Multiscale Genomics Documentation

Release 0.1

Mark McDowall

Nov 05, 2018

Contents:

Virtual Research Environment- Supporting the 3D/4D genomics community with tools to integrate navigation from sequence to 3D/4D chromatin dynamics data.

Full details of the project can be found on the MuG website.

From this site there are links to all of the documentation for each of the repositories within the MuG GitHub. Each set of documents contains details about the installation and usage both for an end user and for developers creating workflows.

If you want to develop tools and workflows that can run within the VRE please checkout the HOWTO section of the site. This lists how to write a workflow, tool, the configuration files and testing that the workflow works. Please also read about how to apply the Apache2.0 license to your code and a document on Coding Standards that you should adher to to ensure the best chance of your workflow being integrated with the VRE as smoothly as possible.

CHAPTER 1

VRE

VRE MuG Virtual Research Environment. MuG VRE is a web application aimed to allow MuG users to access MuG data and explore and exploit it together with its own data via a selection of tools and visualizers. It is written in PHP, HTML and Javascript.

GitHub VRE

CHAPTER 2

Development APIs

mg-dm-api Data Management API. This API tracks files within the VRE and contains meta data about how the file was generated with access to the file geneology.

GitHub mg-dm-api

mg-tool-api Tool API. This API provides the interface between the pyCOMPSs architecture and the tool. It provides a standard way for all tools to be wrapped to allow for a common interface layer.

GitHub mg-tool-api

CHAPTER 3

Workflows

mg-process-fastq Workflows for processing FASTQ data. These workflows can handle ChIP-seq, MNase-Seq, RNA-Seq and Whole Genome BiSulphate Sequencing (WGBS). There are also scripts for generating the inditial set of indexes for given genome assemblies. There are also workflows for processing Hi-C data to generate adjacency matrices and calculate TAD regions

GitHub mg-process-fastq

mg-process-files Workflows for processing results files into an indexed form for use in a RESTful interface.

GitHub mg-process-files

CHAPTER 4

RESTful APIs

mg-rest-service The root RESTful server. This provides links to the main root end points. Each end point is a provides a unique function within the defined URL so that the service as a whole appears seamless to the end user.

GitHub mg-rest-service

mg-rest-dm RESTful interface to the DM API along with end-points to manage the stored files and track the relevant metadata.

GitHub mg-rest-dm

mg-rest-file RESTful interface to the DM API along with end-points to servicing out regions from basic file based data, such as Bed, Wig and TSV files.

GitHub mg-rest-file

mg-rest-adjacency Interface for RESTfully querying adjacency matrices generated by the TADbit workflows developed in mg-process-fastq.

GitHub mg-rest-adjacency

mg-rest-3d Interface for RESTfully querying 3D models generated by the TADbit workflows developed in mg-process-files.

GitHub mg-rest-3d

mg-rest-util Set of common functions that are required by the RESTful interfaces for interacting with the DM API.

GitHub mg-rest-util

4.1 HOWTO

4.1.1 Development Checklist

This document describes the standard work flow to help developers when creating a new tool or pipeline. The purpose is to aid the developer in the most efficient way for integrating a new tool or pipeline and ensure that all steps have

been addressed so that they have a ready to deploy Tool and Pipeline within the MuG VRE.

Note: If you are adding a new tool and pipeline to an already existing repository then you can skip ahead and concentrate on **steps 1 to 6**.

Note: If you are adding just a new pipeline that just integrates already existing tools then you need to look at **steps 3 to 6**.

---

0 - Copy mg-process-test from GitHub

0.0 - Create an empty repository

In GitHub create a blank repository with no README, license or *.gitignore* file. These files will be inherited from the mg-process-test file. For this example it will be called *mg-process-test1.git*.

0.1 - Copy the mg-process-test repository

From GitHub take a copy of the mg-process-test repository:

```
git clone --depth 1 -b master https://github.com/Multiscale-Genomics/mg-process-test

rm -rf mg-process-test/.git
mv mg-process-test mg-process-test1
cd mg-process-test1

git init
git add .
git commit -m 'Initial commit'

git remote add origin https://github.com/<USERNAME>/mg-process-test1.git
git remote -v

git push origin master
```

From here you can then customise the following files to match your new repository:

- *README.md*
- *NOTICE*
- setup.py
- *__init__.py*
- docs/conf.py

The files in *docs* contain boilerplate data that matches the processes and tools already in the repository, so should be updated as you add new pipelines and tools.

1 - Setup Your Python environment

1.1 - pyenv

This is required for managing the version of Python and the installation environment for the Python modules so that they can be installed in the user space.

```
git clone https://github.com/pyenv/pyenv.git ~/.pyenv
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bash_profile
echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bash_profile
echo 'eval "$(pyenv init -)"' >> ~/.bash_profile

# Add the .bash_profile to your .bashrc file
echo 'source ~/.bash_profile"' >> ~/.bashrc

git clone https://github.com/pyenv/pyenv-virtualenv.git ${PYENV_ROOT}/plugins/pyenv-
↪virtualenv

pyenv install 2.7.14
pyenv virtualenv 2.7.14 mg-process-test
```

1.2 - Install Tool API

```
pyenv activate mg-process-test
pip install git+https://github.com/Multiscale-Genomics/mg-tool-api.git
```

2 - Create a Tool

See the HOWTO - Tools for details about writing a tool and HOWTO - Test Your Code about how to write relevant tests

2.1 - Tool Development

Using the *testTool.py* script as a template, create you new tool.

Checklist

1. There is a license at the header of the script
2. Documentation for each function.
3. Code matches the PEP8 standard (by running pylint).
4. Tool has been added to *docs/tool.rst*

3 - Create a Test to run the Tool

3.1 - Test Dataset

Create a small test dataset that can be used when testing the code. This should match the input file type required by the Tool.

When the tool has been run the output for the test datasets should provide a valid result. For example if wrapping a peak caller there should be enough of the genome selected and matching reads that when aligned and the peak caller analyses the alignments it should generate results similar to the original for that region.

Once the datasets have been generated the procedure for how the test sets were created should be documented in a new "NNN.rst" file. This should contain the source of the data, publications, where the files were downloaded from and how the data was handled so that this can be repeated if the datasets need to be regenerated or changed at a later stage. This file should then be linked into the rest of the documentation, this is usually done by linking the file in the table of contents block in the index page.

