bio_pype Documentation

Release 2.0.0

Francesco Favero

Jan 26, 2021

Contents

A lightweight python framework to organize bioinformatics scripts and analyses.

CHAPTER 1

About

Much like other pipeline/workflow managers, *bio_pype* offers improvements in scalability, reproducibility while also simplifying the daily use of bioinformatic analyses.

Bio_pype provides a modular framework for building analysis pipelines. Three major components define the behavior of the system: Snippets, Pipelines and Profiles.

Snippets

Snippets represent the basic unit of the pipeline. Snippets essentially act as wrappers that are responsible for packaging command-line arguments, performing any required preprocessing, and executing the target tool. Most importantly, they provide a uniform interface that is used by *bio_pype* to concatenate analysis steps into a pipeline.

Pipelines

A pipeline is a sequential series of steps required to execute a multi-stage analysis. In bio_pype, a pipeline is defined using a YAML file that outlines the order and combination of snippets required.

Profiles

Profiles provide a list of dependencies available to the snippets and pipelines. Information such as the location of genomes, annotation databases, and applications are contained within the profile file. Profiles provide a mechanism for switching between versions of a pipeline that utilise different dependencies while still maintaining the structure of the pipeline.

CHAPTER 2

Get started

2.1 Requirements

Bio-pype is a python package available from the python package index. So *Python* is required to use the software. Only Python3 (Python >= 3.4) is supported.

Note: Earlier version of *bio-pype* supported *python2.7*, however after the official sunsetting of python2 in 2020 and the increasing divergence of the legacy *python2* from *python3*, *bio-pype* will support only *python3* from version *2.0.0*

2.2 Installation

Note: It is strongly advised to use virtualenv to install the module locally.

2.2.1 From pip:

Tip: The installation from pip may not include the latest fixes/features but it is generally a good choice for a production environment.

```
pip install bio_pype
```

2.2.2 From git:

```
git clone https://bitbucket.org/ffavero/bio_pype
cd bio_pype
python -m unittest discover
python setup.py install
```

2.3 Running tasks in a Snippet

You need to configure *bio_pype* to match your system setting (see *Configuration*).

Additionally, in most cases, the *profiles* files section needs to be adjusted to match your system file structure. See *Profiles* for details.

In short, something like the following highlighted section shows a minimal setup to use pype in the example code for this documentation:

Minimal *pype* setup

In this documentation we set the root location for the *pype modules* to the repository test data:

```
$ mkdir ~/.bio_pype
```

```
$ echo "PYPE_MODULES=`dirname $PWD`/test/data" > ~/.bio_pype/config
```

This results in the config file:

```
$ cat ~/.bio_pype/config
PYPE_MODULES=/home/docs/checkouts/readthedocs.org/user_builds/bio-pype/checkouts/
↪latest/test/data
```

Additionally, we also need to replace the *dummy_file* value in the profile files to match the path of *test/data/files/dummy_file.txt* in the repository test data:

```
$ sed -i "s,/just/a/dummy/file/for/testing.txt,`dirname $PWD`/test/data/files/dummy_
↪file.txt," ../test/data/profiles/test_path.yaml
```

```
$ sed -i "s,/just/a/dummy/file/for/testing.txt,`dirname $PWD`/test/data/files/dummy_
↪file.txt," ../test/data/profiles/test_docker.yaml
```

After the configuration of *pype* with the desired *modules* folders, copy the following lines as a file named *test_base.md* (or any other name. The final snippet name corresponds to the file name -minus the .md extension-) in the *snippet* folder.

```
# Example test Snippet

The snippet in pype is given by the file name
(minus the `.md` extension)

## description

Test snippet example

```

(continues on next page)

(continued from previous page)

````
## requirements

```yaml
ncpu: 1
time: '00:01:00'
mem: 1gb
```

## results

```bash
@/bin/sh, yaml


printf  'file_out: %(output)s'
```

## arguments

1. input/i

    - help: input(s) text file
    - type: str
    - required: true
    - nargs: *

2. output/o

    - help: output file
    - type: str
    - default: output.txt

## snippet

> _input_: input profile_dummy_file*

```bash
@/bin/sh, chk1, stdout=chk2, namespace=alpine_3

files_input='%(input)s'

dummy_file='%(profile_dummy_file)s'

cat $files_input $dummy_file | awk '{ print toupper($0) }'

```

> _output_: results_file_out

