pypiper Documentation

Release 0.7.0

Nathan Sheffield, Johanna Klughammer, Andre Rendeiro

Dec 14, 2017

Getting Started

Making robust pipelines just got easier. Pypiper helps you take your current pipeline (e.g. shell script) and make it better with minimal effort on your part. Pypiper is a lightweight python toolkit that helps you write slick pipelines in python. You'll be running in minutes. Interested? Proceed to the *Introduction* or jump straight to the *Tutorials*. You can find the source code on github.

CHAPTER 1

Contents

1.1 Introduction

Pypiper is a lightweight python toolkit for gluing together restartable command line pipelines. With Pypiper, **simplicity is paramount**. It should take less than 15 minutes to build your first pipeline. Learning all the *features and benefits* takes just an hour or two. At the same time, Pypiper provides immediate advantages over a simple shell script.

Pypiper is an example of a simple bioinformatics pipeline framework.

Many bioinformatics pipelines are written by students or technicians who don't have time to learn a full-scale pipelining framework, so they just end up using simple bash scripts to piece together commands. Pypiper tries to give 80% of the benefits of a professional-scale pipelining system while requiring very little additional effort.

Just take your bash script and pass those commands through `PipelineManager.run()` and you will get automatic restartability, process monitoring for memory use and compute time, pipeline status monitoring, copious log output, robust error handling, easy debugging tools, guaranteed file output integrity, and a bunch of useful pipeline development helper functions.

With Pypiper, simplicity is paramount. A user can start building useful pipelines using Pypiper in under 15 minutes. At the same time, using Pypiper provides immediately clear and significant advantages over a simple bash script.

1.1.1 Who should use Pypiper?

If you have a shell script that would benefit from a layer of "handling code", Pypiper helps you convert that set of shell commands into a production-scale workflow, automatically handling the annoying details (restartablilty, file integrity, logging) to make your pipeline robust and restartable.

If you need a full-blown, datacenter-scale environment that can do everything, look elsewhere. Pypiper's strength is its simplicity. If all you want is a shell-like script, but now with the power of python, and restartability, then Pypiper is for you.

This emphasis on simplicity provides a few advantages:

- Write your pipeline in pure python (no new language to learn).

- Pypiper is easy to learn (3 or 4 functions will be all you need for simple stuff)
- Pypiper does not assume you want a complex dependency structure. You write a simple **ordered sequence of commands**, just like a shell script.

1.1.2 What Pypiper does NOT do

Pypiper tries to exploit the Pareto principle – you'll get 80% of the features with only 20% of the work of other pipeline management systems. So, there are a few things Pypiper deliberately doesn't do:

- Job dependencies. Pypiper runs sequential pipelines. If you want to implement a pipeline with complex task dependencies, there are better options. We view this as an advantage because it makes the pipeline easier to write, easier to understand, and easier to debug. For developmental pipelines, the complexity cost is not worth the minimal benefit – read this post on parallelism in bioinformatics for an explanation.
- Cluster submission. Pypiper does not handle any sort of cluster job submission or resource requesting. Instead, we have divided this into a separate project called looper. This makes a modular system: you can use whatever system you want for cluster management. Pypiper builds individual, single-sample pipelines that can be run one sample at a time. Looper then processes groups of samples, submitting appropriate pipelines to a cluster or server. The two projects are independent and can be used separately, but they are most powerful when combined.

1.1.3 Yet another pipeline system?

As I began to put together production-scale pipelines, I found a lot of relevant pipelining systems, but was universally disappointed. For my needs, they were all overly complex. I wanted something **simple enough to quickly write and maintain** a pipeline without having to learn a lot of new functions and conventions, but robust enough to handle requirements like restartability and memory usage monitoring. Everything related was either a pre-packaged pipeline for a defined purpose, or a heavy-duty development environment that was overkill for a simple pipeline. Both of these seemed to be targeted toward ultra- efficient uses, and neither fit my needs: I had a set of commands already in mind – I just needed a wrapper that could take that code and make it automatically restartable, logged, robust to crashing, easy to debug, and so forth.

Pypiper fills a niche: the individual graduate student who is just putting together a simple series of commands and wants to run that on a few hundred or thousand samples. It's not worth learning a workflow development language and a complex data-center-class tool for your simple research pipeline.

