCH"

        self.mpi = Resource(name="mpi", flag="ntasks")
        self.omp = Resource(name="omp", flag="cpus-per-task")
        self.nodes = Resource(name="nodes", flag="nodes", default=self.mpi * self.omp␣
↪/ self.cores_per_node)
```

Now when we ask for a script without specifying nodes, it is calculated for us.

[31]:
```
test = Computer()

test.mpi = 128
test.omp = 4
```

(continues on next page)

(continued from previous page)

```
print(test.script())
```

```
#!/bin/bash
#SBATCH --ntasks=128
#SBATCH --cpus-per-task=4
#SBATCH --nodes=4
```

And just to hammer the point home, if we double the mpi request, we'll see that the request for nodes also increases automatically.

[32]:
```
test.mpi = 256
test.omp = 4

print(test.script())
```

```
#!/bin/bash
#SBATCH --ntasks=256
#SBATCH --cpus-per-task=4
#SBATCH --nodes=8
```

29.6 Adding to Python objects

While it is safe to add python intrinsics to a variable, the reverse is not true.

These resource objects can be "chained" into each other as they provide special function to handle the operators.

When adding to standard object, these operator modifications are not present, so you'll get an error.

29.6.1 Solutions

There are two solutions to this issue. The first is to create a Resource for the variable.

This is the "heavy" approach, and allows this variable to be tweaked and changed.

However this can rapidly spiral out of control, and some variables simply don't need to have a hook.

For this, there is `concat_basic`. This performs the addition for you, ensuring that the returned object is properly linked.

[33]:
```
from remotemanager.connection.computers import concat_basic

value = Resource(name="value", default=10)

string_prefix = Resource(name="strprefix", default=concat_basic("test_", value))

print(string_prefix.value)
```

```
test_10
```

29.7 This Script Format Doesn't Fit my Scheduler

The scripts that have been generated thus far in the tutorials have been of one single format.

Some schedulers have a more specific requirements, so need to be treated differently.

The tutorial will cover the details of *how* these scripts are generated, which will allow you to personalise them much more deeply.

CHAPTER

THIRTY

UPDATING A COMPUTER

30.1 Fixing a Broken Computer

If you already have Computers from before version `0.11.x`, it is very likely that they are "broken". If they are in .yaml format, the steps to fix them are as follows:

30.1.1 YAML Fixes

1. Import as normal (`from_yaml(...)`). You should see a warning saying that an old style import is detected.
2. If the import succeeds, it means that translation should have worked. Immediately dump this computer to a *different file* (don't overwrite the old one yet, in case something goes wrong)
3. Call `generate_cell()` on this new object. And copy the output to a new cell.
4. Your parser may need editing (see section below).
5. This will create a new computer named `new`. Save this somewhere safe, it is your new Computer.

30.1.2 Class Fixes

If your `Computer` is a `class`, you can make your edits on the original definition.

Change your `optional` and `required` objects to `Resource`.

[1]:
```
from remotemanager.connection.computers.resource import Resource

# optional(...)
Resource(...)

# required(...)
Resource(..., optional=False)
```

[1]:
```
None
```

`parser` no longer has to be assigned to `_parser`, and can be defined right there in the class.

You can take your current function, and define it like you would any other function:

[2]:
```
from remotemanager import BaseComputer

class computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(args, kwargs)

        self.var = Resource(...)
```

(continues on next page)

(continued from previous page)

```
    def parser(self, resources):
        ...
```

You should also make the necessary changes as described below.

30.1.3 Parser fixes

Two things have changed with the parsers:

1. `format_time` has moved to `remotemanager.computers.connection.utils`
2. `resources` is no longer a dict, so needs to be handled differently

For `format_time`, you can delete all references to it and add `"format": "time"` to your `time` Resource. (See formatting info)

30.1.4 `Resources.items()`

Now the `resources` input is a fully fledged `Resources` object, you can change

`for k, v in resources.items():` to `for v in resources:`

And then any reference to `k` to `v.name`.

Added in version 0.11.6.

This is not a requirement, however, since `Resources` now provides an `items()` function that behaves identically to the dict form.

Here is a minimal example of a parser:

[3]:
```
def parser(resources):
    output = []
    for v in resources:
        if v:
            output.append(f"{resources.pragma} --{resource.flag}={resource.value}")
    return output
```

CHAPTER

THIRTYONE

DEEPER CUSTOMISATION WITH PARSER

31.1 Hidden Function

The parser is the function that does the actual jobscript generation. It takes your resources, and spits out a list of lines that should go at the top of your script.

Up until now, we have been secretly relying on the internal "default" parser. This works by iterating over the stored `Resource` objects, and then dumps them into a list according to the schema `"{pragma}{tag}{flag}{separator}{value}"`.

..note:: `tag` and `separator` are configurable keyword args of `Resource`, defaulting to "–" and "=", respectively. You can also set them at the `Computer` level, as demonstrated below.

But what if this doesn't work for your machine? It certainly doesn't work for PBS based machines.

Well that's where we need to specify our own.

This tutorial will explain the general concepts of defining a parser with the end goal of generating a PBS-friendly jobscript, but remember that these topics are general.

[1]:
```
from remotemanager.connection.computers import BaseComputer, Resource

class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.submitter = "sbatch"
        self.shebang = "#!/bin/bash"
        self.pragma = "#SBATCH"

        self.mpi = Resource(name="mpi", flag="ntasks", min=1)
        self.omp = Resource(name="omp", flag="cpus-per-task", min=1, max=64)
        self.nodes = Resource(name="nodes", flag="nodes", optional=False)
        self.time = Resource(name="time", flag="walltime", optional=False, format=
↪"time", default=3600)
```

[2]:
```
test = Computer()

test.nodes = 1

print(test.script())
```

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --walltime=01:00:00
```

In PBS based parsers, the actual resource line is expected to be something that follows the form:

`#PBS -l nodes=1:ppn=4,walltime=01:00:00`

Additionally, `OMP` is usually specified by an environment variable.

With our default behaviour of putting one resource per line, this is obviously not going to work.

So we need to specify our own `parser`. Lets begin by explicitly defining the *current* behaviour:

[3]:
```
class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.submitter = "sbatch"
        self.shebang = "#!/bin/bash"
        self.pragma = "#SBATCH"

        self.mpi = Resource(name="mpi_per_node", flag="ppn", default=4)
        self.omp = Resource(name="omp", flag="cpus-per-task", default=4)
        self.nodes = Resource(name="nodes", flag="nodes", default=1)
        self.time = Resource(name="time", flag="walltime", format="time",␣
↪optional=False, default=3600)

    def parser(self, resources: "Resources") -> list:
        output = []
        for resource in resources:
            if resource:
                output.append(resource.resource_line)

        return output
```

[4]:
```
test = Computer()
print(test.script())
```

```
#!/bin/bash
#SBATCH --ppn=4
#SBATCH --cpus-per-task=4
#SBATCH --nodes=1
#SBATCH --walltime=01:00:00
```

31.1.1 Parser "gotchas"

Theres a few things to note here.

resources

The `resources` keyword accepts a `Resources` object. As you'd imagine, this is a special carrier object that holds our `Resource` objects.

> **Note**
>
> Despite the similar name, `Resources` and `Resource` are not the same object. The former is a collection of the latter.

It functions like a list, with some extra functions. The important thing to note is that you can iterate over it, just like a list.

`for resource in resource:` will give you each resource, one by one.

> **Note**
>
> You can also access a `Resource` by its name, like you would a dictionary. See the section below for info.

`if resource`

`bool(resource)` will evaluate to `True` if the resource has both `value` *and* `flag`. This `if resource` line prevents resources without a value being added to the jobscript.

Omitting this usually results in a line like `#SBATCH --ntasks=None`.

[5]:
```
a = Resource(name="a", flag="resource")
print("A resource without a value will evaluate to False:", bool(a))

a.value = "test"
print("When given a value, it will then evaluate to True:", bool(a))
```

```
A resource without a value will evaluate to False: False
When given a value, it will then evaluate to True: True
```

[6]:
```
b = Resource(name="a")
print("A resource without a flag will always evaluate to False:", bool(b))

b.value = "test"
print("When given a value, it will still evaluate to False:", bool(b))
```

```
A resource without a flag will always evaluate to False: False
When given a value, it will still evaluate to False: False
```

return type

The `parser` should always return a `list` of lines, *not* a string.

31.1.2 Making our PBS parser

Okay now we've got that down, we can start thinking about how to edit our parser to work as we want.

Since we want to combine the resources `mpi_per_node`, `nodes` and `time` into a single line, we should exclude those in the main loop.

Lets also change the format to the right one too.

[7]:
```
class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # change our submitter and pragma to the right values
        self.submitter = "qsub"
        self.shebang = "#!/bin/bash"
        self.pragma = "#PBS"
        # PBS also uses a different format for the lines, lets set that too
```

(continues on next page)

(continued from previous page)

```
        self.resource_tag = "-"
        self.resource_separator = " "
        # same resources, we're changing the FORMAT, not the CONTENT
        self.mpi = Resource(name="mpi_per_node", flag="ppn", default=4)
        self.omp = Resource(name="omp", flag="cpus-per-task", default=4)
        self.nodes = Resource(name="nodes", flag="nodes", default=1)
        self.time = Resource(name="time", flag="walltime", format="time",␣
↪optional=False, default=3600)

    def parser(self, resources: "Resources") -> list:
        output = []
        for resource in resources:
            # exclude by name, we need to treat these separately
            if resource and resource.name not in ["mpi_per_node", "nodes", "time",
↪"omp"]:
                output.append(resource.resource_line)

        return output

test = Computer()
print(test.script())
```

```
#!/bin/bash
```

That looks good, now there's no breaking lines in the output.

Important

It looks like the loop is doing "nothing", since we're excluding everything that we've added. However it's still good practice to add it anyway, since it will catch any extra resources you choose to add in the future.

Now we've cleaned the output, we can form the line we want and add it how we like.

Lets additionally grab the `omp` resource and export its value.

Oh, and add the `cd $PBS_O_WORKDIR` line for good measure.

[8]:
```
class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.submitter = "qsub"
        self.shebang = "#!/bin/bash"
        self.pragma = "#PBS"

        self.resource_tag = "-"
        self.resource_separator = " "

        self.mpi = Resource(name="mpi_per_node", flag="ppn", default=4)
        self.omp = Resource(name="omp", flag="cpus-per-task", default=4)
        self.nodes = Resource(name="nodes", flag="nodes", default=1)
        self.time = Resource(name="time", flag="walltime", format="time",␣
↪optional=False, default=3600)

    def parser(self, resources: "Resources") -> list:
```

(continues on next page)

(continued from previous page)

```
        output = []
        for resource in resources:
            if resource and resource.name not in ["mpi_per_node", "nodes", "time"]:
                output.append(resource.resource_line)
        # extract the values and format them before adding
        ppn = resources["mpi_per_node"]
        nodes = resources["nodes"]
        wtime = resources["time"]
        output.append(
            f"{resources.pragma} -l nodes={nodes}:"
            f"ppn={ppn},"
            f"walltime={wtime}"
        )
        # We can add extra important lines in here, too
        output.append("\ncd $PBS_O_WORKDIR")
        output.append(f"export OMP_NUM_THREADS={resources['omp']}")

        return output

test = Computer()
print(test.script())
```

```
#!/bin/bash
#PBS -cpus-per-task 4
#PBS -l nodes=1:ppn=4,walltime=01:00:00

cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=4
```

> **Tip**
>
> You can also use this to add resource lines that need no value. `#SBATCH --exclusive` for slurm, for example.

31.1.3 Accessing Resources

While the `Resources` object functions primarily as a `list`, you can also access the `Resource` objects like you would a dict.

To demonstrate this, lets go back to a simpler parser format for the sake of brevity.

..note:: The key must be that of the `name` parameter, not the actual assignment.

[9]:
```
class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.submitter = "qsub"
        self.shebang = "#!/bin/bash"
        self.pragma = "#PBS"

        self.mpi = Resource(name="mpi_per_node", flag="ppn", default=4)
        self.omp = Resource(name="omp", flag="cpus-per-task", default=4)
        self.nodes = Resource(name="nodes", flag="nodes", default=1)
```

(continues on next page)

(continued from previous page)

```
        self.time = Resource(name="time", flag="walltime", format="time",␣
↪optional=False, default=3600)

    def parser(self, resources: "Resources") -> list:
        output = []
        for resource in resources:
            if resource:
                output.append(resource.resource_line)

        print(f"the value of the 'mpi' arg is {resources['mpi_per_node']}")

        return output

test = Computer()
test.mpi = 16
print(test.script())
```

```
the value of the 'mpi' arg is 16
#!/bin/bash
#PBS --ppn=16
#PBS --cpus-per-task=4
#PBS --nodes=1
#PBS --walltime=01:00:00
```

31.1.4 run_args

You can also access the `run_args` of the calling `Dataset` from within `resources`.

This is useful for extracting info such as the remote directory. For this we should create a Dataset:

[10]:
```
from remotemanager import Dataset

class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.submitter = "qsub"
        self.shebang = "#!/bin/bash"
        self.pragma = "#PBS"

        self.mpi = Resource(name="mpi_per_node", flag="ppn", default=4)
        self.omp = Resource(name="omp", flag="cpus-per-task", default=4)
        self.nodes = Resource(name="nodes", flag="nodes", default=1)
        self.time = Resource(name="time", flag="walltime", format="time",␣
↪optional=False, default=3600)

    def parser(self, resources: "Resources") -> list:
        output = []
        for resource in resources:
            if resource:
                output.append(f"{resources.pragma} --{resource.flag}={resource.value}
↪")

        output.append(f"export WORKDIR={resources['run_args']['remote_dir']}")
```

(continues on next page)

(continued from previous page)

```
        return output


def f():
    return

ds = Dataset(f, url = Computer(), skip = False, mpi_per_node=32)

ds.append_run()

ds.run(dry_run=True)

print("\njobscript:")
print(ds.runners[0].jobscript.content)
```

```
appended run runner-0
Running Dataset
assessing run for runner dataset-06a84b6d-runner-0... running
launch command: cd temp_runner_remote && rm -f dataset-06a84b6d.manifest && sed -i -e
↪"s#{rootdir}#$(pwd)#" dataset-06a84b6d-repo.sh && source dataset-06a84b6d-repo.sh &&
↪ exec_and_log bash dataset-06a84b6d-master.sh

jobscript:
#!/bin/bash
#PBS --ppn=4
#PBS --cpus-per-task=4
#PBS --nodes=1
#PBS --walltime=01:00:00
export WORKDIR=temp_runner_remote


export DIR_e711be1e={run_rootdir}
source {run_rootdir}/dataset-06a84b6d-repo.sh

exec_and_log python dataset-06a84b6d-runner-0-run.py || write_to_log failed
```

The the `run_args` of the Dataset are available at `run_args` as a dict.

It may be safer in this case to use a `get(..., None)` as if the arg is not present it will cause your parser to Fail.