```bash
@/bin/sh, chk2, namespace=alpine_3

awk '{ print tolower($0) }' > '%(output)s'

```
````

For more detailed information on how to write *snippets* and on their structure see *Snippets*

The *snippets* are run via the *pype* command line:

```
$ pype
usage: pype [-p PROFILE] {pipelines,profiles,repos,snippets} ...

A python pipeliens manager oriented for bioinformatics

positional arguments:
    pipelines       Workflows built combining pipelines and snippets
    profiles        Reference paths and softwares to use in snippets
    repos           Manage pype modules
    snippets        Execute tasks

optional arguments:
  -p PROFILE, --profile PROFILE
                    Choose the pype profile from the available options (see
                    pype profiles). Default: test_docker

This is version 1.9.99 - Francesco Favero - 12 December 2020
```

Using the *snippets* sub-command

```
$ pype snippets
usage: pype snippets [--log LOG]
                     {complement_fa,hello,lower_fa,merge_fa,reverse_fa,test_adv,test_
↪base}
                     ...

positional arguments:
  {complement_fa,hello,lower_fa,merge_fa,reverse_fa,test_adv,test_base}
    complement_fa       lower case a fasta sequence
    hello               hello world with flag
    lower_fa            lower case a fasta sequence
    merge_fa            Concatenate a series of files into a single one
    reverse_fa          reverse a fasta sequence
    test_adv            Test snippet example -in python-
    test_base           Test snippet example

optional arguments:
  --log LOG             Path used for the snippet logs. Default:
                        /home/docs/.bio_pype/logs
```

The selected snippet will prompt the command line interface according to the *arguments* section in the snippet:

```
$ pype snippets test_base
INFO : 2021-01-26 21:06:03,400 : Writing logs to folder /home/docs/.bio_pype/logs/
↪test_base
INFO : 2021-01-26 21:06:03,400 : Using profile test_docker
INFO : 2021-01-26 21:06:03,400 : Prepare snippet test_base
INFO : 2021-01-26 21:06:03,400 : Attempt to execute snippet test_base
error: the following arguments are required: --input/-i
usage: pype snippets test_base --input [INPUT [INPUT ...]] [--output OUTPUT]

optional arguments:
  --input [INPUT [INPUT ...]], -i [INPUT [INPUT ...]]
                        input(s) text file
  --output OUTPUT, -o OUTPUT
                        output file
```

Now that we know how to use it, we may run the *snippets* using any text file available at hand:

```
$ pype -p test_path snippets test_base -i ../test/data/files/input.fa
INFO : 2021-01-26 21:06:03,643 : Writing logs to folder /home/docs/.bio_pype/logs/
↪test_base
INFO : 2021-01-26 21:06:03,643 : Using profile test_path
INFO : 2021-01-26 21:06:03,644 : Prepare snippet test_base
INFO : 2021-01-26 21:06:03,644 : Attempt to execute snippet test_base
INFO : 2021-01-26 21:06:03,648 : Write chunk chk1 code into /home/docs/.bio_pype/logs/
↪test_base/210126210603.644268_69OX_test_base_chk1
INFO : 2021-01-26 21:06:03,649 : Set namespace to path
INFO : 2021-01-26 21:06:03,649 : Write chunk chk2 code into /home/docs/.bio_pype/logs/
↪test_base/210126210603.644324_FWSI_test_base_chk2
INFO : 2021-01-26 21:06:03,650 : Set namespace to path
INFO : 2021-01-26 21:06:03,650 : Pipe in chk1 in chk2 command
INFO : 2021-01-26 21:06:03,650 : Prepare chk1 command line
INFO : 2021-01-26 21:06:03,650 : /home/docs/.bio_pype/logs/test_base/210126210603.
↪644268_69OX_test_base_chk1
INFO : 2021-01-26 21:06:03,650 : Execute chk1 with python subprocess.Popen
INFO : 2021-01-26 21:06:03,651 : Prepare chk2 command line
INFO : 2021-01-26 21:06:03,651 : /home/docs/.bio_pype/logs/test_base/210126210603.
↪644324_FWSI_test_base_chk2
INFO : 2021-01-26 21:06:03,651 : Execute chk2 with python subprocess.Popen
INFO : 2021-01-26 21:06:03,653 : Close  chk1 stdout stream
INFO : 2021-01-26 21:06:03,739 : Terminate chk2
INFO : 2021-01-26 21:06:03,740 : Snippet test_base executed
```

inspecting the output:

```
$ tail -n 2 output.txt
cggacacccagaagtctacatcctaattctc
this is a dummy file!
```

how changed from the original file:

```
$ tail -n 2 ../test/data/files/input.fa
TCAACACCACCTTCTTTGACCCAGCAGGAGGAGGAGACCCAGTACTATACCAGCACCTATTCTGATTCTT
CGGACACCCAGAAGTCTACATCCTAATTCTC
```