3.2 - Test Scripts

Create a script that uses pytest to check that the required output files have been generated and are not empty. Other tests can be added here if there are other aspects that should be tested. Examples could include testing if a JSON object has the expected parameters.

Checklist

1. There is a test to run each single tool
2. There is a license header in each test script
3. All functions in the test script are fully documented with details about how to run the test or if other tests need to be run first
4. Test dataset generation has been fully documented and linked to the index.rst file
5. Any scripts developed to create the datasets are stored in *scripts/*. and have matching license headers and documentation
6. All code matches the PEP8 standard (by running pylint).
7. All new tests have been added to TravisCI
8. All tests are passing
9. Ensure that the output of running the tests matches what you would expect

4 - Create a Pipeline

See the HOWTO - Pipelines for details about writing a pipeline and HOWTO - Test Your Code about how to write relevant tests.

4.1 - Pipeline Development

Using the *process_test.py* script as a template, create a pipeline to accept the configuration and input JSON files that describe the parameters and files to get passed into the pipeline. The pipeline should manage the passing of file locations and parameters to each of the tools.

4.2 - Create a Test to run the Pipeline

Create a script that uses pytest to check that the required input files and configuration parameters are accepted by the pipeline and the relevant output files have been generated and are not empty. Other tests can be added to be more comprehensive.

The pipeline is running tools developed as part of part 1, so there should be no need for creating new datasets.

4.3 - Create test config and input JSON files

JSON files need to be created that duplicate what would be the expected input coming from the VRE and saved in the *tests/json/*. directory of the repository. Example files can be found in the HOWTO on Configuration. There are also examples of these files in mg-process-test in the *test/json/*. These files allow a user to run the sample datasets from the command line either on their own computer or on one with (py)COMPSs installed.

Checklist

1. There is a license in the header of all pipelines and tests
2. There is a test to run each pipeline
3. There is documentation for all functions in the pipeline script and test script
4. Update docs/pipelines.rst to include documentation and links to the new pipeline to import all function documentation
5. All code matches the PEP8 standard (by running pylint).
6. All new tests have been added to TravisCI
7. All tests are passing
8. Ensure that the output of running the tests matches what you would expect
9. The script can be run from the command line

5 - VRE JSON Configuration

See the HOWTO - Configuration Files for details about writing a MuG VRE JSON configuration files.

Checklist

1. Ensure that there is a JSON configuration file present in the tool_json for each pipeline.

6 - Installation Documentation

Checklist

1. Make sure that setup.py, setup.cfg and requirements.txt are updated with any new packages required for installation
2. Update docs/install.rst if there is any external software that is required by tool or pipeline along with the required command to install that software

7 - COMPSs testing

Now that you have a functional pipeline and tool it now needs to be tested within a COMPSs environment. Download the latest version of the COMPSs virtual machine from the BSC website.

Checklist

1. Was it possible to install everything based on the installation scripts and documentation?
2. Do all the test scripts pass when they are run?
3. When the test scripts have run do you get the expected results?
4. Can the pipeline be run using the "runcompss" command?

8 - Hook up your repository for continuous integration

Now that you have a fully documented pipeline, with tests it is possible to hook up your GitHub repository with ReadTheDocs.org, Travisci.org and Landscape.io. These services will automatically build you documentation, run the tests and check the compliance of the code with that of PEP8 respectively.

It is possible to login to each service using your GitHub account and link the repository.

Checklist

1. You have your documentation building on ReadTheDocs.org
2. You have your test scripts running on TravisCI and passing
3. Your code is being continually analysed by Landscape.io

9 - Congratulations

You now have a pipeline that could be integrated into the MuG VRE.

4.1.2 HOWTO - Tools

This document provides a tutorial for the creation of a tool that can be used within a pipeline within the MuG VRE. All functions should be wrapped up as a tool, this then allows for the tools to be easily reused by other pipelines and also deployed onto the compute cluster.

The Tool is the core element when it comes to running a program of function within the COMPSs environment. It defines the procedures that need to happen to prepare the data along with the function that is parallelised to run over the chunks of data provided. A function can be either a piece of code that is written in python or an external package that is run with given chunks of data or defined parameters. The results are then returned to the calling function for merging.

All functions contain at least a *run(self)* function which is called by the pipeline. The run function takes the input files (list), defined output files (list) and relevant metadata (dict). Returned by the run function is a list containing a list of the output files as the first object and a list of metadata dict objects as the second element.

Repository Structure

All tools should be placed within the *tools* directory within the package.

Basic Tool

This is a test tool that takes an input file, then counts the number of characters in that file and then prints the result to a second file. The matching code can be found in the GitHub repository mg-process-test. The file is called *testTool.py*.

```
"""
.. License and copyright agreement statement
"""
from __future__ import print_function

from utils import logger

try:
    from pycompss.api.parameter import FILE_IN, FILE_OUT
    from pycompss.api.task import task
    from pycompss.api.api import compss_wait_on
except ImportError:
    logger.warn("[Warning] Cannot import \"pycompss\" API packages.")
    logger.warn("          Using mock decorators.")

    from utils.dummy_pycompss import FILE_IN, FILE_OUT  # pylint: disable=ungrouped-
↪imports
    from utils.dummy_pycompss import task  # pylint: disable=ungrouped-imports
    from utils.dummy_pycompss import compss_wait_on  # pylint: disable=ungrouped-
↪imports

from basic_modules.tool import Tool
from basic_modules.metadata import Metadata

# ------------------------------------------------------------------------------

class testTool(Tool):
    """
    Tool for writing to a file
    """

    def __init__(self, configuration=None):
        """
        Init function
        """
        print("Test writer")
        Tool.__init__(self)

        if configuration is None:
            configuration = {}

        self.configuration.update(configuration)

    @task(returns=bool, file_in_loc=FILE_IN, file_out_loc=FILE_OUT, isModifier=False)
    def test_writer(self, file_loc):
        """
        Count the number of characters in a file and return a file with the count

        Parameters
        ----------
        file_in_loc : str
            Location of the input file
        file_out_loc : str
```

(continues on next page)