CHAPTER

THIRTYTWO

STORING AND SERIALISING COMPUTERS

A Computer class defines an unchanging machine.

Aside from minor tweaks and updates, the broad sense of the jobscript that a machine requries remains the same across its lifespan.

Therefor it is reasonable to assume that once a Computer is defined to the best it can be, then it should become the "de facto" definition for that machine.

To help with this, you can store, save, and send your computers wherever you wish.

32.1 Dict and YAML Formats

One of the features that `BaseComputer` provides is a serialisation method to "reduce" a Computer definition down to a dictionary (and YAML) format.

The most used function for this is `to_yaml` (and `from_yaml` for loading).

Pass it with a filename, and the Computer spec will be dumped to a file. This can be sent anywhere and unpackaged on a machine that also has `remotemanager` installed.

> **Note**
>
> By default, resources store their `value` alongside their `default` (if possible). To create a true "blank" machine with just `default` specified, you can pass the optional argument `collect_values=False`. This is accepted by `to_yaml` and `to_dict`.

Lets redefine our machine to demonstrate:

[1]:
```
from remotemanager.connection.computers import BaseComputer, Resource

class Computer(BaseComputer):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.submitter = "sbatch"
        self.shebang = "#!/bin/bash"
        self.pragma = "#SBATCH"

        self.mpi = Resource(name="mpi", flag="ntasks", min=1)
        self.omp = Resource(name="omp", flag="cpus-per-task", min=1, max=64)
        self.nodes = Resource(name="nodes", flag="nodes", optional=False)
        self.time = Resource(name="time", flag="walltime", optional=False, format=
↪"time", default=3600)
```

```
[2]: test = Computer(host="remote.connection")
```

```
[3]: test.to_yaml("test_computer.yaml")
```

```
[4]: new = BaseComputer.from_yaml("test_computer.yaml")
```

```
[5]: new.resources
```

```
[5]: ['mpi', 'nodes', 'omp', 'time']
```

```
[6]: new.nodes = 1

print(new.script())
```

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --walltime=01:00:00
```

32.1.1 spec Dicts

This serialisation method is "open". Meaning that the output is not obfuscated, and can be edited in the "stored" form. This also allows for you to create computers directly from a dictionary/yaml file if you so choose.

If you don't want to deal with the required python wrappers, you can create a computer this way. Lets recreate this computer.

```
[7]: spec = {
    "submitter": "sbatch",
    "shebang": "#!/bin/bash",
    "pragma": "#SBATCH",
    "resources": {
        "mpi": {"flag": "ntasks", "min": 1, "default": 4},
        "omp": {"flag": "cpus-per-task", "min": 1, "max": 64,  "default": 4},
        "nodes": {"flag": "nodes", "default": 1, "optional": False},
        "time": {"flag": "walltime", "format": "time", "optional": False, "default":␣
↪3600},
    },
}

dictComputer = BaseComputer.from_dict(spec)
```

Here we have generated the base computer with the submitter, shebang and pragma set as before.

The advantages to doing it this way are twofold.

Firstly, you don't need to bother with the boilerplate `class ... def __init__ ... super() ...`, etc.

Secondly, the `name` keyword is *not* required for this method. It is extracted from the key at which it is set. So in this instance `"mpi": {"flag": "ntasks"}` is equvialent to `self.mpi = Resource(name="mpi", flag="ntasks")`

Important

These dictionaries will be used to internally create `Resource` objects for you. You can pass any of the args that `Resource` can take.

```
[8]: dictComputer.resources
```

```
[8]: ['mpi', 'nodes', 'omp', 'time']
```

```
[9]: dictComputer.missing
```

```
[9]: []
```

```
[10]: dictComputer.time = 7200

print(dictComputer.script())
```

```
#!/bin/bash
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=4
#SBATCH --nodes=1
#SBATCH --walltime=02:00:00
```

32.1.2 Creating a parser with a dict

Just like with resources, you can specify the parser when using a `dict` to create your `Computer`.

In this instance, you would create the function separately, and then add it as an argument.

> **Note**
>
> Things to note here are that we define the parser *without* `self`, and that the object itself is added to the dictionary under `"parser": parser`. `"parser": parser()` will not work.

```
[11]: def parser(resources: "Resources") -> list:
          output = []
          for resource in resources:
              if resource and resource.name not in ["mpi_per_node", "nodes", "time"]:
                  output.append(f"{resources.pragma} -{resource.flag} {resource.value}")
          # extract the values and format them before adding
          ppn = resources["mpi_per_node"]
          nodes = resources["nodes"]
          wtime = resources["time"]
          output.append(
              f"{resources.pragma} -l nodes={nodes}:"
              f"ppn={ppn},"
              f"walltime={wtime}"
          )

          output.append("\ncd $PBS_O_WORKDIR")
          output.append(f"export OMP_NUM_THREADS={resources['omp']}")

          return output

      spec = {
          "pragma": "#PBS",
          "shebang": "#!/bin/bash",
          "submitter": "qsub",
          "resources": {
              "mpi_per_node": {
```

(continues on next page)

(continued from previous page)

```
            "flag": "ppn",
            "default": 4
        },
        "nodes": {
            "flag": "nodes",
            "default": 1
        },
        "omp": {
            "flag": "cpus-per-task",
            "default": 4
        },
        "time": {
            "flag": "walltime",
            "optional": False,
            "format": "time"
        }
    },
    "parser": parser
}

test = BaseComputer.from_dict(spec)
test.time = 1800
print(test.script())
```

```
#!/bin/bash
#PBS -cpus-per-task 4
#PBS -l nodes=1:ppn=4,walltime=00:30:00

cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=4
```

32.2 generate_cell

You can also dump a `Computer` to "code" by calling `Computer.generate_cell()`. This will print out a string that you can copy-paste into a cell, giving you an editable dict form.

[12]:
```
test.generate_cell()
```

```
# Copy the following into a jupyter cell or python script to generate a modifiable␣
↪source

# Parser source code
def parser(self, resources: 'Resources', *args, **kwargs) -> list:
    output = []
    for resource in resources:
        if resource and resource.name not in ["mpi_per_node", "nodes", "time"]:
            output.append(f"{resources.pragma} -{resource.flag} {resource.value}")
    # extract the values and format them before adding
    ppn = resources["mpi_per_node"]
    nodes = resources["nodes"]
    wtime = resources["time"]
    output.append(
        f"{resources.pragma} -l nodes={nodes}:"
        f"ppn={ppn},"
```

(continues on next page)

(continued from previous page)

```
        f"walltime={wtime}"
    )

    output.append("\ncd $PBS_O_WORKDIR")
    output.append(f"export OMP_NUM_THREADS={resources['omp']}")

    return output

# JSON compatibility
true = True
false = False

# spec dict

spec = {
    "timeout": 5,
    "max_timeouts": 3,
    "python": "python",
    "shebang": "#!/bin/bash",
    "quiet_ssh": true,
    "resource_tag": "--",
    "resource_separator": "=",
    "pragma": "#PBS",
    "resources": {
        "mpi_per_node": {
            "default": 4,
            "flag": "ppn"
        },
        "nodes": {
            "default": 1,
            "flag": "nodes"
        },
        "omp": {
            "default": 4,
            "flag": "cpus-per-task"
        },
        "time": {
            "value": 1800,
            "optional": false,
            "format": "time",
            "flag": "walltime"
        }
    },
    "substitutions": {},
    "internal_extra": "",
    "extra": "",
    "submitter": "qsub",
    "host": "localhost",
    "port": 22,
    "~serialisedclass~": {
        "mod": "remotemanager.connection.computers.base",
        "name": "BaseComputer"
    }
}
spec["resource_parser"] = parser
```

(continues on next page)

(continued from previous page)

```
new = BaseComputer.from_dict(spec)
```

Important

Since we're using JSON to do the dumping, there are some slight misalignments with what python expects for a dict. To get around this, there exists the "JSON compatibility" section. This just defines `true=True` and `false=False`. You can make these changes yourself and remove this section, if you wish.

See the generated API docs for specific queries and advanced usage:

CHAPTER

THIRTYTHREE

REMOTEMANAGER

33.1 remotemanager package

33.1.1 Subpackages

remotemanager.JUBEInterop package

Submodules

remotemanager.JUBEInterop.JUBEInterop module

class remotemanager.JUBEInterop.JUBEInterop.**JUBETemplate**(*template: str*, *platform: str*, ***kwargs*)

Bases: *Computer*

Extends BaseComputer to provide compatibility with JUBE platforms

classmethod from_repo(*path: str*, *branch: str = 'develop'*, *repo: str = 'https://gitlab.com/max-centre/JUBE4MaX'*, *platform_name: str = 'platform.xml'*, *template_name: str = 'submit.job'*, *local_dir: None | str = None*, ***kwargs*)

Pulls a platform and template from a gitlab repository

Files will be pulled from a combination of repo, branch and name

e.g. from_repo(

repo="https://gitlab.com/l_sim/remotemanager", branch="devel", name="tests/standard/foo"

)

This will look for the two links: https://gitlab.com/l_sim/remotemanager/-/raw/devel/tests/standard/foo/platform.xml https://gitlab.com/l_sim/remotemanager/-/raw/devel/tests/standard/foo/submit.job

A general form of the links are: {repo}/-/raw/{branch}/{name}/{platform_name} {repo}/-/raw/{branch}/{name}/{template_name}

property platform_path: str

Path to the platform file

get_unevaluated_links(*sub:* Substitution) → list

Performs a regex search for unlinked JUBE $parameters within a value

Parameters

sub – Substitution object to check

link_sub(*sub:* Substitution) → None

Performs recursive linking for target sub

remotemanager.JUBEInterop.JUBEInterop.**attempt_eval**(*value: str*)

Attempt to evaluate *value*

remotemanager.connection package

Subpackages

remotemanager.connection.computers package

Submodules

remotemanager.connection.computers.base module

class remotemanager.connection.computers.base.**BaseComputer**(***kwargs*)

Bases: *URL*

Base computer module for HPC connection management.

Extend this class for connecting to your machine

property uuid

property short_uuid

get_parser()

unreduce_args() → None

Attempts to find any arguments who have had their chains "reduced", and unreduce them back into dynamic variables

classmethod from_dict(*spec: dict*, ***url_args*)

Create a Computer class from a *spec* dictionary. The required values are:

- **resources:**
  a dict of required resources for the machine (mpi, nodes, queue, etc.)
- **resource_parser:**
  a function which takes a dictionary of {resource: Option}, returning a list of valid jobscript lines

You can also provide some optional arguments:

- **required_or:**
  list of resources, ONE of which is required. Note that this must be a _list_ of dicts, one for each or "block"
- **optional_resources:**
  as with *resources*, but these will not be stored as required values
- **optional_defaults:**
  provide defaults for the names given in optional_resources. When adding the optional arg, the optional_defaults will be checked to see if a default is provided
- **host:**
  machine hostname. Note that this _must_ be available for a job run, but if not provided within the spec, can be added later with `url.host = 'hostname'`
- **submitter:**
  override the default submitter

- **python:**
  override the default python
- **extra:**
  any extra lines that should be appended after the resource specification. Note that this includes module loads/swaps, but this can be specified on a per-job basis, rather than locking it into the *Computer*

The *resources* specification is in *notebook*:*machine* order. That is to say that the *key* is what will be required in the _notebook_, and the *value* is what is placed in the jobscript:

```
>>> spec = {'resources': {'mpi': 'ntasks'}, ...}
>>> url = BaseComputer.from_dict(spec)
>>> url.mpi = 12
>>> url.script()
>>> "--ntasks=12"
```

Parameters

- **spec** (*dict*) – input dictionary
- **url_args** – any arguments to be passed directly to the created *url*

Returns
Computer class as per input spec

property parser_source: str

to_dict(*include_extra: bool = True*, *include_version: bool = True*, *collect_values: bool = True*) → dict
Generate a spec dict from this Computer

Parameters

- **include_extra** – includes the *extra* property if True (default True)
- **include_extra** – includes the current remotemanager version if True (default True)
- **collect_value** – Also collects the stored values of the arguments if True

Returns
dict

classmethod from_yaml(*filepath: str*, ***url_args*)
Create a Computer from *filepath*.

Parameters

- **filepath** – path containing yaml computer spec
- ****url_args** – extra args to be passed to the internal URL

Returns
BaseComputer

to_yaml(*filepath: str | IO | None = None*, *include_extra: bool = True*, *include_version: bool = True*, *collect_values: bool = True*) → str | None
Dump a computer to yaml *filepath*.

Parameters

- **filepath** – path containing yaml computer spec
- **include_extra** – includes the *extra* property if True (default True)
- **collect_value** – Also collects the stored values of the arguments if True

static download_file(*file_url: str*, *filename: str*) → None

Download file at url *file_url* and write the content out to *filename*

Parameters

- **file_url** – url of file
- **filename** – name to write content to

classmethod from_repo(*name: str*, *branch: str = 'main'*, *repo: str = 'https://gitlab.com/l_sim/remotemanager-computers/'*, ***url_args*)

Attempt to access the remote-computers repo, and pull the computer with name *name*

Parameters

- **name** (*str*) – computer name to target
- **branch** (*str*) – repo branch (defaults to main)
- **repo** (*str*) – repo web address (defaults to main l_sim repo)

Returns

BaseComputer instance

generate_cell(*name: str | None = None*, *return_string: bool = False*) → None | str

Prints out copyable source which regenerates this Computer

Parameters

- **name** (*str, None*) – Optional name for new computer. Defaults to *new*
- **return_string** (*bool*) – Also returns the string if True. Defaults to False

Returns

(None, str)

property is_super

property arguments: list

property substitutions: list

property resources: list

property argument_objects: list

property substitution_objects: list

property resource_objects: list

property argument_dict: dict

property substitution_dict: dict

property resource_dict: dict

property required: list

Returns a list of required arguments

property missing: list

Returns the currently missing arguments

property valid: bool

Returns True if there are no missing attributes

parser(*resources*) → list

Default parser for use on basic "SLURM style" machines.

Will iterate over resource objects, creating a script of the format:

{pragma} –{flag}={value}

..note::
This method can (and should) be overidden for a custom parser.

Parameters
resources – Resources object, to be created by BaseComputer

Returns
list of resource lines

property extra

property template

script(***kwargs*) → str

Takes job arguments and produces a valid jobscript

Parameters
insert_stub (*add to kwargs*) – inserts the submission block stub if True (Used by Dataset)

Returns
script

Return type
(str)

apply_substitutions(*script: list*, *empty_treatment: str = 'wipe'*) → str

Apply Substitution objects to the script

Parameters

- **script** – Base script to operate on
- **empty_treatment** – Defines the default behaviour for valueless args wipe: deletes the whole line (default) ignore: skip the value, leaving the substitution object there local: deletes only the missing argument

pack(*file=None*)

classmethod unpack(*data: dict = None*, *file: str = None*, *limit: bool = True*)

remotemanager.connection.computers.base.**unpack_parser**(*parser_data*) → *Function*

Parser data can be either a string, dict or callable. Handle all and return the Function

Parameters
parser_data – stored parser

Returns
Function

remotemanager.connection.computers.base.**legacy_unpack**(*computer:* BaseComputer, *payload: dict*) → None

Handle legacy style dictionary specs

Parameters

- **computer** – BaseComputer subclass to be updated
- **payload** – spec payload

Returns
None

remotemanager.connection.computers.computer module

class remotemanager.connection.computers.computer.**Computer**(*template*, ***kwargs*)

Bases: *Script*, *URL*

Combo class that allows for connection to a machine and generating jobscripts

pack(*collect_values: bool = True*, *ignore_none: bool = True*, *prune_defaults: bool = True*, **args*, ***kwargs*) → dict

Package up this Computer to a dictionary that can be stored as a yaml file.

A note on collection: The collected values are explicitly stated in *package_collect*. Automated collection is possible, but not feasible for "human readable" outputs

__dict__ collects the internal variables So instead of *user*, you get the *_conn* dictionary

Using dir() is an option, but collects far too many variables, and also has the possibility to accidentally call functions as it crawls the object

Parameters

- **collect_values** (*bool*) – Also collect any stored values if True. Defaults to True.
- **ignore_none** (*bool*) – skip any None values in serialisation. Defaults to True.
- **prune_defaults** (*bool*) – Collects only non-default values if True. Defaults to True.

Returns
serialised output

Return type
dict

classmethod unpack(*data: dict*, ***kwargs*) → *Computer*

Re-create an object from a packaged payload coming from `obj.pack`

> **Note**
>
> use this function to unpack from a payload _outside_ an object
>
> ```
> newobj = MyObject.unpack(payload)
> ```
>
> Where `MyObject` is a subclass of SendableMixin, and `payload` is a dict-type coming from `MyObject.pack()`

Parameters

- **data** (*dict*) – __dict__ payload from the object that was packaged
- **file** (*str*) – filepath to unpack from, if data is not given
- **limit** (*bool*) – set False to allow outside classes to be unserialised

Returns
re-created object

to_dict(**args*, ***kwargs*) → dict

classmethod from_dict(*data: dict*, ***kwargs*) → *Computer*

to_yaml(*file: str = None*, ***kwargs*) → None | str

Dump the computer to yaml. Returns the yaml content as a string if file is None

classmethod from_yaml(*file: str = None*, *data: str = None*, ***kwargs*) → *Computer*

remotemanager.connection.computers.dynamicvalue module

DynamicValue stub class allows for deferred calculation of values.

Constructing a "tree" of values using these objects allows for later assessment. Used in Computers for dynamic resource assignment.