This was an amazingly useless task, but the *snippet* showcased the use of the *profiles*, enabling to reuse the same code in different environments (eg using *-p test_docker* instructs to use *docker* to run the code in the snippet chunks), abstracting technical amenities.

Now that you can run a simple task using *profiles* and *snippets*, you can chain tasks to form complex dependencies using *Pipelines*.

CHAPTER 3

Configuration

There are various aspect of *pype* that can be configured. Notably the installation paths of the modules are the most common variables that needs to be set.

By default the pype modules (*snippets*, *pipelines*, *profiles* and *queues*) are installed in the python library installation, within the package *site-packages* folder.

Retrieving the modules in the installation folder can made unnecessarily cumbersome to add/edit/change modules.

Configuring different paths for the modules, it also makes possible to easily switch from a set of modules to another (eg from a "stable" to a "development" set of modules).

The set of configuration variables are listed in the section "*Available variables*"

3.1 Local configuration file

When a configuration file in *~/.bio_pype/config* exists, the program will read the configuration to set the variables.

The configuration file looks like this:

```
PYPE_TMP=/tmp
PYPE_LOGDIR=/tmp/logs
```

3.2 Environment variable

Variables can also be set as environment variables. Setting a variable in the environment will override the corresponding variable if also set in the configuration file.

3.3 Available variables

Variable	Description
PYPE_MODULES	Sets the path of pipelines, profiles, queues and snippets at a prefix with the specified folder. It also overrides the path variable for each separate module.
PYPE_SNIPPETS	Sets the path for the snippets module to the specified folder
PYPE_PROFILES	Seta the path for the profiles module to the specified folder
PYPE_PIPELINES	Sets the path for the pipelines module to the specified folder
PYPE_QUEUES	Sets the path for the queues module to the specified folder
PYPE_REPOS	Sets the path for the repos.yaml file to read
PYPE_NCPU	It can be used to set the number of maximum CPUs to use when launching jobs in parallel without using an external scheduler.
PYPE_MEM	It can be used to set the maximum amount of memory to use, when launching jobs in parallel without using an external scheduler
PYPE_TMP	Sets a temporary folder that can be used in snippets using the '%(pype_tmp)s' tag (or by loading __config__.PYPE_CONFIG in python snippets)
PYPE_LOGDIR	Sets the path for the log files. By default is set to'~/.bio_pype/logs'
PYPE_DOCKER	Sets the binary to launch docker or other container-based solution. The default value is *docker*. I also accepts full path of the binary. It support also *udocker* and *singularity*
PYPE_SINGULARITY_CACHE	Sets the path for the singularity sif images. There is no default path for this variable

CHAPTER 4

Available modules:

4.1 Browse and install available modules

There are available set of snippets and pipelines. You can access the list of repository and manage the installed modules with the *repos* sub command

```
$ pype repos
usage: pype repos [-r REPO_LIST] {list,install,init,clean,info} ...

positional arguments:
  {list,install,init,clean,info}
    list                List the available repositories
    install             Install modules from selected repository
    init                Initiate an empty repository
    clean               Cleanup all module folders
    info                Print location of the modules currently in use

optional arguments:
  -r REPO_LIST, --repo REPO_LIST
                        Repository list. Default:
                        /home/docs/checkouts/readthedocs.org/user_builds/bio-
                        pype/envs/latest/lib/python3.7/site-packages/bio_pype-
                        1.9.99-py3.7.egg/pype/pype_modules/repos.yaml
```

The currently available sets are:

```
$ pype repos list
- gatk4
        Pype modules following the gatk4 best practice
        homepage: https://bitbucket.org/ffavero/pype_modules
        source: https://bitbucket.org/ffavero/pype_modules/get/master.tar.gz
- sequenza
        Pype modules to run sequenza
        homepage: https://bitbucket.org/sequenzatools/sequenza_pype_modules
```

(continues on next page)

(continued from previous page)

```
        source: https://bitbucket.org/sequenzatools/sequenza_pype_modules/get/master.
↪tar.gz
- weischenfeldt
        Pype modules for the Weischenfeldt group tools in computerome
        homepage: https://bitbucket.org/weischenfeldt/pype_weischenfeldt_computerome
        source: https://bitbucket.org/weischenfeldt/pype_weischenfeldt_computerome/
↪get/master.tar.gz
```

It's possible to install the modules from a repository in the list, by invoking the *pype repos install* with the selected repository.