(continued from previous page)

```
            Location of an output file

        Returns
        -------
        bool
            Writes to the file, which is returned by pyCOMPSs to the defined location
        """
        try:
            with open(file_loc, "w") as file_handle:
                file_handle.write("This is the test writer")
        except IOError as error:
            logger.fatal("I/O error({0}): {1}".format(error.errno, error.strerror))
            return False

        return True

    def run(self, input_files, input_metadata, output_files):
        """
        The main function to run the test_writer tool

        Parameters
        ----------
        input_files : dict
            List of input files - In this case there are no input files required
        input_metadata: dict
            Matching metadata for each of the files, plus any additional data
        output_files : dict
            List of the output files that are to be generated

        Returns
        -------
        output_files : dict
            List of files with a single entry.
        output_metadata : dict
            List of matching metadata for the returned files
        """

        results = self.test_writer(
            input_files["input_file_location"],
            output_files["output_file_location"]
        )

        results = compss_wait_on(results)

        if results is False:
            logger.fatal("Test Writer: run failed")
            return {}, {}

        output_metadata = {
            "test": Metadata(
                data_type="<data_type>",
                file_type="txt",
                file_path=output_files["test"],
                sources=[input_metadata["input_file_location"].file_path],
                taxon_id=input_metadata["input_file_location"].taxon_id,
                meta_data={
                    "tool": "testTool"
```

(continues on next page)

(continued from previous page)

```
                    }
                )
            }

            return (output_files, output_metadata)
```

This is this simplest case of a Tool that will run a function within the COMPSS environment. The run function takes the input files, if the output files are defined it can use those as the output locations and any relevant metadata. The locations of the output files can also be defined within the run function as sometimes functions can generate a large number of files that are not always easy to define up front if the Tool is being run as part of the VRE or as part of a larger pipeline.

The run function then calls the *test_writer* function. This uses the python decorator syntax to highlight that it is a function that can be run in parallel to pyCOMPSs library. The *task* decorator is used to define the list of files and parameters that need to be passed to the function. It also requires a list of the files a that are to be returned. As such the most common types will be *FILE_IN*, *FILE_OUT*, *FILE_INOUT*.

The *__init__* function is important as it loads the configuration parameters into the class from the VRE. In this case there are no parameters used, but these can be parameters required for the tool that has been wrapped by the code.

Decorators can also be used to define the resources that are required by function. They can be used to define a set of machines that the task should be run on, required CPU capacity or the amount of RAM that is required by the task. Defining these parameters helps the COMPSS infrastructure correctly allocate jobs so that they are able to run as soon as the resources allow and prevent the job failing by being run on a machine that does not have the correct resources.

Further details about COMPSS and pyCOMPSs can be found at the BSC website along with specific tutorials about how to write functions that can utilise the full power of COMPSS.

pyCOMPSs within the Tool

When importing the pyCOMPSs modules it is important to provide access to the dummy_pycompss decorators as well. This will allow scripts to be run on computers where COMPSs has not been installed.

Practical Example

Now that we know the basics it is possible to apply this to writing a tool that can run and perform a real operation within the cluster.

Here is a tool that uses BWA to index a genome sequence file that has been saved in FASTA format.

The run function takes the input FASTA file, from this it generates a list of the locations of the output files. The input file and output files are passed to the bwa_indexer function. The files do not need to be listed in the return call so True is fine. COMPSS handles the passing back of the files to the run function. The run function then returns the output files to the pipeline or the VRE.

```
from __future__ import print_function

import os
import shlex
import shutil
import subprocess
import sys
import tarfile

from utils import logger
```

(continues on next page)

(continued from previous page)

```

try:
    if hasattr(sys, '_run_from_cmdl') is True:
        raise ImportError
    from pycompss.api.parameter import FILE_IN, FILE_OUT
    from pycompss.api.task import task
    from pycompss.api.api import compss_wait_on
except ImportError:
    logger.warn("[Warning] Cannot import \"pycompss\" API packages.")
    logger.warn("          Using mock decorators.")

 from utils.dummy_pycompss import FILE_IN, FILE_OUT # pylint: disable=ungrouped-
↪imports
 from utils.dummy_pycompss import task # pylint: disable=ungrouped-imports
 from utils.dummy_pycompss import compss_wait_on # pylint: disable=ungrouped-imports

from basic_modules.tool import Tool
from basic_modules.metadata import Metadata

# ------------------------------------------------------------------------------

class bwaIndexerTool(Tool):
    """
    Tool for running indexers over a genome FASTA file
    """

    def __init__(self, configuration=None):
        """
        Init function
        """
        print("BWA Indexer")
        Tool.__init__(self)

        if configuration is None:
            configuration = {}

        self.configuration.update(configuration)

    def bwa_index_genome(self, genome_file):
        """
        Create an index of the genome FASTA file with BWA. These are saved
        alongside the assembly file. If the index has already been generated
        then the locations of the files are returned

        Parameters
        ----------
        genome_file : str
            Location of the assembly file in the file system

        Returns
        -------
        amb_file : str
            Location of the amb file
        ann_file : str
            Location of the ann file
        bwt_file : str
            Location of the bwt file
```

(continues on next page)

(continued from previous page)

```
        pac_file : str
            Location of the pac file
        sa_file : str
            Location of the sa file

        """
        command_line = 'bwa index ' + genome_file

        amb_name = genome_file + '.amb'
        ann_name = genome_file + '.ann'
        bwt_name = genome_file + '.bwt'
        pac_name = genome_file + '.pac'
        sa_name = genome_file + '.sa'

        if os.path.isfile(bwt_name) is False:
            args = shlex.split(command_line)
            process = subprocess.Popen(args)
            process.wait()

        return (amb_name, ann_name, bwt_name, pac_name, sa_name)

    @task(file_loc=FILE_IN, idx_out=FILE_OUT)
    def bwa_indexer(self, file_loc, idx_out): # pylint: disable=unused-argument
        """
        BWA Indexer

        Parameters
        ----------
        file_loc : str
            Location of the genome assebly FASTA file
        idx_out : str
            Location of the output index file

        Returns
        -------
        bool
        """
        amb_loc, ann_loc, bwt_loc, pac_loc, sa_loc = self.bwa_index_genome(file_loc)

        # tar.gz the index
        print("BS - idx_out", idx_out, idx_out.replace('.tar.gz', ''))
        idx_out_pregz = idx_out.replace('.tar.gz', '.tar')

        index_dir = idx_out.replace('.tar.gz', '')
        os.mkdir(index_dir)

        idx_split = index_dir.split("/")

        shutil.move(amb_loc, index_dir)
        shutil.move(ann_loc, index_dir)
        shutil.move(bwt_loc, index_dir)
        shutil.move(pac_loc, index_dir)
        shutil.move(sa_loc, index_dir)

        index_folder = idx_split[-1]

        tar = tarfile.open(idx_out_pregz, "w")
```