```
>>> val_a = DynamicValue(10)
>>> val_b = DynamicValue(6)
>>> val_c = DynamicValue(val_a + val_b)
>>> val_c.value
16
```

class remotemanager.connection.computers.dynamicvalue.**ChainingMixin**

Bases: `object`

Adds chaining ability for DynamicMixin and DynamicValue

property shortform_op: str | None

Returns the operator in a readable form for calc insertion

eg +, -, * instead of add, sub, mul, etc.

property value: NotImplemented

class remotemanager.connection.computers.dynamicvalue.**DynamicMixin**(*assignment: str | None = None*, *default: Any | None = None*, *value: Any | None = None*, *optional: bool = True*, *requires: str | list | None = None*, *replaces: str | list | None = None*, *hidden: bool = False*, *min: int | None = None*, *max: int | None = None*, *format: str | None = None*, *empty_treatment: str = 'line'*, *static: bool = False*)

Bases: *ChainingMixin*

Provides functions to enable Entities using DynamicValue to chain properly

Important

The DynamicValue in question must be directly available at *_value*

Parameters

- **assignment** (*str*) – The variable to which this object is assigned, for introspection
- **default** (*Any, None*) – Default value, marks this Resource as optional if present

- **value** (*Any, None*) – Sets the value directly. You should ideally set the default, as this is easy to override and break
- **optional** (*bool*) – Marks this resource as Optional. Required as there are actually three states:
  - Required input, required by scheduler.
  - Optional input, required by scheduler.
  - Optional input, optional by scheduler.
- **requires** (*str, list*) – Stores the name(s) of another variable which is required alongside this one
- **replaces** (*str, list*) – Stores the name(s) of another variable which is replaced by this one
- **hidden** (*bool*) – The value will be present, but requests to not be displayed
- **min** (*int*) – Minimum value for numeric inputs
- **max** (*int*) – Maximum value for numeric inputs
- **format** (*str*) – Expected format for number. Allows None, "time" or "float"
- **empty_treatment** (*bool*) – Dictates how an empty value is treated. Possible options are: - line

  default, removes the whole line

  - **local**
    locally removes the output
  - **ignore**
    does nothing, leaves the output unchanged
- **static** (*bool*) – Does not chain with other values if True

hidden

format

static

property name: str

Returns the name under which this resource is stored

property min: int | None

Minimal numeric value

property max: int | None

Maximal numeric value

property default

Returns the default, if available

property optional

Returns True if this Resource is optional at Dataset level

property replaces: list

List of arguments whom are no longer considered *required* if this resource is specified

property requires: list

List of requirements if this resource is specified. e.g. nodes for mpi_per_node

property empty_treatment

Dictates how this parameter is treated if it is empty.

Possible behaviours:

line:
This is the default, and deletes the whole line. Useful when parameterising resource requests such as

`#SBATCH nodes=#nodes#`

local:
Locally deletes the value, leaving the remainder of the line. Note that this does not delete anything _outside_ of the extents of the parameter. For example:

`#SBATCH nodes=#NODES:empty_treatment=local#` -> `#SBATCH nodes=`

Useful when arguments may follow one another:

`#a:empty_treatment=local# #b:empty_treatment=local#`

In this case, a missing `a` or `b` will not delete the other

ignore:
Disables any treatment, values will be left as is:

`#a=#a:empty_treatment=local#` -> `#a=#a:empty_treatment=local#`

property value

Attempt to safely return the value (default) from self

property temporary_value

reset_temporary_value()

Reset the temporary value back to None

property linked: bool

Returns True if this Value has been linked by Script

set_value(*value*)

Sets the value, separating out the function allows for property overloading

Since this function handles value setting for both Resource/Substitution AND the DynamicValues within, we have some extra edge cases to catch

case 1
We have a resource, and are setting the value to a static int

case 2
We have a resource and are setting the value to directly mirror another resource

case 3
We have a resource and are setting the value to be a combination of other resources (DV)

property has_value: bool

property reduced

pack(*collect_value: bool = True*)

Packs this Dynamic object down to a dictionary for storage

Parameters
collect_value – Also collects the stored value if True

processed

class remotemanager.connection.computers.dynamicvalue.**DynamicValue**(*a: Number | DynamicValue | None*, *b: Number | DynamicValue | None = None*, *op: str | None = None*, *default: Number | DynamicValue | None = None*, *assignment: str | None = None*)

Bases: *ChainingMixin*

Parameters

- **a** – "First" number in operation
- **b** – "Second" number in operation. Can be None, in which case this value is considered "toplevel"
- **op** – Operation to use. Can be None for toplevel values
- **default** – Default value can be set in case the primary value is set to None

temporary_value

property a

Returns: Value of "first" number

property b

Returns: Value of "second" number

property op

Returns: Operation string

property default

Returns: The default value

property assignment: str | None

The variable at which this value is assigned, if available

property static: bool

Returns True if this Dynamic variable is static, rather than dynamic

property value

Calculates value by calling the whole chain of numbers

Returns
Value

property reduced: str

Returns the string form "reduced" version of this DynamicValue.

In theory this should be storable as text within a database, without losing dependency information

Extracts details from the assignment, if provided. Else returns the value.

e.g. .. code:: python

a = Resource(name="a") b = Resource(name="b") c = Resource(name="c")

c = a + b

c.reduce > "(a + b)"

remotemanager.connection.computers.dynamicvalue.**treat_for_storage**(*var: Any*) → str

Ensures that var is stored properly

Strings must be quoted, but _not_ those that are actually ints

remotemanager.connection.computers.dynamicvalue.**concat_basic**(*a: int* | DynamicValue | Resource | Substitution, *b: int* | DynamicValue | Resource | Substitution) → *DynamicValue*

Concat two values a and b

Required in the case where we want to add a value to a string

Since str + Resource will call the __add__ method of the str object, we get a TypeError

We need to sidestep this by adding the string to the _value_, then reversing the value.

Doing it this way calls the corrent DynamicMixin.__add__(...) function, rather than str.__add__(...)

..note::
This function will at least _try_ to return a+b beforehand

Parameters

- **a** – first value
- **b** – second value

Returns
a+b

remotemanager.connection.computers.dynamicvalue.**entry_format**(*val*, *format: str* | *None* = *None*)

remotemanager.connection.computers.example module

class remotemanager.connection.computers.example.**ExampleTorque**(***kwargs*)

Bases: *BaseComputer*

example class for connecting to a remote computer using a torque scheduler

parser(*resources*) → list

Example parser for a torque based computer

Parameters
resources – resources dictionary, provided by BaseComputer

Returns
list of resource request lines

class remotemanager.connection.computers.example.**ExampleSlurm**(***kwargs*)

Bases: *BaseComputer*

example class for connecting to a remote computer using a slurm scheduler

remotemanager.connection.computers.parsers module

remotemanager.connection.computers.parsers.**slurm**(*resources:* Resources) → list

Example parser for a slurm based computer

Parameters
resources – resources dictionary, provided by BaseComputer

Returns
list of resource request lines

remotemanager.connection.computers.parsers.**torque**(*resources:* Resources) → list

Example parser for a torque based computer

Parameters
resources – resources dictionary, provided by BaseComputer

Returns
list of resource request lines

remotemanager.connection.computers.resource module

This module stores the placeholder arguments who's job it is to convert arguments from the Dataset level *mpi*, *omp*, *nodes*, etc. to what the scheduler is expecting within a jobscript.

Note

Placeholders without a *value* are "falsy". So checking their value in an if statement will return *True* if they have a value, *False* otherwise.

class remotemanager.connection.computers.resource.**Resource**(*name: str*, *flag: str | None = None*, *tag: str | None = None*, *separator: str | None = None*, ***kwargs*)

Bases: *DynamicMixin*

Stub class to sit in place of an option within a computer.

Parameters

- **name** (*str*) – name under which this arg is stored
- **flag** (*str*) – Flag to append value to e.g. *–nodes*, *–walltime*
- **separator** (*str*) – Override the separator between flag and value (defaults to "=")
- **tag** (*str*) – Override the tag preceding the flag (defaults to "–")

Note

For other args see the DynamicMixin class

flag

pragma

tag

separator

property resource_line: str

Shortcut to output a suitable resource request line

Returns
resource request line

Return type
str

pack(*collect_value: bool = True*) → dict

Store this Resource in dict form

class remotemanager.connection.computers.resource.**runargs**(**args*, ***kwargs*)

Bases: dict

Class to contain the dataset run_args in a way that won't break any loops over the resources

Parameters
args (*dict*) – Dataset run_args

property value

Prevents an AttributeError when a parser attempts to access the value.

Returns
internal dict

Return type
(dict)

property flag

Parsers should not access the flag method of the run_args, doing so likely means that a loop has iterated over this object and is attempting to insert it into a jobscript.

Converts an AttributeError to one more tailored to the situation.

Returns
RuntimeError

class remotemanager.connection.computers.resource.**Resources**(*resources*, *pragma*, *tag*, *separator*, *run_args*)

Bases: object

Container class to store Resource objects for use by a parser

pragma

get(*name: str*, *default: any = '_unspecified'*)

Allows resource.get(name)

items()

dict.items() like proxy

property run_args: dict

Returns the stored run_args

remotemanager.connection.computers.substitution module

class remotemanager.connection.computers.substitution.**Substitution**(*target: str*, *name: str*, *mode: str | None = None*, ***kwargs*)

Bases: *DynamicMixin*

Stores a jobscript template substitution

Parameters

- **target** – String to _replace_ in the template. Ideally should begin with a commenting character (#)
- **name** – String to replace _with_. At script generation, *target* will be replaced with *name*
- **mode** – Used by JUBETemplate

Note

For other args see the DynamicMixin class

target

mode

executed

dependencies

classmethod from_string(*string: str*, *warn_invalid: bool = True*, ***kwargs*) → *Substitution*

Create a substitution object from template string

Parameters

- **string** (*str*) – Input string to generate from
- **warn_invalid** (*bool*) – Invalid args name, target will be deleted. Warn if True
- **kwargs** – Any keyword args to override the string with

property target_kwargs

Attempts to extract the kwargs from the target string

static get_target_kwargs(*string: str*) → dict

Attempts to generate kwargs from input string

property value: any

Returns: value if present, else default

property arg: str

Returns: Argument which is exposed to the underlying URL for setting

property entrypoint: str

Returns the name/name for this sub

pack(*collect_value: bool = True*) → dict

Store this Substitution in dict form

remotemanager.connection.computers.utils module

remotemanager.connection.computers.utils.**format_time**(*time: int | float | str*) → str | None

Take integer seconds and generate a HH:MM:SS timestring

Parameters
time (*int*) – seconds

Returns
HH:MM:SS format timestamp

Return type
(str)

remotemanager.connection.computers.utils.**time_to_string**(*time: int*) → str

Converts integer seconds to HH:MM:SS format

remotemanager.connection.computers.utils.**time_to_s**(*time: str*) → int

Convert back from HH:MM:SS to integer seconds

remotemanager.connection.computers.utils.**semantic_to_int**(*time: str*) → int

Convert "semantic" time strings to integer format

i.e. 24h => 86400, 30m => 1800

remotemanager.connection.computers.utils.**try_value**(*inp: Any*) → Any

Try to access the value property

This _needs_ to be used in the presence of DynamicValues, since they override a lot of basic functions. This can cause recursion errors if not handled properly

Submodules

remotemanager.connection.cmd module

remotemanager.connection.cmd.**detect_locale_error**(*stderr*)

Given a stderr output string *stderr*, will regex search for the locale errors

Parameters
stderr – stderr text

Returns
True if the error appears to be a locale issue

Return type
bool

class remotemanager.connection.cmd.**CMD**(*cmd: str*, *asynchronous: bool = False*, *stdout: str = None*, *stderr: str = None*, *stream: bool = False*, *timeout: int = 5*, *max_timeouts: int = 3*, *raise_errors: bool = True*, *force_file: bool = False*, *verbose: None | int | bool | Verbosity = None*)

Bases: *SendableMixin*

This class stores a command to be executed, and the returned stdout, stderr

Parameters

- **cmd** (*str*) – command to be executed
- **asynchronous** (*bool*) – execute commands asynchronously defaults to False
- **stdout** (*str*) – optional file to redirect stdout to
- **stderr** (*str*) – optional file to redirect stderr to
- **stream** (*bool*) – enables output streaming if True
- **timeout** (*int*) – time to wait before issuing a timeout
- **max_timeouts** (*int*) – number of times to attempt communication in case of a timeout
- **force_file** (*bool*) – always use the fexec method if True

property verbose: *Verbosity*

Verbose property

property uuid

property short_uuid

property tempfile

property sent: str

The command passed at initialisation

property cmd: str

Alias for init command

property asynchronous: bool

True if commands are to be executed asynchronously

property is_redirected: bool

True if the cmd is redirected to a file

property redirect

property cached: bool

property stdout: str

Directly returns the stdout from the cmd execution. Attempts to communicate with the subprocess in the case of an async run.

Returns None if the command has not been executed yet.

Returns (str):
the stdout from the command execution

property stderr: str

Directly returns the stderr from the cmd execution. Attempts to communicate with the subprocess in the case of an async run.

Returns None if the command has not been executed yet.

Returns (str):
the stdout from the command execution

property pwd: str

Present working directory at command execution

Returns None if the command has not been executed yet.

Returns (str):
working dir of command execution

property whoami: str

Present user at command execution

Returns None if the command has not been executed yet.

Returns (str):
username who executed the command

property pid: int

The Process ID of the spawned process

Returns None if the command has not been executed yet.

Returns (int):
the PID of the spawned shell for this command

property returncode: [<class 'int'>, None]

Attempt to retrieve the returncode of the subprocess. This call will not disturb an asynchronous run, returning None

Returns (int, None):
The exit status of the subprocess, None if it is still running. None otherwise.

property is_finished: bool

Returns True if this command has finished execution. This will NOT talk to the process, as to not disturb async runs, so will always return False in those instances

Returns (bool):
True if the command has completed

property succeeded: [None, <class 'bool'>]

True if the command successfully executed

Returns
None if not finished, True if returncode is 0

property duration

property latency

exec(*verbose: None | int | bool | Verbosity = None*) → None
Executes the command, storing execution info and in the case of a non-async run; returned values

Returns
None

communicate(*use_cache: bool = True*, *ignore_errors: bool = None*, *verbose: None | int | bool | Verbosity = None*) → dict
Communicates with the subprocess, returning the stdout and stderr in a dict

Parameters

- **use_cache** (*bool*) – use cached value if it is available
- **ignore_errors** (*bool*) – do not raise error regardless of base setting

Returns (dict):
{'stdout': stdout, 'stderr': stderr}

kill(*pid: int = None*, *verbose: None | int | bool | Verbosity = None*) → None
Kill the process associated with this command, if one exists

Returns
None

remotemanager.connection.testing_object module

class remotemanager.connection.testing_object.**ConnectionTest**(*parent*)
Bases: *SendableMixin*

Object to store an instance of a connection test.

Runs the tests, storing any important information

Parameters
parent – parent URL

property parent

exec() → None
Execute the connection test

property passed

property data

property extra

property latency

test_basic() → bool
Connects to the host and returns the entry directory

Returns
True if test succeeded

Return type
(bool)

test_files() → bool

Attempts to create and delete files in several directories

Returns
True if test succeeded

Return type
(bool)

test_transport() → bool

Tests all available transport methods, returning True if at least one is functional

Returns
True if test succeeded

Return type
(bool)

remotemanager.connection.url module

URL base class for connecting to remote systems

class remotemanager.connection.url.**URL**(*host: str = None*, *user: str = None*, *port: int = None*, *verbose: None | int | bool | Verbosity = 1*, *timeout: int = 5*, *max_timeouts: int = 3*, *python: str = 'python'*, *submitter: str = 'bash'*, *shell: str = 'bash'*, *raise_errors: bool = True*, *keyfile: str = None*, *passfile: str = None*, *envpass: str = None*, *sshpass_override: str = None*, *cmd_history_depth: int = 10*, *landing_dir: str = None*, *ssh_insert: str = ''*, *ssh_prepend: str = None*, *ssh_override: str = None*, *quiet_ssh: bool = True*, *shebang: str = '#!/bin/bash'*, *transport: None | Transport = None*, ***kwargs*)

Bases: object

Container to store the url info for a Remote run

The url should contain everything pertaining to the _remote_, allowing Dataset to be remote-agnostic

Parameters

- **host** (*str*) – host address of the remote system
- **user** (*str*) – username for the remote system
- **port** (*int, str*) – port to connect to for ssh tunnels
- **verbose** (*bool*) – base-verbosity for connections
- **timeout** (*int*) – time to wait before issuing a timeout for cmd calls
- **max_timeouts** (*int*) – number of times to attempt cmd communication in case of a timeout
- **python** (*str*) – string used to initiate a python instance
- **raise_errors** (*bool*) – set false to ignore errors by default in cmd calls
- **passfile** (*str*) – absolute path to password file for sshpass calls
- **envpass** (*str*) – environment variable containing absolute path to password file for
- **sshpass_override** (*str*) – override the sshpass string sshpass calls
- **cmd_history_depth** (*int*) – number of cmd calls to store history for
- **landing_dir** (*str*) – set the directory which is treated as the ssh endpoint
- **ssh_insert** (*str*) – any extra flags you wish to add to the ssh call

- **quiet_ssh** (*bool*) – Option to add -q flag to ssh calls. This suppresses errors and warnings that machines can sometimes put on stderr, breaking runs. Defaults True
- **kwargs** – any extra args that may end up here from a Dataset or Computer are discarded

property verbose: *Verbosity*

Verbose property

property uuid

property short_uuid

property raise_errors: bool

property ignore_errors: bool

property call_count: int

property user: str

Currently configured username

property host: str

Currently configured hostname

property port: int

Currently configured port (defaults to 22)

property userhost: str

user@host string if possible, just *host* if user is not present

property ssh_insert: str

property ssh_prepend: str

property submitter

property shell

property home: str

gethome() → str

clearhome()

Clear the home property

Returns
None

property landing_dir

property transport

property ssh: str

ssh insert for commands on this connection

clear_ssh_override()

Wipe any override applied to ssh. Can also be done by setting url.ssh = None

Returns
None

property keyfile

property sshpass_override: str

property passfile: [<class 'str'>, None]

Returns the sshpass string

Added in version 0.10.2: No longer returns just the file, to facilitate overriding

Returns
sshpass full string

tunnel(*local_port: int*, *remote_port: int*, *local_address: str = None*, *background: bool = False*, *verbose: None | int | float | Verbosity = None*, *dry_run: bool = False*) → *CMD*

Create a tunnel to the host, between the local and remote ports

Parameters

- **local_port** (*int*) – port to open on the local side
- **remote_port** (*int*) – port to access on the remote side