CHAPTER 5

Snippets

A *snippet* is the executor of the tasks. It can be written as a *markdown* file, using *code chunks* to run arbitrary code, or it can be written as a python modules (see *Advanced Snippets in Python*)

5.1 Basic Snippet Structure

A full *snippet* has been shown already in the *Running tasks in a Snippet* section.

The structure of a snippet in composed by the following sections header:

1. *requirements*: which include a code chunk returning a dictionary which specifies the necessary resource to run the snippet (eg. used to allocate resource in a queuing systems)
2. *results*: which include a code chunk returning a dictionary listing all the files produced by the execution of the snippet
3. *arguments*: a numbered list, which is interpreted by `argparse` to produce the command line interface of the snippet
4. *snippet*: containing the code chunks with the instruction to perform the desired task
5. *name*: an optional section containing a chunk returning a string with a "friendly name". This name overrides in certain aspects the default snippet name. This can be used to identify more easily log folders and job ids running on the system.

The input and output arguments are passed to the various chunk via *variable substitutions by name*, a method used in python strings formatting.

In practice it means that a string *%(hello)s* present in a chunk, would be replace by the value of the variable *hello*

There are few ways of setting variables:

1. The *arguments* section
2. The *profiles.files* (See *Profiles*)
3. The keys from the *results* object

The arguments variables are named after the argument name, and the value is the value passed to the the command line.

The variables from the *profile* and from the *results* section are prefixed with *profile_* and *results_* respectively. This means that in order to pass a key, eg. *genome_fa*, present in the *profile.file*, in the snippet chunk it corresponds to *%(profile_genome_fa)s*.

More detail on the argument passing in the following section

5.2 Reference arguments results and files

5.3 Using Namespaces

<<This section may go to *Profiles*>> The *namespace* are set in the *profile* file. Ideally the snippet should be agnostic on the final runtime execution, and it may be possible to run it as-is in different environment by only change the namespace in the profile.

More broadly the *namespace* is a mechanism to set the environment to where execute the chunk.

Supported namespace are:

1. Path: assumes that the commands in the chunks are present in the environment *$PATH*
2. Environment Modules: loads a set of specified modules before running the chunk
3. Docker: run the chunk within a container image. This *namespace* supports also uDocker and singularity

5.3.1 Path

5.3.2 Environment Modules

5.3.3 Docker/Singularity/uDocker

5.4 Advanced Snippets in Python

The snippets are located in a python module (mind the __init__.py in the folder containing the snippets). In order to function, each snippet need to have 4 specific function:

1. *requirements*: a function returning a dictionary with the necessary resource to run the snippet (used to allocate resource in queuing systems)
2. *results*: a function accepting a dictionary with the snippet arguments and returning a dictionary listing all the files produced by the execution of the snippet
3. *add_parser*: a function that implement the `argparse` module and defines the command line arguments accepted by the snippet
4. a function named as the snippet file name (without the *.py* extension), containing the code for the execution of the tool

```
from pype.process import Command


def requirements():
```

(continues on next page)

(continued from previous page)

```
    return({
        'ncpu': 1,
        'time': '00:01:00',
        'mem': '1gb'})


def results(argv):
    output = None
    try:
        output = argv['-o']
    except KeyError:
        try:
            output = argv['--output']
        except KeyError as e:
            raise e
    return({'file_out': output})


def add_parser(subparsers, module_name):
    return subparsers.add_parser(
        module_name, help='Test snippet example -in python-',
        add_help=False)


def test_adv_args(parser, argv):
    parser.add_argument(
        '-i', '--input', dest='input', nargs='*',
        help='input(s) text file', type=str, required=True)
    parser.add_argument(
        '-o', '--output', dest='output', type=str,
        default='output.txt', help='output file')
    return parser.parse_args(argv)


def test_adv(subparsers, module_name, argv, profile, log):
    args = test_adv_args(
        add_parser(subparsers, module_name), argv)

    dummy_file = profile.files['dummy_file']

    cmd1 = 'cat %s %s' % (
        ' '.join(args.input), dummy_file)
    cmd2 = 'awk \'{ print toupper($0) }\''
    cmd3 = 'awk \'{ print tolower($0) }\''

    cat = Command(
        cmd1, log, profile, 'cat')
    to_up = Command(
        cmd2, log, profile, 'to_up')
    to_low = Command(
        cmd3, log, profile, 'too_low')
    for input_file in args.input:
        cat.add_input(input_file)
    cat.add_input(dummy_file)
    to_low.add_output(args.output)
    cat.add_namespace(profile.programs['alpine_3'])
    to_up.add_namespace(profile.programs['alpine_3'])
```