1.2 Installing and Hello World

Release versions are posted on the GitHub releases page. You can install the latest version directly from GitHub using pip:

```
pip install --user https://github.com/epigen/pypiper/zipball/master
```

Update with:

```
pip install --user --upgrade https://github.com/epigen/pypiper/zipball/master
```

Now, to test pypiper, follow the commands in the `Hello, Pypiper!` tutorial: just run these 3 lines of code and you're running your first pypiper pipeline!

```
# Install the latest version of pypiper:
pip install --user https://github.com/epigen/pypiper/zipball/master
```

```
# download hello_pypiper.py
wget https://raw.githubusercontent.com/epigen/pypiper/master/example_pipelines/hello_
↪pypiper.py

# Run it:
python hello_pypiper.py
```

The actual code, in `hello_pypiper.py` is a very simple but complete pipeline:

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

import pypiper
outfolder = "hello_pypiper_results" # Choose a folder for your results
pm = pypiper.PipelineManager(name="hello_pypiper", outfolder=outfolder)


pm.timestamp("Hello!")
target_file = "hello_pypiper_results/output.txt"
cmd = "echo 'Hello, Pypiper!' > " + target_file
pm.run(cmd, target_file)

pm.stop_pipeline()
```

When you run it, you should see printed to screen some output like this:

```
----------------------------------------
##### [Pipeline run code and environment:]
*               Command:  `hello_pypiper.py`
*          Compute host:  puma
*           Working dir:  /home/nsheff
*             Outfolder:  hello_pypiper_results/
*  Pipeline started at:   (08-24 12:34:34) elapsed:0:00:00 _TIME_

##### [Version log:]
*       Python version:  2.7.12
*          Pypiper dir:  `/home/nsheff/.local/lib/python2.7/site-packages/pypiper`
*      Pypiper version:  0.6.0
*         Pipeline dir:  `/home/nsheff`
*     Pipeline version:  None

##### [Arguments passed to pipeline:]

----------------------------------------


Change status from initializing to running
Hello! (08-24 12:34:34) elapsed:0:00:00 _TIME_

Target to produce: `hello_pypiper_results/output.txt`
> `echo 'Hello, Pypiper!' > hello_pypiper_results/output.txt`

<pre>
</pre>
Process 2325 returned: (0). Elapsed: 0:00:00.

Change status from running to completed
> `Time`         0:00:00 hello_pypiper   _RES_
> `Success`      08-24-12:34:34  hello_pypiper   _RES_
```

```
##### [Epilogue:]
*   Total elapsed time:  0:00:00
*     Peak memory used:  0.0 GB
* Pipeline completed at:  (08-24 12:34:34) elapsed:0:00:00 _TIME_