- **local_address** (*str*) – change the local address of the tunnel
- **background** (*bool*) – creates the tunnel asynchronously if True (default False)
- **verbose** – override verbose setting for this call
- **dry_run** (*bool*) – do not execute if True> Defaults to False

Returns
The CMD instance responsible for the tunnel

Return type
CMD

property is_local

True if this connection is purely local

ping(*n: int = 5*, *timeout: int = 30*, *verbose: None | int | bool | Verbosity = None*) → float

Perform and monitor a ping command

Parameters

- **n** (*int*) – number of pings to aim for
- **timeout** (*int*) – kill the process and return 0 if this period is elapsed

Returns
(float) latency in ms

expandvars(*string: str*) → str

'echo' a string on the remote, returning the result

Parameters
string – string to be expanded

Returns
str

cmd(*cmd: str*, *asynchronous: bool = False*, *local: bool | None = None*, *stdout: str = None*, *stderr: str = None*, *timeout: int = None*, *max_timeouts: int = None*, *raise_errors: bool = None*, *dry_run: bool = False*, *prepend: bool = False*, *force_file: bool = False*, *landing_dir: str = None*, *stream: bool = False*, *verbose: None | int | bool | Verbosity = None*) → *CMD*

Creates and executes a command

Parameters

- **asynchronous** (*bool*) – run this command asynchronously
- **cmd** (*str*) – command to execute
- **local** (*bool, None*) – force a local or remote execution. Defaults to None

- **stdout** (*str*) – optional file to redirect stdout to
- **stderr** (*str*) – optional file to redirect stderr to
- **timeout** (*int*) – time to wait before issuing a timeout
- **max_timeouts** (*int*) – number of times to attempt communication in case of a timeout
- **raise_errors** (*bool*) – override for global setting. Raise any stderr if encountered
- **dry_run** (*bool*) – don't exec the command if True, just returns the string
- **prepend** (*bool*) – always attempt to use ssh_prepend
- **force_file** (*bool*) – passthrough for CMD force_file argument
- **landing_dir** – set the directory which is treated as the ssh endpoint
- **stream** – Attempts to stream the output as it arrives on stdout

Returns (CMD):
returned command instance

property cmd_history

reset_cmd_history()

property cmd_history_depth

property utils
Handle for the URLUtils module

test_connection() → None
Create a ConnectionTest instance and run the tests

Returns
None

property connection_test: *ConnectionTest* | None
Return the connection test object

Returns
testing object

Return type
ConnectionTest

property connection_data: dict
Returns the results of a previous connection test

Returns
(dict) connection data

property latency: [<class 'float'>, None]
Attempts to access the latency property of the stored ConnectionTest

Returns
connection latency in seconds, if available. Else None

Return type
(float)

script(***kwargs*)

static download_file(*file_url: str*, *filename: str*) → None

Download file at url *file_url* and write the content out to *filename*

Parameters

- **file_url** – url of file
- **filename** – name to write content to

class remotemanager.connection.url.**URLUtils**(*parent:* URL)

Bases: object

Extra functions to go with the URL class, called via URL.utils

As it requires a parent *URL* to function, and is instantiated with a *URL*, there is little to no purpose to using this class exclusively

Parameters
parent (URL) – parent class to provide utils to

file_mtime(*files: list*, *local: bool = None*, *python: bool = False*, *ignore_empty: bool = False*, *dry_run: bool = False*) → dict

Check file modification times of [files]

Parameters

- **files** (*list*) – list of paths to files
- **local** (*bool*) – force a local search
- **python** (*bool*) – ensure python style search is used
- **ignore_empty** (*bool*) – also check the filesize, ignoring empty files
- **dry_run** (*bool*) – print command only

Returns (dict):
{file: mtime (unix)} dictionary

file_presence(*files: list*, *local: bool = None*, *dry_run: bool = False*) → dict

Search for a list of files, returning a boolean presence dict

Parameters

- **files** (*list*) – list of paths to files
- **local** (*bool*) – force a local search
- **dry_run** (*bool*) – print command only

Returns (dict):
{file: present} dictionary

search_folder(*files: list*, *folder: str*, *local: bool = None*, *dry_run: bool = False*) → dict

Search *folder* for *files*, returning a boolean presence dict

Parameters

- **files** (*list*) – list of filenames to check for. Optionally, a string for a single file
- **folder** (*str*) – folder to scan
- **local** (*bool*) – perform the scan locally (or remotely)
- **dry_run** (*bool*) – print command only

Returns (dict):
{file: present} dictionary

touch(*file: str*, *local: bool = None*, *raise_errors: bool = None*, *dry_run: bool = False*) → *CMD*

perform unix *touch*, creating or updating *file*

Parameters

- **file** (*str*) – filename or path to file
- **local** (*bool*) – force local (or remote) execution
- **raise_errors** (*bool*) – raise any stderr encountered
- **dry_run** (*bool*) – print command only

Returns (CMD):
CMD instance for the command

mkdir(*file: str*, *local: bool = None*, *raise_errors: bool = None*, *dry_run: bool = False*) → *CMD*

perform unix *mkdir -p*, creating a folder structure

Parameters

- **file** (*str*) – name or path to folder
- **local** (*bool*) – force local (or remote) execution
- **raise_errors** (*bool*) – raise any stderr encountered
- **dry_run** (*bool*) – print command only

Returns (CMD):
CMD instance for the command

ls(*file: str*, *as_list: bool = True*, *local: bool = None*, *raise_errors: bool = None*, *dry_run: bool = False*) → [<class 'remotemanager.connection.cmd.CMD'>, <class 'list'>]

Identify the files present on the directory

Parameters

- **file** (*str*) – name or path to folder.
- **as_list** (*bool*) – convert to a list format
- **local** (*bool*) – force local (or remote) execution
- **raise_errors** (*bool*) – raise any stderr encountered
- **dry_run** (*bool*) – print command only

Returns (CMD, list):
CMD instance for the command, or the list if as_list is True

remotemanager.dataset package

Submodules

remotemanager.dataset.dataset module

Main Dataset module

This is the primary class used by the user

class remotemanager.dataset.dataset.**Dataset**(*function: ~typing.Callable | str | None*, *url: ~remotemanager.connection.url.URL = None*, *dbfile: str = None*, *transport: ~remotemanager.transport.transport.Transport = None*, *serialiser: <module 'remotemanager.serialisation.serial' from '/home/test/remotemanager/remotemanager/serialisation/serial.py'> = None*, *script: str = None*, *shebang: str = None*, *name: str = None*, *extra_files_send: list = None*, *extra_files_recv: list = None*, *verbose: None | int | bool | ~remotemanager.logging_utils.verbosity.Verbosity = None*, *add_newline: bool = True*, *skip: bool = True*, *extra: str = None*, ***global_run_args*)

Bases: *SendableMixin*, *ExtraFilesMixin*

Bulk holder for remote runs. The Dataset class handles anything regarding the runs as a group. Running, retrieving results, sending to remote, etc.

Parameters

- **function** (*Callable, str, None*) – Function to run. Can either be the function object, source string or None If None, Runner will pass arguments to the *script* method
- **url** (URL) – connection to remote (optional)
- **transport** (*tp.transport.Transport*) – transport system to use, if a specific is required. Defaults to transport.rsync
- **serialiser** (serial.serial) – serialisation system to use, if a specific is required. Defaults to serial.serialjson
- **script** (*str*) – callscript required to run the jobs in this dataset
- **submitter** (*str*) – command to exec any scripts with. Defaults to "bash"
- **name** (*str*) – optional name for this dataset. Will be used for runscripts
- **extra_files_send** (*list, str*) – extra files to send with this run
- **extra_files_recv** (*list, str*) – extra files to retrieve with this run
- **skip** (*bool*) – skip dataset creation if possible. Defaults True
- **extra** – extra text to insert into the runner jobscripts
- **global_run_args** – any further (unchanging) arguments to be passed to the runner(s)

default_url

a default url can be assigned to all Datasets.

Type
URL

default_url = None

property verbose: *Verbosity*

Verbose property

classmethod recreate(**args*, *raise_if_not_found: bool = True*, ***kwargs*)

Attempts to extract a dataset matching the given args from the python garbage collection interface

Parameters

- **raise_if_not_found** (*bool*) – raise ValueError if the Dataset was not found
- ***args** – args as passed to Dataset
- ****kwargs** – keyword args as passed to Dataset

Returns
Dataset

classmethod from_file(*file: str*, *url:* URL = *None*)

Alias for Dataset.unpack(file=...)

Parameters

- **file** (*str*) – Dataset dbfile
- **url** (URL) – the URL to apply to this Dataset

Returns
unpacked Dataset

Return type
(*Dataset*)

property database: *Database*

Access to the stored database object. Creates a connection if none exist.

Returns (Database):
Database

property dbfile: str

Name of the database file

sanitise_run_arg_paths(*run_args: dict*) → dict

Checks for issues in the paths within the given run_args

static sanitise_path(*path*) → str

Ensures a clean unix-type path

property remote_dir: str | bool

Accesses the remote_dir property from the run args. Tries to fall back on run_dir if not found, then returns default as a last resort.

property run_dir: str | bool

Accesses the remote_dir property from the run args. Tries to fall back on run_dir if not found, then returns default as a last resort.

property run_path: str | bool

Accesses the remote_dir property from the run args. Tries to fall back on run_dir if not found, then returns default as a last resort.

property local_dir: str

Accesses the local_dir property from the run args. Returns default if not found.

property repo_prefix: str

override for repo names and manifest file in a dependency situation

property repofile: *TrackedFile*

Returns the TrackedFile instance responsible for the repository

property bash_repo: *TrackedFile*

Returns the TrackedFile instance responsible for the repository

property master_script: *TrackedFile*

Returns the TrackedFile instance responsible for the master script

property manifest_log: *TrackedFile*

Returns the TrackedFile instance responsible for the manifest

property global_run_args: dict

Returns the toplevel global run args

set_run_arg(*key: str*, *val*)

Set a single run arg *key* to *val*

Parameters

- **key** – name to set
- **val** – value to set to

Returns
None

set_run_args(*keys: list*, *vals: list*)

Set a list of *keys* to `vals

> **Note**
> List lengths must be the same

Parameters

- **keys** – list of keys to set
- **vals** – list of vals to set to

Returns
None

update_run_args(*d: dict*)

Update current global run args with a dictionary *d*

Parameters
d – dict of new args

Returns
None

property do_not_recurse: bool

Internal function used for blocking recursion in dependency calls

property dependency: *Dependency* | None

Returns the stored dependency

property is_child: bool

Returns True if this dataset is a child, False otherwise

property is_parent: bool

Returns True if this dataset is a parent, False otherwise

set_downstream(*dataset*) → None

Add a child to this dataset

set_upstream(*dataset*) → None

Add a parent to this dataset

pack(*file: str = None*, ***kwargs*) → dict | None

Override for the SendableMixin.pack() method, ensuring the dataset is always below a `uuid`

Parameters
****kwargs** – Any arguments to be passed onwards to the SendableMixin.pack()

Returns
(dict) packing result

update_db(*dependency_call: bool = False*) → None

Force updates the database

set_run_option(*key: str*, *val*) → None

Update a global run option *key* with value *val*

Parameters

- **key** (*str*) – option to be updated
- **val** – value to set

append_run(*args: dict = None*, *arguments: dict = None*, *name: str = None*, *extra_files_send: list | str | None = None*, *extra_files_recv: list | str | None = None*, *dependency_call: bool = False*, *verbose: int = None*, *quiet: bool = False*, *skip: bool = True*, *force: bool = False*, *lazy: bool = False*, *chain_run_args: bool = True*, *extra: str = None*, *return_runner: bool = False*, ***run_args*)

Serialise arguments for later runner construction

Parameters

- **args** (*dict*) – dictionary of arguments to be unpacked
- **arguments** (*dict*) – alias for args
- **name** (*str*) – append a runner under this name
- **extra_files_send** (*list, str*) – extra files to send with this run
- **extra_files_recv** (*list, str*) – extra files to retrieve with this run
- **dependency_call** (*bool*) – True if called via the dependency handler
- **verbose** (*int, Verbose, None*) – verbose level for this runner (defaults to Dataset level)
- **quiet** (*bool*) – disable printing for this append if True
- **skip** (*bool*) – ignores checks for an existing runner if set to False
- **force** (*bool*) – always appends if True
- **lazy** (*bool*) – performs a "lazy" append if True, skipping the dataset update. You MUST call ds.finish_append() after you are done appending to avoid strange behaviours
- **chain_run_args** (*bool*) – for dependency runs, will not propagate run_args to other datasets in the chain if False (defaults True)
- **extra** – extra string to add to this runner
- **return_runner** – returns the appened (or matching) runner if True
- **run_args** – any extra arguments to pass to runner

insert_runner(*runner:* Runner, *skip: bool = True*, *force: bool = False*, *lazy: bool = False*, *verbose: None | int | bool |* Verbosity *= None*, *quiet: bool = False*, *return_runner: bool = False*) → None | *Runner*

Internal runner insertion.

Parameters

- **runner** – Runner object to insert
- **skip** – don't insert if it exists
- **force** – force inserts
- **lazy** – Attempts a lazy append if True (does not update DB)
- **verbose** – Verbosity level for this runner

- **quiet** – inserts runner quietly if True
- **return_runner** – Returns the runner object if True

Returns
None or Runner

finish_append(*dependency_call: bool = False*, *print_summary: bool = True*, *verbose: None | int | bool | Verbosity = None*) → None

Completes the append process by updating the database, and printing a summary if necessary

Parameters

- **dependency_call** – Will not attempt to relay to a dependency if True (called by dependency)
- **print_summary** – Prints a summary if True
- **verbose** – verbosity level for this call

lazy_append() → *LazyAppend*

Access a LazyAppend object, which handles the append finalisation

copy_runners(*dataset: Dataset*) → None

Copy the runners from dataset over to this dataset

property append_log: str

Returns the log from the previous append session

remove_run(*ident: int | str | dict*, *dependency_call: bool = False*, *verbose: None | int | bool | Verbosity = None*) → bool

Remove a runner with the given identifier. Search methods are identical get_runner(id)

Parameters

- **ident** – identifier
- **dependency_call** (*bool*) – used by any dependencies that exist, prevents recursion
- **verbose** – local verbose level

Returns
True if succeeded

Return type
(bool)

get_runner(*ident: int | str | dict*, *dependency_call: bool = False*, *verbose: None | int | bool | Verbosity = None*) → *Runner* | None

Collect a runner with the given identifier. Depending on the type of arg passed, there are different search methods:

- int: the runners[ident] of the runner to remove
- str: searches for a runner with the matching uuid
- dict: attempts to find a runner with matching args

Parameters

- **ident** – identifier
- **dependency_call** (*bool*) – used by the dependencies, runners cannot be removed via uuid in this case, as the uuids will not match between datasets

Returns
collected Runner, None if not available

Return type
(*Runner*)

wipe_runs(*dependency_call: bool = False*) → None

Removes all runners

Parameters
dependency_call (*bool*) – used by any dependencies that exist, prevents recursion

reset_runs(*wipe: bool = False*, *dependency_call: bool = False*) → None

Remove any results from the stored runners and attempt to delete their result files if *wipe=True*

> **Warning**
>
> This is a potentially destructive action, be careful with this method

Parameters

- **wipe** – Additionally deletes the local files if True. Default False
- **dependency_call** (*bool*) – used by any dependencies that exist, prevents recursion

collect_files(*remote_check: bool*, *results_only: bool = False*, *extra_files_send: bool = True*) → list

Collect created files

Parameters

- **remote_check** – search for remote paths if True
- **results_only** – only collect files that are returned from a run such as Results and extra_files_recv if True
- **extra_files_send** – collects extra_files_send if True

Returns
list of filepaths

wipe_local(*files_only: bool = True*, *dry_run: bool = False*, *dependency_call: bool = False*) → None

Clear out the local directory

Parameters

- **files_only** (*bool*) – delete individual files instead of whole folders (preserves extra files)
- **dry_run** (*bool*) – print targets and exit
- **dependency_call** (*bool*) – used by any dependencies that exist, prevents recursion

Returns
None

wipe_remote(*files_only: bool = True*, *dry_run: bool = False*, *dependency_call: bool = False*) → None

Clear out the remote directory (including run dir)

Parameters

- **files_only** (*bool*) – delete individual files instead of whole folders (preserves extra files)
- **dry_run** (*bool*) – print targets and exit
- **dependency_call** (*bool*) – used by any dependencies that exist, prevents recursion

Returns
None

hard_reset(*files_only: bool = True*, *dry_run: bool = False*, *dependency_call: bool = False*) → None

Hard reset the dataset, including wiping local and remote folders

Parameters

- **files_only** (*bool*) – delete individual files instead of whole folders (preserves extra files)
- **dry_run** (*bool*) – print targets and exit
- **dependency_call** (*bool*) – used by any dependencies that exist, prevents recursion

Returns
None

backup(*file=None*, *force: bool = False*, *full: bool = False*) → str

Backs up the Dataset and any attached results/extra files to zip file

Parameters

- **file** – target path
- **force** – overwrite file if it exists
- **full** – only collects runner results if False (defaults `False`)

Returns
path to zip file

classmethod restore(*file*, *force: bool = False*) → *Dataset*

Restore from backup file *file*

Parameters

- **file** – File to restore from
- **force** – Set to True to overwrite any existing Dataset

Returns
Dataset

property runner_dict: dict

Stored runners in dict form, where the keys are the append id

property runners: list

Stored runners as a list

property function: *Function* | *Script* | None

Currently stored Function wrapper

property is_python

property extra: str

Returns the global level extra

property shebang: str

returns the url shebang

property script: str

Currently stored run script

Parameters
sub_args – arguments to substitute into the script() method

Returns
arg-substituted script

Return type
(str)

property add_newline: bool

Returns True if add_newline is set

This controls if scripts have an additional newline enforced at the end

property submitter: str

Currently stored submission command

property shell: str

property url: *URL*

Currently stored URL object

property transport: *Transport*

Currently stored Transport system

property serialiser: *serial*

Returns the stored serialiser object

remove_database() → None

Deletes the database file

property name: str

Name of this dataset

property uuid: str

This Dataset's full uuid (64 characcter)

property short_uuid: str

This Dataset's short format (8 character) uuid

set_runner_states(*state: str*, *uuids: list = None*, *extra: str = None*) → None

Update runner states to *state*

Parameters

- **state** (*(str)*) – state to set
- **uuids** (*(list)*) – list of uuids to update, updates all if not passed

get_all_runner_states() → list

Check all runner states, returning a list

Returns (list):
states

check_all_runner_states(*state: str*) → bool

Check all runner states against *state*, returning True if *all* runners have this state

Parameters
state (*str*) – state to check for

Returns (bool):
all(states)

property last_run: int | None

Returns the unix time of the last _run call

Returns
unix time of last _run call, or None if impossible

Return type
(int)

property run_summary_limit: int

If there are more runners than this number, the run output will be summed up rather than printed

retry_failed(**args*, ***kwargs*) → None

Retries all failed runners

Takes args and kwargs, passes them to run

run(*force: bool = False*, *dry_run: bool = False*, *verbose: None | int | bool | Verbosity = None*, *uuids: list = None*, *extra: str = ''*, *force_ignores_success: bool = False*, *dependency_call: bool = False*, ***run_args*)

Run the functions

Parameters

- **force** (*bool*) – force all runs to go through, ignoring checks