(continues on next page)

(continued from previous page)

```
    to_low.add_namespace(profile.programs['alpine_3'])
    to_up.pipe_in(cat)
    to_low.pipe_in(to_up)

    with open(args.output, 'wt') as output:
        to_low.run()
        for line in to_low.stdout:
            output.write(line.decode('utf-8'))
        to_low.close()
```

CHAPTER 6

Pipelines

6.1 Structure of Pipelines files:

The pipelines are YAML files located in the pipelines folder (see *Configuration*).

A *pipeline* YAML file is structured in two sections defined by the keys *info* and *items*.

6.1.1 Info header

The *info* section contains the following sub-keys

Table 1: Pipeline Info

keys	values
description	A quick description of the pipeline
date	The date of writing/editing
arguments	An optional key containing information used to customize the description of the pipeline arguments

6.1.2 Pipeline items

The *items* section contains a hierarchical structure, constructed by combining multiple *items*.

Each element is composed by the following keys:

Table 2: Pipeline Item

key	value
name	The name of the pipeline or the snippet which the item represent
type	Indicate if the item is a snippet or a pipeline. The value can be any between the choices: *snippet*, *pipeline*, *batch_snippet* or *batch_pipeline*
arguments	The list of arguments of the pipeline/snippets with the template variables used. See *Pipeline Item Arguments* section
dependencies	Optional, Indicate the start of a child *items* structure constructing a hierarchical dependency between *snippets*
requirements	Optional, a dictionary used to alter the default'requirements' defined in the snippet (it applies only for *snippets* and *batch_snippets*)
mute	Optional, if set and its value is *true*, the item will not pass any dependency on the parent items

6.2 Pipeline Item Arguments

CHAPTER 7

Profiles

7.1 Extending profiles:

Similarly to the pipelines, the profiles are YAML files in a python module (it is requires a __init__.py file in the folder containing the files). In order to be parsed correctly, the profile YAML file needs to have a specific structure:

```
info:
   description: Test Profile
   date:        23/11/2020
genome_build: hg38
files:
   genome_fa:    /abs/path/to/fasta.fa
   genome_fa_gz: /abs/path/to/fasta.fa.gz
   dummy_file:   /home/docs/checkouts/readthedocs.org/user_builds/bio-pype/checkouts/
↪latest/test/data/files/dummy_file.txt

programs:
   samtools_0:
      namespace: path@samtools
      version: 0.1.19
   samtools_1:
      namespace: path@samtools
      version: 1.2
   bwa:
      namespace: path@bwa
      version: 0.7.10
   star:
      namespace: path@star
      version: 2.5.1b
   alpine_3:
      namespace: path@alpine
      version: latest
```

```
info:
   description: Test Profile with Docker Namespace
   date:        23/11/2020
genome_build: hg38
files:
   genome_fa:    /abs/path/to/fasta.fa
   genome_fa_gz: /abs/path/to/fasta.fa.gz
   dummy_file:   /home/docs/checkouts/readthedocs.org/user_builds/bio-pype/checkouts/
↪latest/test/data/files/dummy_file.txt

programs:
   samtools_0:
      namespace: path@samtools
      version: 0.1.19
   samtools_1:
      namespace: docker@biocontainers/samtools
      version: v1.7.0_cv4
   bwa:
      namespace: docker@biocontainers/bwa
      version: v0.7.15_cv4
   star:
      namespace: path@star
      version: 2.5.1b
   alpine_3:
      namespace: docker@alpine
      version: latest
```

CHAPTER 8

API

8.1 Utils

8.1.1 Pipelines

class pype.utils.arguments.**BatchFileArgument**(*argument*)
BatchFileArgument read the arguments from a file and return the list of arguments. It is required for the execution of a batch snippet or batch pipeline.