(continues on next page)

(continued from previous page)

```
        tar.add(index_dir, arcname=index_folder)
        tar.close()

        command_line = 'pigz ' + idx_out_pregz
        args = shlex.split(command_line)
        process = subprocess.Popen(args)
        process.wait()

        return True

    def run(self, input_files, metadata, output_files):
        """
        Function to run the BWA over a genome assembly FASTA file to generate
        the matching index for use with the aligner

        Parameters
        ----------
        input_files : dict
            List containing the location of the genome assembly FASTA file
        meta_data : dict
        output_files : dict
            List of outpout files generated

        Returns
        -------
        output_files : dict
            index : str
                Location of the index file defined in the input parameters
        output_metadata : dict
            index : Metadata
                Metadata relating to the index file
        """
        results = self.bwa_indexer(
            input_files["genome"],
            output_files["index"]
        )
        results = compss_wait_on(results)

        if results is False:
            logger.fatal("BWA Indexer: run failed")
            return {}, {}

        output_metadata = {
            "index": Metadata(
                data_type="sequence_mapping_index_bwa",
                file_type="TAR",
                file_path=output_files["index"],
                sources=[metadata["genome"].file_path],
                taxon_id=metadata["genome"].taxon_id,
                meta_data={
                    "assembly": metadata["genome"].meta_data["assembly"],
                    "tool": "bwa_indexer"
                }
            )
        }

        return (output_files, output_metadata)
```

(continues on next page)

(continued from previous page)

```
181
182 # ------------------------------------------------------------------------------
```

Troubleshooting Common Issues

Program is installed but fails to run

There are several points that need to be checked in this instance:

1. **Is the program available on your $PATH?** - If not either add it, or place a symlink in a directory that is.
2. **Check that the command that you are running matches the command run by subprocess** - Use the logger.info() to print the command and check that it works.
3. **Subprocess runs commands in a sandbox** - The normal way to run subprocess() is to use *subprocess.Popen(args)* and pass it a list of arguments that represent the command to be run (as shown in the practical example above). Sometimes this fails as extra environment parameters may be required by the program, in this case it is possible to run the whole command as a single string and tell the subprocess to use a shell:

```
1 command_line = "python --version"
2 process = subprocess.Popen(command_line, shell=True)
3 process.wait()
```

4.1.3 HOWTO - Pipelines

This document is a tutorial about creating pipelines that can be easily integrated into the MuG VRE. The aim of a pipeline is to bring together a number of tools (see Creating a Tool) and running them as part of a workflow for end to end processing of data.

Each pipeline consists of the main class for the pipeline, a main function for running the class and a section of global code to catch if the pipeline has been run from the command line. All functions should have full documentation describing the function, inputs and outputs. For details about the coding style please consult the coding style documentation.

Example Pipeline

This example code uses the testTool.py from the Creating a Tool tutorial. The matching code cn be found in the GitHub repository mg-process-test.

There are 2 ways of calling this function, either directly from another program or via the command line.

```
1  #!/usr/bin/env python
2
3  """
4  .. License and copyright agreement statement
5  """
6  from __future__ import print_function
7
8  # Required for ReadTheDocs
9  from functools import wraps # pylint: disable=unused-import
10
11 import argparse
12
```

(continues on next page)

(continued from previous page)

```
from basic_modules.workflow import Workflow
from utils import logger

from mg_process_test.tools.testTool import testTool

# ------------------------------------------------------------------------------

class process_test(Workflow):
    """
    Functions for demonstrating the pipeline set up.
    """

    configuration = {}

    def __init__(self, configuration=None):
        """
        Initialise the tool with its configuration.

        Parameters
        ----------
        configuration : dict
            a dictionary containing parameters that define how the operation
            should be carried out, which are specific to each Tool.
        """
        logger.info("Processing Test")
        if configuration is None:
            configuration = {}

        self.configuration.update(configuration)

    def run(self, input_files, metadata, output_files):
        """
        Main run function for processing a test file.

        Parameters
        ----------
        input_files : dict
            Dictionary of file locations
        metadata : list
            Required meta data
        output_files : dict
            Locations of the output files to be returned by the pipeline

        Returns
        -------
        output_files : dict
            Locations for the output txt
        output_metadata : dict
            Matching metadata for each of the files
        """

        # Initialise the test tool
        tt_handle = testTool(self.configuration)
        tt_files, tt_meta = tt_handle.run(input_files, metadata, output_files)

        return (tt_files, tt_meta)

```

(continues on next page)

(continued from previous page)

```
70
71  # ------------------------------------------------------------------------------
72
73  def main_json(config, in_metadata, out_metadata):
74      """
75      Alternative main function
76      -------------
77
78      This function launches the app using configuration written in
79      two json files: config.json and input_metadata.json.
80      """
81      # 1. Instantiate and launch the App
82      logger.info("1. Instantiate and launch the App")
83      from apps.jsonapp import JSONApp
84      app = JSONApp()
85      result = app.launch(process_test,
86                          config,
87                          in_metadata,
88                          out_metadata)
89
90      # 2. The App has finished
91      logger.info("2. Execution finished; see " + out_metadata)
92
93      return result
94
95  # ------------------------------------------------------------------------------
96
97  if __name__ == "__main__":
98
99      # Set up the command line parameters
100     PARSER = argparse.ArgumentParser(description="Index the genome file")
101     PARSER.add_argument("--config", help="Configuration file")
102     PARSER.add_argument("--in_metadata", help="Location of input metadata file")
103     PARSER.add_argument("--out_metadata", help="Location of output metadata file")
104     PARSER.add_argument("--local", action="store_const", const=True, default=False)
105
106     # Get the matching parameters from the command line
107     ARGS = PARSER.parse_args()
108
109     CONFIG = ARGS.config
110     IN_METADATA = ARGS.in_metadata
111     OUT_METADATA = ARGS.out_metadata
112     LOCAL = ARGS.local
113
114     if LOCAL:
115         import sys
116         sys._run_from_cmdl = True  # pylint: disable=protected-access
117
118     RESULTS = main_json(CONFIG, IN_METADATA, OUT_METADATA)
119     print(RESULTS)
```

Code Walk Through

I'll step through each of the sections of the example code describing what is happening at each point.