- **dry_run** (*bool*) – create files, but do not run
- **verbose** – Sets local verbose level
- **uuids** (*list*) – list of uuids to run
- **extra** – extra text to add to runner jobscripts
- **failed_only** (*bool*) – If True, *force* will submit only failed runners
- **force_ignores_success** (*bool*) – If True, *force* takes priority over *is_success* check
- **dependency_call** (*bool*) – Internally used to block recursion issues with dependencies
- **run_args** – any arguments to pass to the runners during this run. will override any "global" arguments set at Dataset init

property run_log: str

Returns the output created during the run process

property run_cmd: *CMD*

Access to the storage of CMD objects used to run the scripts

Returns
List of CMD objects

Return type
(list)

check_states(*state: str*) → dict

Call the repo "last_time" method remotely

check_started() → dict

Check when runners started remotely, using the manifest

property is_finished: list

Queries the finished state of this Dataset

property is_finished_force: list

Queries the finished state of this Dataset

property all_finished: bool

Check if *all* runners have finished

Returns (bool):
True if all runners have completed their runs

property all_success: bool

Returns True if all runners report that they have succeeded

wait(*interval: int | float = 10*, *timeout: int | float = None*, *watch: bool = False*, *success_only: bool = False*, *only_runner: Runner = None*, *force: bool = False*) → None

Watch the calculation, printing updates as runners complete

Parameters

- **interval** – check interval time in seconds
- **timeout** – maximum time to wait in seconds
- **watch** – print an updating table of runner states
- **success_only** – Completion search ignores failed runs if True
- **only_runner** – wait for only this runner to complete
- **force** – Raises dataset level errors as errors if True

Returns
None

fetch_results(*results: bool = True*, *errors: bool = True*, *extras: bool = True*, *force: bool = False*, *verbose: None | int | bool | Verbosity = None*)

Fetch results from the remote, and store them in the runner results property

Parameters

- **results** – fetch result files
- **errors** – fetch error files
- **extras** – fetch extra files

Returns
None

update_runners(*runners: list | None = None*, *dependency_call: bool = False*)

Collects the manifest file, updating runners

Parameters

- **runners** – list of runners to update, usually used for dependencies
- **dependency_call** – internal flag to avoid dependecy loops

property results: list

Access the results of the runners

Returns (list):
`runner.result` for each runner

property errors: list

Access the errors of the runners

Returns (list):
`runner.error` for each runner

property failed: list

Returns a list of failed runners

Returns
list of failed runners

prepare_for_transfer() → None

Ensures that the Transport class is able to function

avoid_runtime() → None

Call for last_runtime sensitive operations such as is_finished and fetch_results

Waits for 1s if we're too close to the saved _last_run time

Returns
None

remotemanager.dataset.dataset.**line_starts_with_uuid**(*line: str*) → bool

Checks if line starts with a short uuid

returns True if line starts like "a1b2c3d4", False otherwise

remotemanager.dataset.dependency module

The Dependency module handles the chaining of Datasets.

It provides overrides for any linked Dataset to propagate required calls.

class remotemanager.dataset.dependency.**Dependency**

Bases: *SendableMixin*

Main Dependency class.

Note

This class is internal and should not be interacted with directly.

Handles the chaining of Datasets and stores the "network". It provides overrides for any linked Dataset to propagate required calls.

add_edge(*primary:* Dataset, *secondary:* Dataset) → None

Adds an "edge" to the dependency network.

This edge represents a parent-child link.

Parameters

- **primary** (Dataset) – Parent Dataset object.
- **secondary** (Dataset) – Child Dataset object.

property network: List[Tuple[*Dataset*, *Dataset*]]

Returns the internal dependency network.

This is a list of tuples representing the parent-child relationship.

Returns
List(Tuple(Dataset, Dataset)

get_children(*dataset:* Dataset) → List[*Dataset*]

Collect and return all children for a given Dataset object.

Parameters
dataset (Dataset) – Dataset to query.

Returns
List of children of the given Dataset.

Return type
List

get_parents(*dataset:* Dataset) → List[*Dataset*]

Collect and return all parents for a given Dataset object.

Parameters
dataset (*Dataset*) – Dataset to query.

Returns
List of parents of the given Dataset.

Return type
List

property ds_list: List[*Dataset*]

Returns a flattened list of all datasets within the network.

Returns
description

Return type
type

update_db()

Ensure all datasets within a dependency update their databases together

remove_run(*ident: int | str | dict*) → bool

Chains the Dataset.remove_run function to all Datasets within the dependency.

Parameters
ident (*Union[int, str, dict]*) – Runner identifier.

Returns
True if all runners were removed successfully, False otherwise.

Return type
bool

clear_runs() → None

Remove all Runners from the Datasets.

clear_results(*wipe: bool*) → None

Ensure all datasets within a dependency clear their results together

Parameters
wipe (*bool*) – Additionally delete the local resultfiles if True

wipe_local(*files_only: bool = False*) → None

Clears out the local directory.

Parameters

- **files_only** (*bool, optional*) – Attepts to only delete files.
- **False** (*Defaults to*)
- **directory.** (*deleting the*)

wipe_remote(*files_only: bool = False*) → None

Clears out the remote directory.

Parameters

- **files_only** (*bool, optional*) – Attepts to only delete files.
- **False** (*Defaults to*)
- **directory.** (*deleting the*)

hard_reset(*files_only: bool = False*) → None

Hard resets all datasets. This will wipe local and remote directories.

Parameters

- **files_only** (*bool, optional*) – Attepts to only delete files.
- **False** (*Defaults to*)
- **directory.** (*deleting the*)

append_run(*caller*, *chain_run_args*, *run_args*, *force*, *lazy*, **args*, ***kwargs*)

Appends runs with the same args to all datasets

Parameters

- **lazy** (*bool*)
- **caller** – (Dataset): The dataset which defers to the dependency
- **chain_run_args** (*bool*) – for dependency runs, will not propagate run_args to other datasets in the chain if False (defaults True)
- **run_args** (*dict*) – runner arguments
- **force** (*bool*) – force append if True
- **lazy** – do not update the database after this append (ensure you call `update_db()` after appends are complete, or use the `lazy_append()` contex)
- ***args** – append_run args
- ****kwargs** – append_run keyword args

Returns

None

finish_append() → None

Propagates the completion of runner append to all datasets in the chain.

static get_runner_remote_filepath(*runner*, *workdir: str*, *filetype: str*) → str

Generates the relative remote path from workdir to a runner file The file is specified by filetype

run(*dry_run: bool = False*, *extra: str = None*, *force_ignores_success: bool = False*, *verbose: None | int | bool | Verbosity = None*, ***run_args*)

Handles the Dataset Run for a dependency situation.

Parameters

- **dry_run** (*bool*) – create files, but do not run
- **extra** – str extra text to add to runner jobscripts
- **force_ignores_success** (*bool*) – If True, *force* will submit only failed runners
- **verbose** – Sets local verbose level

update_runners()

Manifest only needs to be collected once, then all the runners can be updated by that call

check_failure()

Raises a RuntimeError if an error is detected in any of the runners

Relies on the runner.is_failed property

remotemanager.dataset.files_mixin module

class remotemanager.dataset.files_mixin.**ExtraFilesMixin**

Bases: object

Provides extra files related functionality to Runner and Dataset

Requires either: - implementation of _extra_files dict - implementation of extra_files_send/recv properties

Also requires a run_path property

run_path = NotImplemented

local_dir = NotImplemented

property extra_files: dict

Returns the extra files set for this runner

property extra_files_send: list

Methods for Dataset, should be overridden in Runner

property extra_files_recv: list

Methods for Dataset, should be overridden in Runner

remotemanager.dataset.function_generation module

remotemanager.dataset.function_generation.**create_run_function**(*submitter: str*, *manifest_filename: str*, *script_run: bool = False*, *add_docstring: bool = True*) → str

Generates a submission function for submitter

script_run should be used if the target function (or script) does not log its output to the manifest (as is the case with None as the function)

$1 is the runner short_uuid $2 is the path to the jobscript $3 is the path to the error file $4 is the path to the result file

Parameters

- **submitter** – submitter to generate for
- **manifest_filename** – path to manifest file
- **script_run** – handle completed and failed status updates in the function
- **add_docstring** – adds docstring to function if True

remotemanager.dataset.lazy_append module

class remotemanager.dataset.lazy_append.**LazyAppend**(*parent*)

Bases: object

Object which enables a lazy_append context manager

append_run(**args*, ***kwargs*)

remotemanager.dataset.repo module

class remotemanager.dataset.repo.**Manifest**(*instance_uuid*, *path_override: str = None*, *name_override: str = None*)

Bases: object

now() → str

to_timestamp(*timestring: str*) → int

property path

get_data(*path: str = None*) → list

write(*string: str*)

get(*uuid: str*) → list

parse_log(*path: str = None*, *string: str = None*, *cutoff: int = None*) → dict

Convert the log into a dict of {uuid: [time, log]}

Parameters

- **path** – Override filepath
- **string** – Override file read with string input
- **cutoff** – Only consider events after this timestamp

last_time(*state: str*) → dict

Takes parsed log, returning the last time state appeared for each uuid

remotemanager.dataset.runner module

Main Runner object. This object handles data I/O and metadata for a single run instance

remotemanager.dataset.runner.**format_time**(*t: ~.datetime.time*) → str

Format the datetime object into a dict key

Parameters
t (*datetime.time*) – time object to be formatted to string

Returns
formatted time

Return type
(str)

class remotemanager.dataset.runner.**Runner**(*arguments: dict*, *dbfile: str*, *parent*, *self_id: str*, *extra_files_send: list | str | None = None*, *extra_files_recv: list | str | None = None*, *verbose: None | int | bool | Verbosity = None*, *extra: str = None*, ***run_args*)

Bases: *SendableMixin*, *ExtraFilesMixin*

The Runner class stores any info pertaining to this specific run. E.g. Arguments, result, run status, files, etc.

Warning

Interacting with this object directly could cause unstable behaviour. It is best to allow Dataset to handle the runners. If you require a single run, you should create a Dataset and append just that one run.

property verbose: *Verbosity*

Verbose property

property database: *Database*

Access to the stored database object. Creates a connection if none exist.

Returns
Database

property parent

Returns the parent Dataset object

property identifier: str

Returns a unique identifier for this, also used in the names of Runner created files.

property serialiser

Returns the parent Serialiser object

property uuid: str

The uuid of this runner

property short_uuid: str

A short uuid for filenames

property id: str

Returns this Runner's current ID

property name: str

Returns this Runner's name

property runfile: *TrackedFile*

Filename of the python runfile

property jobscript: *TrackedFile*

Filename of the run script

property resultfile: *TrackedFile*

Result file name

property errorfile: *TrackedFile*

File tracker for error dumpfile

property local_dir: str

Local staging directory

property remote_dir: str

Target directory on the remote for transports

property run_path: str | None

Intended running directory. If not set, uses remote_dir

> **Note**
>
> If both remote_dir and run_dir are set, the files will be transferred to remote_dir, and then executed within run_dir

property run_dir: str | None

Intended running directory. If not set, uses remote_dir

> **Note**
>
> If both remote_dir and run_dir are set, the files will be transferred to remote_dir, and then executed within run_dir

property derived_run_args: dict

Returns the base run args.

Returns
_run_args

set_run_arg(*key: str*, *val*) → None

Set a single run arg *key* to *val*

Parameters

- **key** – name to set
- **val** – value to set to

Returns
None

set_run_args(*keys: list*, *vals: list*) → None

Set a list of *keys* to `vals

> **Note**
>
> List lengths must be the same

Parameters

- **keys** – list of keys to set
- **vals** – list of vals to set to

update_run_args(*d: dict*) → None

Update current global run args with a dictionary *d*

Parameters
d – dict of new args

property args: dict

Arguments for the function

property extra_files_send: list

Returns the list of extra files to be sent

property extra_files_recv: list

Returns the list of extra files to be retrieved

property history: dict

Sorted state history of this runner

property status_list: list

Returns a list of status updates

insert_history(*t: datetime | int | float*, *newstate: str*, *force: bool = False*, *utc: bool = False*) → bool

Insert a state into this runner's history

Parameters

- **t** (*datetime.time*) – time this state change occurred

- **newstate** (*str*) – status to update
- **force** (*bool*) – skips checks if True
- **utc** – indicates that the set time is UTC

property state: *RunnerState*

Returns the most recent runner state

set_state(*newstate: str*, *state_time: datetime | int | float | None = None*, *force: bool = False*, *check_state: bool = True*, *utc: bool = False*) → None

Update the state and store within the runner history

Added in version 0.9.3: now checks the current state before setting, won't duplicate states unless force=True

Parameters

- **newstate** – state to set to
- **state_time** – set the state at this time, rather than now
- **force** – skip currentstate checking if True
- **check_state** – raises a ValueError if True and newtate is not a valid state
- **utc** – indicates that the set time is UTC

property last_submitted_utc: int

Return the timestamp of the last submission time. If no submission has been made yet, returns -1.

property last_submitted: int

Return the timestamp of the last submission time with a timezone offset

generate_runline(*submitter: str*, *remote_dir: str | None = None*, *child: bool = False*) → str

Generates a runline for this runner

Parameters

- **submitter** – submitter for this run
- **remote_dir** – override the remote directory
- **child** – Used by Dependencies, if this runner is a child, we should skip any path modifications

stage(*extra_files_send: list = None*, *extra_files_recv: list = None*, *repo: str = None*, *extra: str = None*, *summary_only: bool = False*, *parent_check: str = ''*, *child_submit: list = None*, *force_ignores_success: bool = False*, *verbose: None | int | bool | Verbosity = None*, ***run_args*) → bool

Prepare this runner for a run by creating files in the local dir

Parameters

- **extra_files_send** – list of extra files to send
- **extra_files_recv** – list of extra files to receive
- **repo** (*str*) – override the repo target
- **extra** (*str*) – extra lines to append to jobscript. This goes _last_.
- **summary_only** (*bool*) – INTERNAL, used during a lazy append to skip printing. (Calls assess_run with quiet enabled)
- **parent_check** (*str*) – INTERNAL, extra string to check that the parent result exists
- **child_submit** (*list*) – INTERNAL, list of extra lines to submit children
- **force_ignores_success** (*bool*) – If True, *force* takes priority over *is_success* check

- **verbose** – local verbosity
- **run_args** – temporary run args for this run

Returns
True if runner is ready

Return type
bool

generate_jobscript(*parent_check: str*, *child_submit: list*, *extra: str*) → None

Generates and writes the jobscript for this runner.

Parameters

- **parent_check** (*str*) – Internal argument, the bash code to check for the parent result
- **child_submit** (*list*) – Internal argument, the bash code to submit the child
- **extra** (*str*) – Any extra content for the jobscript

Returns
None

stage_python(*repo: str*) → None

Stage a "python" based run. Generates the python runfile.

Parameters
repo (*str*) – Remote relative path to the respository

stage_script(*script:* Script)

Stage a "script" based run. Generates the python runfile.

Parameters
script (Script) – Script entity to parameterise.

stage_none(*extra: str*) → None

Fallback run mode when python or script are not used.

Parameters
extra (*str*) – Extra args

assess_run(*run_args: dict*, *force_ignores_success: bool = False*, *verbose: None | int | bool |* Verbosity *= None*) → bool

Check whether this runner should be running.

If *force* is True we always run

If *skip* is True, we have to check if a run is ongoing, or a result exists

Parameters

- **quiet** – Do not print status if True
- **run_args** – Temporary args specific to this run instance
- **force_ignores_success** (*bool*) – If True, *force* takes priority over *is_success* check
- **verbose** – local verbosity

Returns
True if runner has the green light

Return type
bool

property is_finished: bool | None

Returns True if this runner has finished

(None if the runner has not yet been submitted)

property is_success: bool

Returns True if this runner is considered to have succeeded

property is_failed: bool | None

Returns True if this runner is considered to have failed

(None if incomplete)

read_local_files() → None

Reads all local files attached to this Runner.

Fills out the resulting attributes (result, error, state, etc.)

Returns
None

verify_local_files() → bool

Check the existence of local files on disk

Returns
True if everything is okay

Return type
(bool)

property result

Returns the result attribute, if available

property error

Error (If one exists)

property full_error: str | None

Reads the error file, returning the full error

Returns
str

clear_result(*wipe: bool = True*) → None

Clear the results properties and set the state to "reset", which blocks some functions until the runner is rerun

Parameters
wipe – Additionally deletes the local files if True. Default True

Returns
None

run(**args*, ***kwargs*) → None

Run a single runner. See Dataset.run() for args.

This function is inefficient and should not be used in a general workflow

class remotemanager.dataset.runner.**RunnerFailedError**(*message: str*)

Bases: object

Temporary "exception" to be passed in lieu of a missing result due to a failure.

Parameters
message – error message

remotemanager.dataset.runnerstates module

class remotemanager.dataset.runnerstates.**RunnerState**(*state: str = None*)

Bases: *SendableMixin*

State tracker for a runner

property finished

Returns True if this state is considered "finished"

property state

property success

property failed

property value

remotemanager.dataset.summary_instance module

Small container class for tracking runner insertion

class remotemanager.dataset.summary_instance.**SummaryInstance**(*runner_id: str*, *mode: str*, *quiet: bool*)

Bases: *SendableMixin*

Tracks a Runner insert instance

Parameters

- **runner_id** – uuid of runner
- **mode** – append mode: force, skip, etc.
- **quiet** – True if silent append

runner_id

mode

quiet

remotemanager.decorators package

Submodules

remotemanager.decorators.magic module

class remotemanager.decorators.magic.**RCell**(***kwargs: Any*)

Bases: Magics

Magic function that allows running an ipython cell on a remote machine with minimal lines of code.

sanzu(*line: str*, *cell: str*, *local_ns: dict*) → None

Execute a decorators cell using an implicit remote Dataset

Parameters

- **line** – magic line, includes arguments for cell and Dataset
- **cell** – Cell contents
- **local_ns** – dict containing current notebook runtime attributes

parse_line(*cell: list*, *fargs: list*, *pull: list*)

Generate a callable function from the remaining cell content

Parameters

- **pull** – (list): list of object names to add to return instead of final line
- **cell** – (list): cell content
- **fargs** – (list): function arguments

Returns
formatted function string

Return type
(str)

extract_in_scope(*line: str*, *local_ns: dict*)

This will convert a line to a dictionary of arguments.

It is separated out because we are going to have to selectively populate the local scope with *local_ns*.

parse_tree(*fstr*, *add_return: bool = True*)

This routine adds a return to the end of our generated function.

magics = {'cell': {'sanzu': 'sanzu'}, 'line': {}}

registered = True

remotemanager.decorators.remotefunction module

class remotemanager.decorators.remotefunction.**RemoteFunction**(*function*)

Bases: object

Decorator class to store a function within the "cached functions" property.

Wrap a function with *@RemoteFunction* to store:

```
>>> @RemoteFunction
>>> def func(val):
>>>     return val
```

This function will then be made available in all runs:

```
>>> def main():
>>>     val = ...
>>>     return func(val)
>>>
>>> ds = Dataset(main)
>>> ...
```

Added in version 0.3.6.

property function

remotemanager.decorators.sanzufunction module

class remotemanager.decorators.sanzufunction.**SanzuWrapper**(*function*, **args*, ***kwargs*)

Bases: object

Decorator class to allow you to

Wrap a function with *@SanzuFunction* to store:

```
>>> from remotemanager import URL
>>> url = URL(...)
>>> @SanzuFunction(url=url)
>>> def func(val):
>>>     return val
>>> func(val=3) # creates a Dataset and runs this function on `url`
```

Call this function will transparently create a Dataset, execute the function remotely (synchronously), and return the result.

The function should be called with explicitly named keyword arguments, but logic exists to attempt to match args to kwargs

property dataset: *Dataset*

Return the current Dataset used for this function

Returns
associated Dataset

Return type
(*Dataset*)

remotemanager.decorators.sanzufunction.**SanzuFunction**(**args*, ***kwargs*)

Actual decorator wrapper for SanzuFunction

In order to make a decorator callable, Python seems to require an actual function call that returns the class.

Parameters

- ***args** – args to pass through to the Dataset
- ****kwargs** – kwargs to pass through to the Dataset

Returns
decorator

remotemanager.logging_utils package

Submodules

remotemanager.logging_utils.decorate_verbose module

Mixin class to provide underlying verbose properties

remotemanager.logging_utils.decorate_verbose.**make_verbose**(*cls*)

Adds the correct verbose properties to the class cls

Parameters
cls – class to treat

Returns
cls, modified

remotemanager.logging_utils.log module

class remotemanager.logging_utils.log.**Handler**

Bases: object

Handler class for the global logging. Allows for limited user control

refresh()

deletes and re-creates the log

property write_mode

returns the current write-mode of the logger

property overwrite

property empty

property level

Return the string format of the current logging level

property path

Attribute determining the current log path

debug(*msg*, **args*, ***kwargs*)

Direct passthrough for *debug* logging method

info(*msg*, **args*, ***kwargs*)

Direct passthrough for *info* logging method

warning(*msg*, **args*, ***kwargs*)

Direct passthrough for *warning* logging method

error(*msg*, **args*, ***kwargs*)

Direct passthrough for *error* logging method

critical(*msg*, **args*, ***kwargs*)

Direct passthrough for *critical* logging method

remotemanager.logging_utils.utils module

remotemanager.logging_utils.utils.**format_iterable**(*iterable*, *exclude=None*, *print_types: bool = False*, *indent: int = 1*) → str

Recursive formatting of an iterable, producing log-safe string

Parameters

- **iterable** (*list, tuple, set, dict*) – iterable to sort
- **exclude** (*list, optional*) – keys to exclude
- **indent** (*int, optional*) – indent level for recursion

Returns

newline separated string

Return type

(str)

remotemanager.logging_utils.verbosity module

Verbosity enables deeper verbose levels than True and False

class remotemanager.logging_utils.verbosity.**Verbosity**(*level: None | int | bool | Verbosity = None*)

Bases: *SendableMixin*

Class to store verbosity information

Initialise with Verbosity(level), where level is the integer level

Printing can be requested with Verbose.print(msg, level)

If the verbose level is set above *level*, the message will be printed

Parameters
level (*int, bool, Verbosity*) – level above which to print

level

property value: int
Alias for self.level

print(*message: str*, *level: int*, *end: str = '\n'*)
Request that a message be printed. Compares against the set verbosity level before printing.

Parameters

- **message** (*str*) – message to print
- **level** (*int*) – If this number is higher priority than (lower numeric value) (or equal to) the set limit, print
- **end** (*str*) – print(..., end= ...) hook

remotemanager.script package

Submodules

remotemanager.script.script module

This module holds the Script class, which handles the generation and parameterization of scripts.

class remotemanager.script.script.**EscapeStub**(*content: Any*)

Bases: object

Stub class for avoiding regex's internal escape sequence handling

If the *repl* argument of *re.sub* is a callable, escape sequences will not be processed, allowing us to handle them at a later stage.

This is important for allowing templates to escape the *:* character

content: str

class remotemanager.script.script.**Script**(*template: str*, *empty_treatment: str | None = None*, ***init_args*)

Bases: *SendableMixin*

Class for a generic, parameterisable script.

Parameters
template (*str*) – Base script to use. Accepts #parameters#

property uuid: str

return the uuid of this script

property short_uuid: str

return the shortened uuid of this script

property template: str

Returns the template

property subs: list[str]

Returns a list of all substitution names

property sub_objects: list[*Substitution*]

Returns a list of all substitution objects

property args: list

Alias for self.subs

property arguments

Alias for self.subs

property required: list[str]

Returns a list of all required values

Returns
a list of all required values

Return type
list[str]

property missing: list[str]

Returns a list of all missing required values

Returns
a list of all missing required values

Return type
list[str]

property valid: bool

Returns True if the script is currently "valid"

Validation is defined as having no missing required values and all required values being present.

..note::
Note that this property only considers values that have been flagged as required in the template. This essentially makes it up to the user to enable this functionality in their templates.

Returns
True if the script is currently "valid"

Return type
bool

property empty_treatment: str | None

Returns the currently set global behaviour for empty treatment.

script(*empty_treatment: str | None = None*, ***run_args*) → str

Generate the script

Parameters
empty_treatment (*str, None*) – Overrides any local setting of `empty_treatment` if not None

Returns
the formatted script

Return type
str

pack(*values: bool = True*, *file: str | None = None*) → str

Store the Script

Parameters

- **values** (*bool*) – includes any set values if True
- **file** (*str*) – path to save to, returns the content if None

Returns
File path if file is not None, else the storage content

Return type
str

classmethod unpack(*input: str*)

Re-create an object from a packaged payload coming from `obj.pack`

> **Note**
>
> use this function to unpack from a payload _outside_ an object
>
> ```
> newobj = MyObject.unpack(payload)
> ```
>
> Where `MyObject` is a subclass of SendableMixin, and `payload` is a dict-type coming from `MyObject.pack()`

Parameters

- **data** (*dict*) – __dict__ payload from the object that was packaged
- **file** (*str*) – filepath to unpack from, if data is not given
- **limit** (*bool*) – set False to allow outside classes to be unserialised

Returns
re-created object

remotemanager.serialisation package

Submodules

remotemanager.serialisation.serial module

class remotemanager.serialisation.serial.**serial**

Bases: *SendableMixin*

Baseclass for holding serialisation methods. Subclass this class when implementing new serialisation methods

dump(*obj*, *file: str*) → None

Dump object *obj* to file *file*

Base behaviour tries to write the output of `self.dumps` to a file. Overwrite for custom behaviour

Parameters

- **obj** – object to be dumped

- **file** (*str*) – filepath to dump to

Returns
None

load(*file: str*)

Load previously dumped data from file `file`

Base behaviour tries to load file via `self.dumps` Overwrite for custom behaviour

Parameters
file (*str*) – filepath to load

Returns
Stored object

dumps(*obj*)

loads(*string*)

static wrap_to_list(*obj*)

Preserves the python tying of a set or tuple by inserting an identifier at the start.

If passed with a list starting with this identifier, it will unwrap the typing, restoring the type.

property extension: str

Returns (str): intended file extension

property importstring: str

Returns (str): Module name to import. See subclasses for examples

property callstring: str

Returns (str): Intended string for calling this module's dump. See subclasses for examples

property bytes: bool

Set to True if serialiser requires open(..., 'wb')

property write_mode

Mode for writing to dumped files.

property read_mode

Mode for reading dumped files.

property loadstring: str

property dumpstring: str

property loadfunc_name

property dumpfunc_name

dumpfunc() → str

loadfunc() → str

remotemanager.serialisation.serialdill module

class remotemanager.serialisation.serialdill.**serialdill**

Bases: *serial*

subclass of serial, implementing dill methods

dumps(*obj*)

loads(*string*)

property callstring

Returns (str): Intended string for calling this module's dump. See subclasses for examples

property bytes

Set to True if serialiser requires open(..., 'wb')

remotemanager.serialisation.serialjson module

class remotemanager.serialisation.serialjson.**serialjson**

Bases: *serial*

subclass of serial, implementing json methods

dumps(*obj*)

loads(*string*)

property callstring

Returns (str): Intended string for calling this module's dump. See subclasses for examples

remotemanager.serialisation.serialjsonpickle module

class remotemanager.serialisation.serialjsonpickle.**serialjsonpickle**

Bases: *serial*

subclass of serial, implementing jsonpickle methods

dumps(*obj*)

loads(*string*)

dump(*obj*, *file*)

Dump object *obj* to file *file*

Base behaviour tries to write the output of `self.dumps` to a file. Overwrite for custom behaviour

Parameters

- **obj** – object to be dumped
- **file** (*str*) – filepath to dump to

Returns
None

load(*file*)

Load previously dumped data from file `file`

Base behaviour tries to load file via `self.dumps` Overwrite for custom behaviour

Parameters
file (*str*) – filepath to load

Returns
Stored object

dumpfunc() → str

loadfunc() → str

property callstring: str
Returns (str): Intended string for calling this module's dump. See subclasses for examples

property dumpstring: str

property loadstring: str

remotemanager.serialisation.serialyaml module

class remotemanager.serialisation.serialyaml.**serialyaml**
Bases: *serial*

subclass of serial, implementing yaml methods

dumps(*obj*)

loads(*string*)

property callstring
Returns (str): Intended string for calling this module's dump. See subclasses for examples

property loadstring: str

dumpfunc() → str

remotemanager.storage package

Submodules

remotemanager.storage.database module

class remotemanager.storage.database.**Database**(*file*)
Bases: object

Database file handler for use in the Dataset.

> **Warning**
> Interacting with this object directly could cause unstable behaviour. It is best to allow Dataset to handle the Database

Parameters
file (*str*) – filename to write to

read()
Read the database file

write()
Write the database to file

update(*payload*)

Update the database with payload (dict)

Parameters
payload (*dict*) – Dictionary to recursively update with. Usually called with the output of object.pack()

property stored_uuid: str | None

property path

Path to current database file

property tree: list

Returns a list of path-like strings for the stored database dict

Returns (list):
List of path-like strings

climb(*data: dict*, *branch: list = None*) → list

"climb" a dictionary tree, returning a list of path-like strings for each element

> **Note**
>
> This method is intended for use within the *tree* property, and could cause unintended behaviour if called directly, though will allow you to produce a *tree* like list for any dictionary. Use with caution.

Parameters

- **data** (*dict*) – dictionary to treat
- **branch** (*list*) – current branch, used for recursion

Returns (list):
list of path-like strings

find(*key: str*, *greedy: bool = False*) → dict

Find the first instance of key within the database tree

Parameters

- **key** – key (uuid) to look for
- **greedy** – returns the first value found if True. Faster, but does not respect the tree "layers"

Returns
database tree below key

backup(*file=None*) → str

Back up the database file to *file*, defaults to *{self.path}.bk*

Parameters
file – backup file target location

Returns
backup file path

remotemanager.storage.database.**chain_get**(*d: dict*, *keys: list*)

Get item from a nested dict using a list of keys

Parameters

- **d** (*dict*) – nested dict to query
- **keys** (*list*) – list of keys to use

Returns
item from nested dict *d* at [list, of, keys]

remotemanager.storage.database.**check_version_barriers**(*old: Version*, *new: Version*, *barriers: dict*) → bool

Takes two versions, and a dict of {old: new} that contain breaking changes

Returns True if the two versions are compatible, False otherwise

Parameters

- **old** – old version
- **new** – new version
- **barriers** – dictionary of incompatible {old: new} parings.

  Uses the syntax of Version.match

remotemanager.storage.function module

remotemanager.storage.function.**line_is_end_of_function**(*line: str*) → bool

Returns True if the line is the end of a function definition

class remotemanager.storage.function.**Function**(*func: [typing.Callable, <class 'str'>]*, *force_self: bool = False*)

Bases: *SendableMixin*

Serialise function to an executable python file for transfer

Parameters
func – python function for serialisation

static get_raw_signature(*source*)

Strips the signature as it is typed. inspect.signature does some formatting which can cause replacement to break in some conditions

Parameters
source (*str*) – raw source

Returns
raw signature as typed

Return type
(str)

static prepare_signature(*sig*, *is_self: bool = False*) → str

Inserts *args and **kwargs into any signature that does not already have it

Parameters

- **sig** – inspect.signature(func)
- **is_self** – inspect.signature ignores *self*, yet we want to preserve it. Adds the argument if True

Returns
formatted sig

Return type
(str)

replace_signature(*source: str*, *rawsig: str*, *signature: str*) → str

Performs replacement of the original "raw" signature with one with inserted *args, **kwargs

property name: str

Function name

property raw_source: str

Function source

property source: str

Function source

Returns
source code

Return type
(str)

property return_annotation: str | None

property signature: str

property args: list

returns a list of arguments, without annotations or defaults

property uuid: str

Function uuid (64 characters)

property object

Recreates the function object by writing out the source, and importing.

Returns
the originally passed function

Return type
Callable

dump_to_string(*args*) → str

Dump this function to a serialised string, ready to be written to a python file

Parameters
args (*dict*) – arguments to be used for this dumped run

Returns
serialised file

Return type
(str)

remotemanager.storage.sendablemixin module

This module handles the base functionality for serialising and unserialising objects.

The SendableMixin class provides the necessary methods to convert objects to YAML format and vice versa.

class remotemanager.storage.sendablemixin.**SendableMixin**

Bases: object

Mixin class to create "sendable" object. Provides methods for conversion to yaml format

Create a sendable object by subclassing SendableMixin at class creation