class pype.utils.arguments.**BatchListArgument**(*argument*)
BatchArgument read the arguments from a file and return the list of arguments. It is required for the execution of a batch snippet or batch pipeline.

class pype.utils.arguments.**CompositeArgument**(*argument*)
A CompositeArgument retrieve the results from the results method of the specified snippet. It will not appear listed in the arguments help message so it's value is None. In itself it contains a *PipelineItemArguments* object, defining the argument to pass to the results method of the snippets

class pype.utils.arguments.**ConstantArgument**(*argument*)
xxxx

class pype.utils.arguments.**PipelineItemArguments**
An object to gather the Argument of a PipelineItem.

This is meant to collect the structure and the type of the arguments defined in a pipeline yaml file.

add_argument(*argument*, *argument_type='argv_arg'*)
Add the appropriate Argument class to the *PipelineItemArguments* argument list

Parameters

- **argument** (*dict*) – An item from the list of arguments from the pipeline yaml file. It should contain the keys *prefix* an *pipeline_arg*. The key *prefix* indicate the flag usd in the snippet/pipeline to which the PipelineItem is configured to execute. The key *pipeline_arg* indicate the keyword or object that the pipeline engine need to interpret to convert into arguments and also to construct the command line interface and.

- **argument_type** (*str*) – The type of argument, this parameter will select which argument class would be used to parse the argument. possible choices are composite_arg, batch_list_arg and argv_arg. Default argv_arg.

to_dict(*args_dict=None*)

Converts the argument in the *PipelineItemArguments* into dictionaries simlar to argparse

Example

8.1.2 Queues

class pype.utils.queues.**SnippetRuntime**(*command*, *log*, *profile*)

A class to help building queue modules implementation for *bio_pype*.

An helper class that generalize various tasks to build queues modules and in the meantime creates a *yaml* file that records running jobs and job dependencies, agnostic of the underlying queueing system used.

Parameters

- **command** (*str*) – The snippet name with valid arguments
- **log** (*pype.logger.PypeLogger*) – Log object of the main pipeline
- **profile** (*str*) – The name of the selected profile

A Usage example of this class is the following implementation of the pbs (torque) queue system:

Listing 1: ../test/data/queues/pbs.py

```
import os
import datetime
from pype.utils.queues import SnippetRuntime


def submit(command, snippet_name, requirements, dependencies, log, profile):
    runtime = SnippetRuntime(command, log, profile)
    runtime.get_runtime(requirements, dependencies)
    queue_dependencies = runtime.queue_depends()
    stdout = os.path.join(log.__path__, 'stdout')
    stderr = os.path.join(log.__path__, 'stderr')
    stdout_pbs = os.path.join(log.__path__, 'stdout.pbs')
    stderr_pbs = os.path.join(log.__path__, 'stderr.pbs')

    now = datetime.datetime.now()
    now_plus_10 = now + datetime.timedelta(minutes=10)
    startime_str = now_plus_10.strftime("%H%M.%S")

    log.log.info('Execution qsub into working directory %s' % os.getcwd())
    log.log.info('Redirect stdin/stderr to folder %s' % log.__path__)
    command = '''#!/bin/bash
    exec 1>%s
    exec 2>%s
    exec %s''' % (stdout, stderr, runtime.command)
    log.log.info('Retrive custom group environment variable')
    largs = []

    if len(queue_dependencies) > 0:
        cmd_dependencies = [
            'afterok:%s' % dep for dep in queue_dependencies]
```

(continues on next page)

(continued from previous page)

```
        depend = ['-W', 'depend=%s' % ','.join(cmd_dependencies)]
        largs += depend
    if 'time' in requirements.keys():
        time = ['-l', 'walltime=%s' % requirements['time']]
        largs += time
    if 'mem' in requirements.keys():
        mem = ['-l', 'mem=%s' % requirements['mem']]
        largs += mem
    if 'type' in requirements.keys():
        if requirements['type'] == 'exclusive':
            exclusive = ['-l', 'naccesspolicy=singlejob']
            largs += exclusive
    if 'ncpu' in requirements.keys():
        try:
            nodes = int(requirements['nodes'])
        except KeyError:
            nodes = 1
        cpus = ['-l', 'nodes=%i:ppn=%i' % (nodes, int(requirements['ncpu']))]
        largs += cpus
    qsub_group = os.environ.get('PYPE_QUEUE_GROUP')
    if qsub_group:
        log.log.info('Custom qsub group set to %s' % qsub_group)
        largs += ['-W', 'group_list=%s' % qsub_group, '-A', qsub_group]
    else:
        log.log.info('Custom qsub group not set')
    echo = 'echo \'%s\'' % command
    qsub = [
        'qsub', '-V', '-o', stdout_pbs, '-e', stderr_pbs,
        '-d', os.getcwd(), '-a', startime_str, '-N', snippet_name] + largs
    runtime.add_queue_commands(
        [echo, ' '.join(qsub)])
    runtime.submit_queue(5)
    runtime.commit_runtime()
    return(runtime.run_id)


def post_run(log):
    log.log.info('Done')
```