Header

This section defines the license and any modules that need to be loaded for the code to run correctly. As a bare minimum is shown in the example with the license, import of the Workflow and Metadata basic_tools and the Data Management (DM) API. Theoretically the pipeline does not have to call a tool, but for completeness this uses the Tool generated as part of the HOWTO - Tools tutorial.

def main_json()

This is the main entry point into the pipeline. It allows the pipeline to be run either locally or as part of a series of function calls within the VRE.

The *main_json()* function is the primary function of the script and is what initiates running the pipeline. It is from here that the VRE or locally run function will call to with any matching input file, defined output files (is required) and any necessary meta data.

At the bottom of the script the *__main__* is triggered when being run from the command line. It can take in parameters from the command line and pass them to the *main_json()* function. As the VRE is responsible for loading of files into the Data Management (DM) API, if files that are used locally are to be tracked then they should also be loaded into the DM API at this point. For clarity of creating a pipeline this has not been included within the example.

Once *main_json()* has been called it launches the *WorkflowApp()* with the name of the pipeline (*process_test* in this case) along with the input files, output files (if known) and relevant meta data for running the application.

process_test - __init__

Instantiates the pipeline and passes on any configuration data to the WorkFlowApp.

process_test - run

This is a required function which is called by the *main_json()* function. It is responsible for orchestrating the flow of data within the pipeline. The run function ensures that the Tools are initiated correctly and are passed the correct variables. If there are multiple Tools in the pipeline each relying on the output from the previous then the *run()* function is responsible for handing the output files from one tool to the next. At this point the handling of files is managed by the pyCOMPSs API and files only become accessible from the final location once the *run()* function has returned to *main_json()*. If you require the output of a tool locally for launching the next then you need to stream the file out of compss, this can be done with the following snippet:

```
if hasattr(sys, '_run_from_cmdl') is True:
    pass
else:
    with compss_open(intermediate_file_in_compss, "rb") as f_in:
        with open(local_loc_for_file, "wb") as f_out:
            f_out.write(f_in.read())
```

This will only work within the COMPSs environment so you will need to test for how your code is getting run.

4.1.4 HOWTO - Documentation

As part of the development of sustainable software it is important that code is well documentated to inform developers that need to implement, extend or replace the code about what it does, the inputs, outputs and any dependencies on other software or code. All classes and functions should have matching documentation.

There are 2 key parts of the documentation, the first is for the classes and functions. The documentation should match the PEP8 standard, an example of this is in the MuG Coding Guidlines. The second part is the Architetural Design Record. The ADR should record why key choices have been made, this is especially true if the choices do not match the borm or there has been a major change in a function (addition, removal or completely rewritten). The ADR provides the reasoning behind the code and the documentation string in the functions describe the code. Between them they provide a log of the development of the project.

An example function description should therefore match the following:

```
"""
Assembly Index Manager

Manges the creation of indexes for a given genome assembly file. If the
downloaded file has not been unzipped then it will get unzipped here.
There are then 3 indexers that are available including BWA, Bowtie2 and
GEM. If the indexes already exist for the given file then the indexing
is not rerun.

Parameters
----------
file_name : str
   Location of the assembly FASTA file

Returns
-------
dict
   bowtie : str
      Location of the Bowtie index file
   bwa : str
      Location of the BWA index file
   gem : str
      Location of the gem index file

Example
-------
.. code-block:: python
  :linenos:

  from tool.common import common
  cf = common()

  indexes = cf.run_indexers('/<data_dir>/human_GRCh38.fa.gz')
  print(indexes)


"""
```

Building the Documentation

Full documentation for a repository can be built using Sphinx. If the pipeline has been developed based on a fork of the mg-process-test repository it can be done by:

```
cd ${mg-process-test}
pip install sphinx

cd docs/
make html
```

Updating the documentation

If new pipelines or tools are added to the repository then it is important that they are included in the documentation.

Updates for a new tool - docs/tool.rst

A new section can be added to the docs/tool.rst file to reflect the new tool.

Before:

```
.. automodule:: tool

   Test Tool
   -----------
   .. autoclass:: tool.testTool.testTool
      :members:
```

After:

```
.. automodule:: tool

   Test Tool
   ---------
   .. autoclass:: tool.testTool.testTool
      :members:

   Test Tool 2
   -----------
   .. autoclass:: tool.testTool.testTool2
      :members:
```

Updates for a new pipeline - docs/pipelines.rst

A new section can be added to the docs/pipelines.rst file to reflect the new pipeline. This requires providing a larger description about the input required for running the pipeline, what it returns and examples about how to run the code locally and within the COMPSs environment.

An example of a pipeline block is as follows:

```
Test Tool
---------
.. automodule:: process_test

   This is a demonstration pipeline using the testTool.

   Running from the command line
   =============================

   Parameters
   ----------
   config : file
      Location of the config file for the workflow
   in_metadata : file
      Location of the input list of files required by the process
   out_metadata : file
```

(continues on next page)

(continued from previous page)

```
      Location of the output results.json file for returned files

   Returns
   -------
   output : file
      Text file with a single entry

   Example
   -------
   To run the script locally this can be done as follows:

   .. code-block:: none
      :linenos:

      cd ${mg-process-test}
      python mg_process_test/process_test.py --config mg_process_test/tests/json/
↪process_test.json --in_metadata mg_process_test/tests/json/input_test.json --out_
↪metadata mg_process_test/tests/results.json --local

   The `--local` parameter should be used if the script is being run within an␣
↪environment where (py)COMPSs is not installed. It can also be used in an␣
↪environment where (py)COMPSs is installed, but the script needs to be run locally␣
↪for testing purposes.

   When using a local verion of the [COMPS virtual machine](http://www.bsc.es/
↪computer-sciences/grid-computing/comp-superscalar/downloads-and-documentation):

   .. code-block:: none
      :linenos:

      cd /home/compss/code/mg-process-test
      runcompss --lang=python mg_process_test/process_test.py --config /home/compss/
↪code/mg-process-test/mg_process_test/tests/json/process_test.json --in_metadata /
↪home/compss/code/mg-process-test/mg_process_test/tests/json/input_test.json --out_
↪metadata /home/compss/code/mg-process-test/mg_process_test/tests/results.json

   Methods
   =======
   .. autoclass:: process_test.process_test
      :members:
```

4.1.5 HOWTO - Licensing

All software developed as part of the VRE by the MuG consortium should be openly licensed using the Apache 2.0 software license. This should encompass the APIs, Tool wrappers and pipelines that have been developed.