```
>>> class MyObject(SendableMixin):
>>>     ...
```

Instances of this object will now have the required methods to be converted to and from dict format

```
>>> new = MyObject()
>>> payload = new.pack()  # store the object in a dict object
>>> recreated = MyObject.unpack(payload)  # create an instance from dict
```

serialiser = NotImplemented

pack(*uuid: str = None*, *file: str = None*) → dict | None

"packs" the object into a dict-format, ready for yaml-dump

Parameters

- **uuid** (*str*) – Optional uuid string to package this data inside. Adds a toplevel *uuid* to the payload dict: >>> p = {...}

  Becomes: >>> p = {uuid: {...}}

- **file** (*str*) – Directly package to file with *yaml.dump*

Returns
object payload

Return type
(dict)

classmethod unpack(*data: dict = None*, *file: str = None*, ***kwargs*)

Re-create an object from a packaged payload coming from `obj.pack`

Note

use this function to unpack from a payload _outside_ an object

```
newobj = MyObject.unpack(payload)
```

Where `MyObject` is a subclass of SendableMixin, and `payload` is a dict-type coming from `MyObject.pack()`

Parameters

- **data** (*dict*) – __dict__ payload from the object that was packaged
- **file** (*str*) – filepath to unpack from, if data is not given
- **limit** (*bool*) – set False to allow outside classes to be unserialised

Returns
re-created object

inject_payload(*payload: dict*)

inject payload into the __dict__, effectively re-creating the object

Note

use this function to unpack _within_ an object