add_queue_commands(*commands*)
Add the list of commands to launch the job in the queue system.

The commands will be run in a pipe, so the output of the first item in the command list will be *stdin* of the second item, and so on.

Parameters **commands** (*list*) – List of string with the commands

add_queue_id(*queue_id*)
Add a job ID for the snippet.

This is useful when the queue command is not submitted using *SnippetRuntime.submit_queue()*, so the job id is not automatically registered in the runtime object.

Parameters **queue_id** (*str*) – Job id string

change_sleep(*sleep_sec*)
Change the number of seconds to wait after submitting a job in the queue system.

It is used in *SnippetRuntime.submit_queue()*. It alters the attribute `SnippetRuntime.`

sleep

Parameters **sleep_sec** (*int*) – Number of seconds

commit_runtime()
Save the runtime dictionary in the *pipeline_runtime.yaml* file

The path of *pipeline_runtime.yaml* is the parent directory of the snippet log.

get_runtime(*requirements*, *dependencies*)
Load the runtime object, if does not exists initiate a new runtime dictionary.

Parameters

- **requirements** (*dict*) – Dictionary specifying the snippet requirements
- **dependencies** (*list*) – List of other snippets ids to which this snippets depends (it will run if/when the other job are terminated)

queue_depends()
Returns the list of queue ids to which this command depends

The list in the runtime dictionary, in the key *dependencies* consinst on unique ids of the runtime object, this methods simply converts the runtime ids into queue ids.

Returns Queue id dependency list

Return type list

submit_queue(*retry=1*)
Execute the queue commands, and add the resulting job id in the runtime dictionary.

The method accepts a number of *retry* attempts, which will enable to reiterate the specified number of time in case of failure, before failing the pipeline

Parameters **retry** (*int, optional*) – Number of attempts before failing, defaults to 1

8.1.3 Snippets and Profiles

8.2 Process

class pype.process.**Command**(*cmd*, *log*, *profile*, *name=''*)
High level class to use `subprocess.Popen` combined with *Volume* and *Namespace* classes.

The *Command* class is a wrapper around `subprocess.Popen` that results in a more succinct code, increasing the readability of the command lines that are going to be executed rather then the `subprocess.Popen` boilerplate.

The class initialization requires the command line string, a `Profile` class and a log object (eg the *snippet* log object).

Parameters

- **cmd** (*str*) – Command line string
- **log** (`pype.logger.PypeLogger`) – Log class of the running snippet
- **profile** (`pype.utils.profiles.Profile`) – A Profile object
- **name** (*str, optional*) – String used to identify the process in the log, defaults to ''

add_input(*in_file*, *match='exact'*)
The match argument can be either exact or recursive. - exact will match only the specified file - recursive will match all the file with the same prefix

of the specified file

[summary]

[extended_summary]

Parameters

- **in_file** (*[type]*) – [description]
- **match** (*str, optional*) – [description], defaults to 'exact'

add_namespace(*namespace*)
[summary]

[extended_summary]

add_output(*out_file*)
[summary]

[extended_summary]

Parameters **out_file** (*[type]*) – [description]

add_volume(*path*, *output=False*)
[summary]

[extended_summary]

Parameters

- **path** (*[type]*) – [description]
- **output** (*bool, optional*) – [description], defaults to False

child_close()
[summary]

[extended_summary]

close()
[summary]

[extended_summary]

Returns [description]

Return type [type]

docker(*local_script*)
[summary]

[extended_summary]

Raises **Exception** – [description]

pipe_in(*command*, *local_script=False*)
[summary]

[extended_summary]

Parameters

- **command** (*[type]*) – [description]

- **local_script** (*bool, optional*) – [description], defaults to False

replace_values_in_code(*code_file*)
[summary]