Implementing the Apache 2.0 license

There are 3 parts to the license

LICENSE file

This is the full Apache license. This should be an unmodified version of the Apache LICENSE file which can be downloaded from:

```
wget http://www.apache.org/licenses/LICENSE-2.0.txt -O LICENSE
```

Often when starting a new project on GitHub this is automatically generated and in included in the repository by default.

File headers

At the top of all code and documentation there needs to be a header including the license agreement:

```
See the NOTICE file distributed with this work for additional information
regarding copyright ownership.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

This should be surrounded by the appropriate commenting depending on the language

NOTICE file

Lists those that own the Copyright on the software in the repository and the dates that they have been involved with the development of the software. There is then a a second list of institutes that have contributed. Often these will be the same.

```
Multiscale Genomics (MuG)
Copyright 2015-2017 Institute A
Copyright 2015-2016 Institute B
Copyright 2016-2017 Institute C


This product includes software developed at:
- Institute A
- Institute B
- Institute C
```

Benefits for developers

For those that are developing software this format means that there is only a single file that needs to be updated at the start of each year to reflect the involvement of those in the development of the software.

If there are additional licenses that need to match specific sections of code then these can be added at the end of the LICENSE file along with the files that they relate to and who owns the Copyright.

There is a single header for all files referring the reader to the NOTICE file for details about the developers and those that have contributed to the code.

4.1.6 HOWTO - Configuration Files

Tool Description

This configuration file is used to describe the Tool and inform the VRE about what arguments are required by the tool and a list of the file types that can be used as inputs and the matching names that should be used as input parameters.

Below is the example Tool config file for the process_test workflow. It is located in the tool_config directory within the repository. For a full description of all of the parameters please consult the Tool Integration Document.

```
{
    "_id": "process_test",
    "name": "Process Test",
    "title": "Test Workflow",
    "short_description": "Generates file with some text",
    "owner": {
        "institution": "EMBL-EBI",
        "author": "Mark McDowall",
        "contact": "mcdowall@ebi.ac.uk",
        "url": "https://github.com/Multiscale-Genomics/mg-process-test"
    },
    "external": true,
    "has_custom_viewer": false,
    "keywords": [
        "dna"
    ],
    "infrastructure": {
        "memory": 12,
        "cpus": 4,
        "executable": "/home/pmes/code/mg-process-test/process_test.py",
        "clouds": {}
    },
    "input_files": [
        {
            "name": "input",
            "description": "Input file",
            "help": "path to the input file",
            "file_type": ["TXT"],
            "data_type": [
                "text"
            ],
            "required": true,
            "allow_multiple": false
        }
    ],
    "input_files_combinations": [
        [
            "input"
        ]
    ],
    "arguments": [],
```

(continues on next page)

(continued from previous page)

```
    "output_files": [
        {
            "name": "output",
            "required": true,
            "allow_multiple": false,
            "file": {
                "file_type": "TXT",
                "meta_data": {
                    "visible": true,
                    "tool": "process_test",
                    "description": "Output"
                },
                "file_path": "test.txt",
                "data_type": "text",
                "compressed": ""
            }
        }
    ]
}
```

Input Files

The input_files section defines the types of files that are able to be processed. This can be one or many files. Each file object within the list needs to have the follwoing key-pairs:

- name
- description
- help
- file_type
- data_type
- required
- allow_multiple

file_type and data_type can have multiple values in. For example in the case of a DNA sequence this can have the type of "sequence_genomic" or "sequence_dna", so a tool that is able to accept both can have both in the list.

The input_files_combinations is a list of lists of the valid permutations of files that can be accepted by the tool. For example with aligners that are able to hande single or paried-end alignments would need to be able to accept 1 or 2 FASTQ files. These lists use the name value from the input_files file objects.

Arguements

If extra arguements are required by a tool to perform its functions these are defined in the arguements section of the JSON. The arguements section is a list of key-value objects consisting of the following keys:

- name
- description
- help
- type

- required
- default

Examples that can be used within the list include:

```
{
    "name": "test_example_bool_param",
    "description": "Example boolean parameter",
    "help": "Example of a boolean selector",
    "type": "boolean",
    "required": false,
    "default": false
},
{
    "name": "test_example_integer_param",
    "description": "Example integer parameter",
    "help": "Example of an integer input",
    "type": "integer",
    "required": false,
    "default": 5
},
{
    "name": "test_example_string_param",
    "description": "Example string parameter",
    "help": "Example of a string input",
    "type": "string",
    "required": false,
    "default": "default_string_value"
},
{
    "name": "test_example_selector_param",
    "description": "Example selector parameter",
    "help": "Example of a selector input",
    "type": {
        "type": "string",
        "enum": ["abc", "def", "xyz"]
    }
    "required": false,
    "default": "xyz"
}
```

Examples

For larger examples of VRE JSON configuration files have a look at the mg-process-fastq configuration files on GitHub.

Test Configuration Files

There are 2 configuration JSON files as inputs for the test instance. These describe the input and output files and an required arguments that need to get passed to the workflow. These configuration files are those that would get passed to the workflow by the VRE.

config.json

Defines the configurations required for by the pipeline including parameters that need to be passed from the VRE submission form, file and the related metadata as well as the output files that need to be produced by the pipeline.

```
{
    "input_files": [
        {
            "required": true,
            "allow_multiple": false,
            "name": "input",
            "value": "<unique_file_id>"
        }
    ],
    "arguments": [
        {
            "name": "project",
            "value": "run001"
        },
        {
            "name": "execution",
            "value": "/../run001"
        },
        {
            "name": "description",
            "value": null
        },
        {
            "name": "<tool_argument>"
            "value": "<value_from_form>"
        }
    ],
    "output_files": [
        {
            "required": true,
            "allow_multiple": false,
            "name": "output",
            "file": {
                "file_type": "TXT",
                "meta_data": {
                    "visible": true,
                    "tool": "testTool",
                    "description": "Output"
                },
                "file_path": "tests/data/test.txt",
                "data_type": "text",
                "compressed": ""
            }
        }
    ]
}
```

In the arguments there are 2 sets (project and execution) that will always be present and are provided by the VRE at the point of submission of the to the tool. These are the name of the project that has been given in the VRE and is defined by the user. The second is the execution path, this is the location for where the input files are located and can be used as the working directory for the tool. The other parameters in the arguments list are from form elements based on what parameters the tool requires from the user at run time.

input_file_metadata.json

Lists the file location that are used as input. The configuration names should match those that are in the config.json file defined above.