```
>>> class MyObject(SendableMixin):
>>>     def __init__(self, ...):
>>>         ...
>>>         self.inject_payload(payload)
```

Parameters
payload (*dict*) – __dict__ payload from the object that was packaged

serialise(*obj*)

Recurse over any iterable objects, or call the pack() method of any *SendableMixin* objects, for serialisation

Parameters
obj – object to be packaged

Returns (yaml-serialisable object):
yaml-friendly object

unserialise(*obj*)

Undo a serialised packaging, by importing the called object and calling its unpack() method

Parameters
obj – payload to be unpacked

Returns
object before packaging

is_missing(*objname: str*) → bool

Determine if object with name objname is missing or uninitialised

Parameters
objname (*str*) – name of the object to look for

Returns
object presence

Return type
(bool)

remotemanager.storage.sendablemixin.**get_class_storage**(*obj*) → Dict[str, str]

Breaks down object into its module and classname.

Parameters
obj – Python object to be broken down

Returns (dict):
module and classname dict

remotemanager.storage.sendablemixin.**get_mro_classnames**(*obj*) → List[str]

Retrieves a list of class names from the Method Resolution Order (MRO) of an object.

Parameters
obj – Python object whose MRO is to be retrieved.

Returns
A list containing the class names in the order they appear in the MRO.

Return type
List[str]

remotemanager.storage.sendablemixin.**basic_available**(*obj*) → bool

attempt a basic JSON serialisation, which will fail fast if we can't dump

remotemanager.storage.trackedfile module

class remotemanager.storage.trackedfile.**TrackedFile**(*local_path: str*, *remote_path: str*, *file: str*, *binary: bool = False*)

Bases: *SendableMixin*

property binary: bool

Access to private _binary attribute.

property name: str

Returns the filename

property importstr: str

Returns the filename without extension

Suitable for python imports

property remote: str

Returns the full remote path

property local: str

Returns the full local path

property remote_dir: str

Returns the remote dir

property local_dir: str

Returns the full local dir

relative_remote_path(*other: str*) → str

Return a path relative to *cwd*

Parameters
other – working dir to compare against

Returns
relative path

property content: str | None

Attempts to read the file contents

Returns None if the file cannot be read

write(*content: str | list*, *add_newline: bool = True*) → None

Write *content* to the local copy of the file

Parameters

- **content** – content to write
- **add_newline** – enforces a newline character at the end of the write if True (default True)

append(*content: str | list*, *add_newline: bool = True*) → None

Append *content* to the local copy of the file

Parameters

- **content** – content to append
- **add_newline** – enforces a newline character at the end of the write if True (default True)

confirm_local(*t: int | None = None*) → None

Confirm sighting of the file locally

Parameters
t – Optionally set the time to *t* instead of time.time()

confirm_remote(*t: int | None = None*) → None

Confirm sighting of the file on the remote

Parameters
t – Optionally set the time to *t* instead of time.time()

property exists_local: bool

Returns True if the file exists locally

last_seen(*where: str*) → int

Returns the last_seen_<where>

Where <where> is remote or local

Parameters
where – remote or local

property last_seen_local: int

Returns the time this file was last confirmed seen on the local machine

property last_seen_remote: int

Returns the time this file was last confirmed seen on the remote machine

property local_mtime_utc: int

Returns the mtime of the local file in utc

property size: int

Returns the filesize (needs to be set externally)

-1 if not set

sub(*source*, *target*, *mode: str = 'python'*) → bool

Substitute source for target

Parameters

- **source** – file content to sub
- **target** – intended replacement
- **mode** – function to use, default to Python replace

Returns
True if a substitution was executed, False otherwise

chmod(*mod: int*)

chmod the local file, if it exists

python3 chmod requires an octal input, so convert the base10 input

remotemanager.transport package

Submodules

remotemanager.transport.cp module

Handles file transfer via *cp*

class remotemanager.transport.cp.**cp**(**args*, ***kwargs*)

Bases: *Transport*

cmd(*primary*, *secondary*)

Returns a formatted command for issuing transfers. It is left to the developer to implement this method when adding more transport classes.

The implementation should take two strings as arguments, *primary* and *secondary*:

Parameters

- **primary** (*str*) – The source folder, containing the files for transfer. Input will be semi-formatted already in bash form.

  e.g. directory_name/{file1,file2,file3,...,fileN}

- **secondary** (*str*) – The destination folder for the files

At its most basic:

``` def cmd(self, primary, secondary):

cmd = "command {primary} {secondary}" base = cmd.format(primary=primary,

secondary=secondary)

return base

```

You can, of course, extend upon this. View the included transport methods for ideas on how to do this.

Returns (str):
formatted command for issuing a transfer

remotemanager.transport.rsync module

Handles file transfer via the *rsync* protocol

class remotemanager.transport.rsync.**rsync**(**args*, ***kwargs*)

Bases: *Transport*

Class for *rsync* protocol

Parameters

- **checksum** (*bool*) – Adds checksum arg, which if `True` will add `--checksum` flag to parameters
- **progress** (*bool*) – request streaming of rsync –progress stdout

check_version(*collect_only: bool = False*, *min_version: str = None*) → *Version*

Queries the installed rsync version and checks that it is recent enough

Does nothing if the version could not be detected, so should be considered a "soft" check, rather than a true safety feature.

Parameters

- **collect_only** (*bool*) – Does not perform a check if True. Defaults to False
- **min_version** (*str*) – override the minimum version (used for testing)

Returns
Version

cmd(*primary*, *secondary*)

Returns a formatted command for issuing transfers. It is left to the developer to implement this method when adding more transport classes.

The implementation should take two strings as arguments, *primary* and *secondary*:

Parameters

- **primary** (*str*) – The source folder, containing the files for transfer. Input will be semi-formatted already in bash form.

  e.g. directory_name/{file1,file2,file3,...,fileN}

- **secondary** (*str*) – The destination folder for the files

At its most basic:

``` def cmd(self, primary, secondary):

cmd = "command {primary} {secondary}" base = cmd.format(primary=primary,

secondary=secondary)

return base

```

You can, of course, extend upon this. View the included transport methods for ideas on how to do this.

Returns (str):
formatted command for issuing a transfer

remotemanager.transport.scp module

Handles file transfer via the *scp* protocol

class remotemanager.transport.scp.**scp**(**args*, ***kwargs*)

Bases: *Transport*

Class to handle file transfers using the scp protocol

Parameters
url (URL) – url to extract remote address from

cmd(*primary*, *secondary*)

Returns a formatted command for issuing transfers. It is left to the developer to implement this method when adding more transport classes.

The implementation should take two strings as arguments, *primary* and *secondary*:

Parameters

- **primary** (*str*) – The source folder, containing the files for transfer. Input will be semi-formatted already in bash form.

  e.g. directory_name/{file1,file2,file3,...,fileN}

- **secondary** (*str*) – The destination folder for the files

At its most basic:

``` def cmd(self, primary, secondary):
```

cmd = “command {primary} {secondary}” base = cmd.format(primary=primary,

secondary=secondary)

return base

```

You can, of course, extend upon this. View the included transport methods for ideas on how to do this.

**Returns (str):**
formatted command for issuing a transfer

### remotemanager.transport.transport module

Baseclass for any file transfer

**class** remotemanager.transport.transport.**Transport**(*url=None*, *dir_mode: bool = False*, *flags: str = None*, *verbose: None | int | bool | Verbosity = None*, *\*args*, *\*\*kwargs*)

Bases: *SendableMixin*

Baseclass for file transfer

**Parameters**

- **url** (URL) – url to extract remote address from
- **dir_mode** – compatibility mode for systems that do not accept multiple explicit files per transfer, copies files to a directory then pulls it

**property verbose: *Verbosity***

Verbose property

**property dir_mode: bool**

**queue_for_push**(*files: [<class 'list'>, <class 'str'>, <class 'remotemanager.storage.trackedfile.TrackedFile'>]*, *local: str = None*, *remote: str = None*)

Queue file(s) for sending (pushing)

**Parameters**

- **files** (*list[str], str,* TrackedFile) – list of files (or file) to add to push queue
- **local** (*str*) – local/origin folder for the file(s)
- **remote** (*str*) – remote/destination folder for the file(s)

**Returns**
None

**queue_for_pull**(*files: [<class 'list'>, <class 'str'>, <class 'remotemanager.storage.trackedfile.TrackedFile'>]*, *local: str = None*, *remote: str = None*)

Queue file(s) for retrieving (pulling)

**Parameters**

- **files** (*list[str], str,* TrackedFile) – list of files (or file) to add to pull queue
- **local** (*str*) – local/destination folder for the file(s)
- **remote** (*str*) – remote/origin folder for the file(s)

**Returns**
None
```

add_transfer(*files: [<class 'list'>, <class 'str'>], origin: str | None, target: str | None, mode: str*)

Create a transfer to be executed. The ordering of the origin/target files should be considered as this transport instance being a "tunnel" between wherever it is executed (origin), and the destination (target)

Parameters

- **files** (*list[str], str*) – list of files (or file) to add to pull queue
- **origin** (*str*) – origin folder for the file(s)
- **target** (*str*) – target folder for the file(s)
- **(str** (*mode*) – "push" or "pull"): transfer mode. Chooses where the remote address is placed

Returns
None

property transfers: dict

Return the current transfer dict

Returns (dict):
{paths: files} transfer dict

print_transfers()

Print a formatted version of the current queued transfers

Returns
None

property address

return the remote address

Returns (str):
the remote address

property url: *URL*

set_remote(*url=None*)

set the remote address with a URL object

Returns
None

property flags

cmd(*primary: str*, *secondary: str*) → str

Returns a formatted command for issuing transfers. It is left to the developer to implement this method when adding more transport classes.

The implementation should take two strings as arguments, *primary* and *secondary*:

Parameters

- **primary** (*str*) – The source folder, containing the files for transfer. Input will be semi-formatted already in bash form.

  e.g. directory_name/{file1,file2,file3,...,fileN}

- **secondary** (*str*) – The destination folder for the files

At its most basic:

``` def cmd(self, primary, secondary):

cmd = "command {primary} {secondary}" base = cmd.format(primary=primary,

secondary=secondary)

return base
```

```

You can, of course, extend upon this. View the included transport methods for ideas on how to do this.

**Returns (str):**
formatted command for issuing a transfer

**transfer**(*dry_run: bool = False*, *prepend: bool = True*, *raise_errors: bool = True*, *dir_mode: bool = None*, *verbose: None | int | bool | Verbosity = None*)

Perform the actual transfer

**Parameters**

- **dry_run** (*bool*) – do not perform command, just return the command(s) to be executed
- **prepend** (*bool*) – enable forced cmd prepending
- **raise_errors** (*bool*) – will not raise any stderr if False
- **dir_mode** – compatibility mode for systems that do not accept multiple explicit files per transfer, copies files to a directory then pulls it

**Returns (str, None):**
the dry run string, or None

**wipe_transfers**()

**property cmds**

**static split_pair**(*pair: str*) → list

Convert a "dir>dir" string into list format

**Parameters**
**pair** (*tuple*) – (dir, dir) tuple to be split

**Returns (list):**
[dir, dir]

**static get_remote_dir**(*path*)

## remotemanager.utils package

## Submodules

## remotemanager.utils.flags module

**class** remotemanager.utils.flags.**Flags**(*\*initial_flags*)

Bases: *SendableMixin*

Basic but flexible handler for terminal command flags

Allows for inplace modification:

```
>>> f = Flags('abcdd')
>>> f -= 'd'
>>> f.flags
>>> '-abcd'
```

**Parameters**
**initial_flags** (*str*) – initial base flags to be used and modified if needed
```

ensure_prefix_exists(*prefix*)

Ensures that the prefix exists in the internal storage, creating it if not

parse_string(*string*) → [<class 'str'>, <class 'bool'>]

Takes a string, and strips away any non-alphanumeric chars. Returns True in secondary return if this is a verbose flag

Parameters
string (*str*) – input flags

Returns (str, bool):
filtered input True if this is a verbose flag

property flags

Returns the fully qualified flags as a string

remotemanager.utils.flags.**strip_non_alphanumeric**(*string*)

remove any non-alphanumeric strings from input string

Parameters
string (*str*) – input string

Returns (str):
input string, sans any non-alphanumeric chars

remotemanager.utils.testingbaseclass module

base testing class to reduce code duplication

class remotemanager.utils.testingbaseclass.**BaseTestClass**

Bases: object

datasets = []

files = []

kwarg_list = []

fn_list = []

wrap()

setUp()

tearDown()

Clean up

property ds: *Dataset*

create_dataset(*function*, *recreate: bool = False*, ***dataset_kwargs*) → *Dataset*

Generate a dataset for test usage

create_datasets(*functions: list*, ***dataset_kwargs*) → list

property previous_ds_kwargs: dict

property previous_ds_fn: Callable | None

recreate_previous_dataset(***dataset_kwargs*) → *Dataset*

Attempt to recreate the previously generated dataset

create_random_file(*content: str | None = None*, *directory: str | None = None*) → str

random_string(*len: int = 16*)

static destroy_dataset(*dataset:* Dataset)

run_ds(*interval=0.1*, *timeout=5*, ***kwargs*) → list

remotemanager.utils.testingbaseclass.**try_remove**(*f*)

remotemanager.utils.timing module

Simple module holding a global commodity function for generating the current timestamp

remotemanager.utils.timing.**utcnow**() → int

Return the current UTC timestamp

remotemanager.utils.timing.**local_ts_to_utc**(*ts: int | float*) → int

convert a local timestamp to a utc one

remotemanager.utils.tokenizer module

Module to store the tokenizer class

class remotemanager.utils.tokenizer.**Tokenizer**(*content: str*)

Bases: object

generate_tokens() requires a file-like object to function

This class mimics this behaviour by raising StopIteration once the end of the content is reached.

content

row

tokenize() → list

Runs the tokenization

property tokens: list

Returns all stored tokens

list format is [(type, string), . . .]

property names: list

Returns the derived name list

property numbers: list

Returns derived number list

exchange_name(*a: str*, *b: str*)

Exchanges name a with name b

property source

Regenerate the source from the stored tokens

untokenize is. . . unreliable, since it focuses on the repeatability of the round trip

So we should reconstruct manually

Key tokens: 0: End marker 1: Name 2: Number 3: String 4: Newline 5: Indent 6: Dedent

Returns

reconstructed source

Return type
(str)

remotemanager.utils.uuid module

remotemanager.utils.uuid.**generate_uuid**(*string: str*) → str

Generates a UUID string from an input

Parameters
string – input string

Returns
(str) UUID

remotemanager.utils.version module

class remotemanager.utils.version.**Version**(*ver*)

Bases: object

Lightweight temporary class for version comparison.

Create at least one instance of Version to compare two semantic versions:

```
>>> v = Version('1.5.2')
>>> v < '4.1.0'
>>> True
```

Parameters
ver (*str*) – Semantic version in x.y.z form

property major: int

property minor: int

property patch: int

property version: str

match(*other:* Version | *str*) → bool

CHAPTER

THIRTYFOUR

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

r

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W