[extended_summary]

Parameters **code_file** (*[type]*) – [description]

run(*local_script=False*)
[summary]

[extended_summary]

Parameters **local_script** (*bool, optional*) – [description], defaults to False

class pype.process.**Namespace**(*program_dict*, *log*, *profile*)
A mechanism to load different environments

Define a basic abstraction layer to load programs and environments to the *Command* class

[summary]

[extended_summary]

Parameters

- **program_dict** (*dict*) – A dictionary with the following keys *namespace*, *version*, *dependencies*. *namespace* is a string composed by the the namespace type and the namespace item, separated by the @ character. The supported namespace types are *docker*, *env_modules* and *path*. the namespace item is a string relevant to the namespace type (eg. the docker container repository url). the *version* is a string defining the tag/version of the docker container or the version of the program to load (again, depending on the namespace type selected). *dependencies* is a key only used for the *env_modules* namespace and is used to load other environment modules to satisfy the loading dependencies.
- **log** (pype.logger.PypeLogger) – Log object to append logging in the snippet log file
- **profile** (pype.utils.profiles.Profile) – Profile object

Raises

- **SnippetNamespaceError** – *Wrong Namespace format* if the namespace does have more then @ characters.
- **SnippetNamespaceError** – *Not supported namespace* if the *namespace type* is not *docker*, *env_modules* or *path*.
- **SnippetNamespaceError** – *All dependencies must be type env_module* if some of the dependencies defined in the *dependencies* key is not a namespace of the *env_modules* type.

class pype.process.**Volume**(*path*, *output=False*, *bind_prefix='/var/lib/pype'*)
Volume class to abstract and parametrize the binding of files while running commands in containerized environments.

The class contains also method to adjust the bind volume argument to implementation such as *udocker* and *singularity*.

Init the class defining the path in the host environment, the prefix in the container environment and flagging if the path is a input or an output target

Parameters

- **path** (*str*) – File or directory to bind in the host system

- **output** (*bool, optional*) – Set to *True* if *path* is an output target, defaults to *False*.
- **bind_prefix** (*str, optional*) – Prefix path in the container environment, defaults to '/var/lib/pype'.

remove_mode()
Removes the trainling mode (eg the ending *:rw*) from the bining string.

replace_bind_dirname(*bind_path*)
Replaces the bind volume in the container environment with the *dirname* of the specified *bind path*.

This is useful to give the same binding point to multiple paths (defined in multiple *Volume* classes) that are in the same folder in the host system.

Parameters **bind_path** (*str*) – Binding point to replace instead of the current one randomly generated by the class.

replace_bind_volume(*bind_path*)
Replaces the bind volume in the container environment with the specified *bind path*.

This is useful to manage binding point to multiple paths (defined in multiple *Volume* classes) that are subfolders of another bind volume in the host system.

Parameters **bind_path** (*str*) – Binding point to replace instead of the current one randomly generated by the class.

singularity_volume()
Format the volume binding string following the *singularity* command line syntax.

to_str()
Returns a string with the bind volume argument relative to the content of the class

Returns Bind volume string

Return type str

8.3 Misc

class pype.misc.**DefaultHelpParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*)

error(*message: string*)
Prints a usage message incorporating the message to stderr and exits.

If you override this in a subclass, it should not return – it should either exit or raise an exception.

class pype.misc.**SubcommandHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

pype.misc.**xopen**(*filename*, *mode='r'*)
Wrap around open/gzip.open and stdin/out.

Replacement for the "open" function that can also open files that have been compressed with gzip. If the filename ends with .gz, the file is opened with gzip.open(). If it doesn't, the regular open() is used. If the filename is '-', standard output (mode 'w') or input (mode 'r') is returned.

class pype.binfmisc.**fastq**(*f*, *n=-1*)
Fastq iterator to extract name, sequence and quality ofr each read.

Specify the file object to iterate.

Parameters

- **f** (*File*) – fastq file
- **n** (*int, optional*) – number of reads to evaluate, defaults to -1

8.4 Logging/Exceptions

exception pype.exceptions.**CommandNamespaceError**

exception pype.exceptions.**EnvModulesError**

exception pype.exceptions.**PipelineError**

exception pype.exceptions.**PipelineItemError**

exception pype.exceptions.**ProfileError**

exception pype.exceptions.**SnippetError**

CHAPTER 9

Indexes and tables

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

T

V

X