```
[
    {
        "_id": "<unique_file_id>",
        "data_type": "text",
        "file_type": "TXT",
        "file_path": "tests/data/test_input.txt",
        "compressed": 0,
        "sources": [],
        "taxon_id": "0",
        "meta_data": {
            "visible": true,
            "validated": 1
        }
    }
]
```

Examples

For larger examples of JSON configuration files that can be used to test pipelines have a look at the mg-process-fastq test configuration files on GitHub.

4.1.7 HOWTO - Logging

As the pipelines and tools with the MuG VRE environment run without the terminal returning to the user, it is important to have a way to communicate to the user that there is an error with the pipeline. As the code is run within a cluster, the text that is printed to screen won't be returned to the user. Within the Tool API a logging interface has been implemented.

Levels of Logging

When there is an issue it can be passed back to the VRE. These are tracked and passed back to the VRE as the application finishes. There is the option to raise errors in 1 of 6 states:

- **INFO** Confirmation that Tool execution is working as expected.
- **DEBUG** Detailed information, typically of interest only when diagnosing problems.
- **WARNING** An indication that something unexpected happened, but that the Tool can continue working successfully.
- **ERROR** A more serious problem has occurred, and the Tool will not be able to perform some function.
- **FATAL** A serious error, indicating that the Tool may be unable to continue running.
- **PROGRESS** Provide the VRE with information about Tool execution progress, in the form of a percentage (0-100)

Using Logging

The code is present within the Tools API, so adding it into a tool or pipeline requires minimal effort. Improting the loggin functions requires the following code:

```
from utils import logger
```

To add elements to the log can be implemented by:

```
logger.info("Processing Text")
```

This logging has been implemented within the mg-process-test repository within the process_test.py and within the testTool.py scripts. There is no logging within the @task as from this it is possible to return an actual object that can then be checked by the run() function to determine the correct error to return to the main pipeline.

4.1.8 HOWTO - Testing Your Code

Running the Code

To run the code it needs a config.json file and an input_metadata.json file to provide the input.

Running the pipeline manually

```
python mg_process_test/process_test.py --config config.json --in_metadata input_files.
↪json --out_metadata output_metadata.json
```

Testing a Tools and Pipelines

As defined in the coding standards documentation, it is important to generate scripts for testing the functionality of the tools and workflows. If there are then changes to the code, if it raises errors this is identified sooner rather than later. Within python the use of pytest provides the relevant framework around testing code functionality.

Scripts should be placed in the *<repo>/tests* directory.

An example pytest for the test_writer tool:

```
"""
.. See the NOTICE file distributed with this work for additional information
   regarding copyright ownership.

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
"""

```

(continues on next page)

(continued from previous page)

```
from __future__ import print_function

import os.path

from mg_process_test.tool.testTool import testTool

def test_testTool():
    """
    Test case to ensure that the GEM indexer works.
    """
    resource_path = os.path.dirname(__file__)
    text_file = resource_path + "/test.txt"

    input_files = {}

    output_files = {
        "output": text_file
    }

    metadata = {}

    print(input_files, output_files)

    tt_handle = testTool()
    tt_files, tt_meta = tt_handle.run(input_files, metadata, output_files)

    assert output_files['output'] == tt_files['output']
    assert os.path.isfile(text_file) is True
    assert os.path.getsize(text_file) > 0
```

Automated Testing

Once you have defined your test functions it is handle to then hook up the repository with an automated testing framework that can notify you if there are unexpected changes to the behaviour of your code. This is often triggered whenever there is a push to the repository.

Running in COMPSs

It is possible to use a local version of the COMPS virtual machine as used by the MuG VRE. Within the VM is is possible to install any required software. To run the application the following command can then be used:

```
runcompss                                                          \\
   --lang=python                                                   \\
   --library_path=${HOME}/bin                                      \\
   --pythonpath=/<pyenv_virtenv_dir>/lib/python2.7/site-packages/  \\
   --log_level=debug                                               \\
   mg_process_test/process_test.py                                 \\
      --config <repo>/tool_config/process_test.json                \\
      --in_metadata <repo>/tests/json/input_process_test.json      \\
      --out_metadata <repo>/tests/json/output_process_test.json
```

The following is a walk through of developing a tool and pipeline wrapper to include new functionality within the MuG VRE. There are several stages covering the Tool development, using the tool within a pipeline and defining the configuration files required so that the final product can be smoothly integrated into the MuG VRE.

4.1.9 Common Coding Standards

When it comes to developing the code all the code should stick to a common standard. This has been defined within the Coding Standards documentation as well as how to set up the licenses correctly so that your package can be integrated.

4.1.10 Adding a new function

All of the examples in the following sections describe code that has been incorporated into a functional pipeline and tool within a demonstration VRE Tool that is ready for deployment within the VRE. The code can be found in GitHub repository mg-process-test.

> GitHub mg-process-test

In the test process there are example pipelines, tools, documentation, setup scripts unit tests and config files. This repository can be forked and used as the base for developing new pipelines and tools.

The following documents will help guide with the creation of all the components required for creating a tool ready to be integrated into the VRE. To help with the development and Development Checklist has been created to provide a generic guide and checklist to help make sure that nothing has been forgotten.

Wrapping a Tool This section guides you through how to wrap an external tool, or create a tool that utilises the pyCOMPSs framework and should be capable of running within the MuG VRE environment.

Creating a Pipeline Once you have created a tool you can now incorporate one or multiple tools into a pipeline. This will handle the passing of variables from the VRE to the tool and the tracking of outputs ready for handing back to the VRE. This document will also help in creating test input metadata and file location JSON files that are required to run the pipeline.

Documentation This provides a overview of the documentation requirements as described by the MuG Coding Standards.

Logging Takes you through adding logging to your pipelines and tools to return messages to the user via the MuG VRE.

Testing Your Code A important part of making sure that a pipeline or tool is ready for integration is ensuring that the code has been tested. This covers testing the code is functional and that it is capable of running within the infrastructure used by the VRE.

Licensung Your Code The Apache 2.0 license is required for pipelines and tools to to be integrated into the MuG VRE.

VRE Configuration This takes you through creating JSON configuration files for your tool. This should define all the inputs, outputs and any arguments that are required by the pipelines and tools.

4.1.11 Integrating a new tool into the VRE

The next step is the integration of the pipeline/tool into the MuG VRE. The Configuration should provide guide to the initial JSON files. The full JSON specification is located in this GoogleDoc. The details the requirements for correctly creating the Tool description JSON file and the requirements for parameters needed for an application.

4.2 MuG Coding Guidelines

The purpose of this document is to provide a description of the standards that code should conform to so that everything can be share and developed with ease.

4.2.1 Language and Versions

- Python 2.7
- Python 3.6

4.2.2 Installation Method

- PIP

4.2.3 Environment Management

- pyenv
- pyenv-virtualenv

4.2.4 Style

This should follow the PEP8 standard defined by the Python community. Also check out the Google Python Style Guide, a quick and easy reference.

This should be enforced with the use of pylint to ensure that we are matching the PEP8 coding standard.

In addition for every script that is written at the top of ALL python scripts/modules should be the stub license agreement

4.2.5 Header

At the top of all scripts and modules there should be the minified license version for the code. There should also be a full copy of the licence in with the repo as part of the root _dir and a reStructuredText version as part of the documentation.

As part of the head section is also the shebang (*#!*) line. This should only be included if the script is an executable and refer to the form:

```
#!/usr/bin/env python
```

If the file just contains classes and functions then no shebang is required.

4.2.6 Repository Structure

This is based on python coding standards (PEP8) and the requirements for installation (pip) and documentation (as detailed below). The base contents of a git repository should include:

```
<repo_name>/
   docs/
      conf.py
      index.rst
      install.rst
      license.rst
      ...
   <module>/
      __init__.py
      ...
```

(continues on next page)

(continued from previous page)

```
   scripts/
      travis/
         <travis_test_scripts>.sh
   tests/
      data/
      test_<function_name>.py
   .travisci.yml
   LICENSE
   README.md
   requirements.txt
   setup.cfg
   setup.py
```

4.2.7 Documentation

For this we use ReadTheDocs. This is based on the Sphinx annotation servers and the reStructuredText format (Primer and RTD related docs)

The code for a basic setup within a repo is as follows:

```
cd <repo_root>

pip install sphinx

mkdir docs
cd docs
sphinx-quickstart
```

Once the *docs* folder has been generated the documentation can be built with:

```
cd <repo_root>/docs
make html
```

It is advisable to buid the repo locally to remove the majority of the bugs before submitting to GitHub and letting the docs build on RTD.

Common extensions include:

```
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.napoleon',
    'sphinx.ext.viewcode',
]
```

The current theme across all projects is *default*. This can be set like so:

```
html_theme = 'default'
```

There is an issue with the display of code blocks, so there needs to be 2 extra style files:

_static/style.css

```
.rst-content .highlight > pre {
    line-height: 1.5;
}
```

_templates/layout.html

```
{% extends "!layout.html" %}
{% block extrahead %}
    <link href="{{ pathto("_static/style.css", True) }}" rel="stylesheet" type="text/
↪css">
{% endblock %}
```

4.2.8 Classes and Functions

All functions should have matching documentation describing the purpose of the function, the inputs, outputs and where relevant an example piece of code showing how to call the function:

```
"""
Assembly Index Manager

Manges the creation of indexes for a given genome assembly file. If the
downloaded file has not been unzipped then it will get unzipped here.
There are then 3 indexers that are available including BWA, Bowtie2 and
GEM. If the indexes already exist for the given file then the indexing
is not rerun.

Parameters
----------
file_name : str
   Location of the assembly FASTA file

Returns
-------
dict
   bowtie : str
      Location of the Bowtie index file
   bwa : str
      Location of the BWA index file
   gem : str
      Location of the gem index file

Example
-------
.. code-block:: python
  :linenos:

  from tool.common import common
  cf = common()

  indexes = cf.run_indexers('/<data_dir>/human_GRCh38.fa.gz')
  print(indexes)


"""
```

4.2.9 Architectural Design Record (ADR)

For all repositories there should be a document called adr.rst. This should record choices that have been made and summaries the reason for those decisions. This is to provide an in-code record of the design process and reasoning behind why technologies have been selected. In the case of python, pytest, pyenv and pyenv-virtualenv this is the standard setup for use within the pyCOMPSs environment. It is the selection of the key technology that is important for the most part, but there will be times that one technology was chosen over another due to the libraries that are used.

4.2.10 Testing

pytest is the standard in the Python community and has been adopted for testing within the MuG WP4 related code.

As with all python scripts these should have the licence stub and documentation for all functions.

Runs of tests should also tidy up after themselves once they have completed so that the environment is clean ready for the next test case to run. This could mean that some files will get generated multiple times, but these should be smalls sample datasets.

To avoid the use of too many datasets and provide function level testing Mock should be used.

The following options should be used to test code:

```
# Run only the tests
pytest

# Run only pylint as a test
pytest --pylint --pylint-rcfile=pylintrc -m pylint

# Run both
pytest --pylint --pylint-rcfile=pylintrc
```

There will also be times when there are sections of code that are under development or when a test needs to not be included as it is long running or has a bug. To handle this pytest has decorators for this. It a test is to not be used within the TravisCI environment then the following decorator should be used:

```
@pytest.mark.underdevelopment
```

pytest can then be run in the following manner:

```
# Runs all tests
pytest

# Runs only those marked as underdevelopment
pytest -m "underdevelopment"

# Runs all tests except those underdevelopment
pytest -m "not underdevelopment"
```

Sample Data

For all test cases there should be matching datasets that are packaged within the repo.

All datasets should be in the directory *<repo>/tests/data* with a name patching the pattern <script_name>.<species>.<assembly>.fasta for genome files and <script_name>.<accession>.fastq for read files.

Only the raw files should be stored. For testing these should be small files (~100kB).

Large files can be store, but in cases like that it might be best to have a generation script that can calculate the relevant file with the data structure. If this is part of a reader then it should be part of the DM API and stored within the *dm_generator* directory. The script should be runnable from the command line but should also be able to be run by the reader when the *user_id* is *test*. The generated file should be saved to the */tmp/* folder as *sample_<reader-tag>.<file-tag>*.

4.3 License

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without

limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "{}" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search