Pypiper terminating spawned child process 2318...
child process terminated
```

This output is printed to your screen and also recorded in a log file (called `hello_pypiper_log.md`). There are a few other outputs from the pipeline as well. All results are placed in a folder called `hello_pypiper_results`. Navigate to that folder to observe the output of the pipeline, which will include these files:

- hello_pypiper_commands.sh
- hello_pypiper_completed.flag
- hello_pypiper_log.md
- hello_pypiper_profile.tsv
- output.txt
- stats.tsv

These files are explained in more detail in the next section, so head right on over to *outputs explained* or to the *features list*. If you're ready, you can also just dive right in with some *more in-depth tutorials*.

1.3 Outputs explained

Assume you are using a pypiper pipeline named *PIPE* (it passes `name="PIPE"` to the PipelineManager constructor). By default, your `PipelineManager` will produce the following outputs automatically (in addition to any output created by the actual pipeline commands you run):

- **PIPE_log.md** The log starts with a bunch of useful information about your run: a starting timestamp, version numbers of the pipeline and pypiper, a declaration of all arguments passed to the pipeline, the compute host, etc. Then, all output sent to screen is automatically logged to this file, providing a complete record of your run.
- **PIPE_status.flag** As the pipeline runs, it produces a flag in the output directory, which can be either `PIPE_running.flag`, `PIPE_failed.flag`, or `PIPE_completed.flag`. These flags make it easy to assess the current state of running pipelines for individual samples, and for many samples in a project simultaneously.
- **stats.tsv** Any results reported by the pipeline are saved as key-value pairs in this file, for easy parsing.
- **PIPE_profile.md** A profile log file that provides, for every process run by the pipeline, 3 items: 1) the process name; 2) the clock time taken by the process; and 3) the memory high water mark used by the process. This file makes it easy to profile pipelines for memory and time resources.
- **PIPE_commands.md** Pypiper produces a log file containing all the commands run by the pipeline, verbatim. These are also included in the main log.

Multiple pipelines can easily be run on the same sample, using the same output folder (and possibly sharing intermediate files), as the result outputs will be identifiable by the `PIPE_` identifier.

These files are markdown making it easy to read either in text format, or to quickly convert to a pretty format like HTML.

1.4 Features at-a-glance

Pypiper provides the following benefits:

- **Restartability:** Commands check for their targets and only run if the target needs to be created, much like a *makefile*, making the pipeline pick up where it left off in case it needs to be restarted or extended.
- **Pipeline integrity protection:** PyPiper uses file locking to ensure that multiple pipeline runs will not interfere with one another – even if the steps are identical and produce the same files. One run will seamlessly wait for the other, making it possible to share steps seamlessly across pipelines.
- **Memory use monitoring:** Processes are polled for high water mark memory use, allowing you to more accurately guage your future memory requirements.
- **Easy job status monitoring:** Pypiper uses status flag files to make it possible to assess the current state (*running*, *failed*, or *completed*) of hundreds of jobs simultaneously.
- **Robust error handling:** Pypiper closes pipelines gracefully on interrupt or termination signals, converting the status to *failed*. By default, a process that returns a nonzero value halts the pipeline, unlike in bash, where by default the pipeline would continue using an incomplete or failed result. This behavior can be overridden as desired with a single parameter.
- **Logging:** Pypiper automatically records the output of your pipeline and its subprocesses, and provides copious information on pipeline initiation, as well as easy timestamping.
- **Easy result reports:** Pypiper provides functions to put key-value pairs into an easy-to-parse stats file, making it easy to summarize your pipeline results.
- **Simplicity:** It should only take you 15 minutes to run your first pipeline. The basic documentation is just a few pages long. The codebase itself is also only a few thousand lines of code, making it very lightweight.
- **Dynamic recovery:** If a job is user-interrupted (with SIGINT or SIGTERM), for example by a cluster resource manager, it will get a dynamic recovery flag set, and the next time the run is started it will automatically pick up where it left off.

Furthermore, Pypiper includes a suite of commonly used pieces of code (toolkits) which the user may use to build pipelines.

1.5 Where to find pipelines

Independent pypiper pipelines are built by many authors at various locations. There's no comprehensive list of all available pipelines, but here's a list of some publicly available pipelines that will give you examples of production-ready pipelines built using pypiper.

- https://github.com/epigen/open_pipelines - A collection of pipelines to process diverse sequencing data types, including ATAC-seq, RNA-seq, and others.
- https://github.com/databio/rnapipe - A set of pipelines specifically for processing RNA.
- https://github.com/databio/dnameth_pipelines - Pipelines for Whole Genome and Reduced Representation Bisulfite Sequencing (WGBS and RRBS).

1.6 Using pipelines

Pypiper pipelines are python scripts. There is no special requirement or syntax, you run it as you would any python script:

```
python pipeline.py
```

Or, if you make the script executable (`chmod o+x pipeline.py`) and it has a shebang at the top (`#!/usr/bin/env python`), you can execute it directly:

```
./pipeline.py
```

Now, you'll need to figure out the command-line arguments required by the pipeline. Usage will vary based on the script, and the pipeline author determines what command line arguments their pipeline will recognize. You should look in the documentation for your pipeline, or you can often figure out the command-line interface by passing the `--help argument`, like so: `python.py --help`.

With that said, there are a few universal (Pypiper-added) options that are frequently (but not necessarily always) honored by pypiper pipelines. These default pypiper arguments are detailed below:

- **`-R, --recover`** Recover mode, overwrite locks. This argument will tell pypiper to recover from a failed previous run. Pypiper will execute commands until it encounters a locked file, at which point it will re-execute the failed command and continue from there.
- **`-F, --follow`** Force run follow-functions. By default, follow-functions are only run if their corresponding `run` command was run; with this option you can force all follow functions to run. This is useful for regenerating QC data on existing output. For more details, see *the follow argument*.
- **`-D, --dirty`** Make all cleanups manual. By default, pypiper pipelines will delete any intermediate files. For debugging, you may want to turn this option off – you can do that by specifying **dirty mode**.

1.7 Tutorials

Start with the 'your first pipeline' tutorial to get a quick overview of a simple pipeline, then look through the other examples for more advanced concepts.

1.7.1 Your first pipeline

Using pypiper is simple. Your pipeline is a python script, say *pipeline.py*. First, import pypiper, specify an output folder, and create a new `PipelineManager` object:

```
#!/usr/bin/env python
import pypiper, os
outfolder = "pipeline_output/" # Choose a folder for your results
pm = pypiper.PipelineManager(name="my_pipeline", outfolder=outfolder)
```

This creates your `outfolder` and places a flag called `my_pipeline_running.flag` in the folder. It also initializes the log file (`my_pipeline_log.md`) with statistics such as time of starting, compute node, software versions, command-line parameters, etc.

Now, the workhorse of `PipelineManager` is the `run()` function. Essentially, you just create a shell command as a string in python, and then pass it and its target (a file it creates) to `run()`. The target is the final output file created by your command. Let's use the built-in `shuf` command to create some random numbers and put them in a file called `outfile.txt`:

```
# our command will produce this output file
target = os.path.join(outfolder, "outfile.txt")
command = "shuf -i 1-500000000 -n 10000000 > " + target
pm.run(command, target)
```

The command (`command`) is the only required argument to `run()`. You can leave `target` empty (pass `None`). If you **do** specify a target, the command will only be run if the target file does not already exist. If you **do not** specify a target, the command will be run every time the pipeline is run.

Now string together whatever commands your pipeline requires! At the end, terminate the pipeline so it gets flagged as successfully completed:

```
pm.stop_pipeline()
```

That's it! By running commands through `run()` instead of directly in bash, you get a robust, logged, restartable pipeline manager for free!

Go to the next page (*basic tutorial*) to see a more complicated example.

1.7.2 Basic tutorial

Now, download basic.py and run it with *python basic.py* (or, better yet, make it executable (*chmod 755 basic.py*) and then run it directly with *./basic.py*). This example is a documented vignette; so just read it and run it to get an idea of how things work.

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

"""Getting Started: A simple sample pipeline built using pypiper."""

# This is a runnable example. You can run it to see what the output
# looks like.

# First, make sure you can import the pypiper package

import os
import pypiper

# Create a PipelineManager instance (don't forget to name it!)
# This starts the pipeline.

pm = pypiper.PipelineManager(name="BASIC",
    outfolder="pipeline_output/")

# Now just build shell command strings, and use the run function
# to execute them in order. run needs 2 things: a command, and the
# target file you are creating.

# First, generate some random data

# specify target file:
tgt = "pipeline_output/test.out"

# build the command
cmd = "shuf -i 1-500000000 -n 10000000 > " + tgt

# and run with run().
pm.run(cmd, target=tgt)

# Now copy the data into a new file.
# first specify target file and build command:
tgt = "pipeline_output/copied.out"
cmd = "cp pipeline_output/test.out " + tgt
```

```
pm.run(cmd, target=tgt)

# You can also string multiple commands together, which will execute
# in order as a group to create the final target.
cmd1 = "sleep 5"
cmd2 = "touch pipeline_output/touched.out"
pm.run([cmd1, cmd2], target="pipeline_output/touched.out")

# A command without a target will run every time.
# Find the biggest line
cmd = "awk 'n < $0 {n=$0} END{print n}' pipeline_output/test.out"
pm.run(cmd, "lock.max")

# Use checkprint() to get the results of a command, and then use
# report_result() to print and log key-value pairs in the stats file:
last_entry = pm.checkprint("tail -n 1 pipeline_output/copied.out")
pm.report_result("last_entry", last_entry)


# Now, stop the pipeline to complete gracefully.
pm.stop_pipeline()

# Observe your outputs in the pipeline_output folder
# to see what you've created.
```

1.7.3 Advanced tutorial

Here we have a more advanced bioinformatics pipeline that adds some new concepts. This is a simple script that takes an input file and returns the file size and the number of sequencing reads in that file. This example uses a function from from the built-in *NGSTk toolkit*. In particular, this toolkit contains a few handy functions that make it easy for a pipeline to accept inputs of various types. So, this pipeline can count the number of reads from files in `BAM` format, or `fastq` format, or `fastq.gz` format. You can also use the same functions from NGSTk to develop a pipeline to do more complicated things, and handle input of any of these types.

First, grab this pipeline. Download count_reads.py, make it executable (`chmod 755 count_reads.py`), and then run it with `./count_reads.py`).

You can grab a few small data files in the microtest repository. Run a few of these files like this:

```
./count_reads.py -I ~/code/microtest/data/rrbs_PE_R1.fastq.gz -O $HOME -S sample1
./count_reads.py -I ~/code/microtest/data/rrbs_PE_fq_R1.fastq -O $HOME -S sample2
./count_reads.py -I ~/code/microtest/data/atac-seq_SE.bam -O $HOME -S sample3
```

This example is a documented vignette; so just read it and run it to get an idea of how things work.

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

"""
Counts reads.
"""

__author__ = "Nathan Sheffield"
__email__ = "nathan@code.databio.org"
__license__ = "GPL3"
__version__ = "0.1"
```

```
from argparse import ArgumentParser
import os, re
import sys
import subprocess
import yaml
import pypiper

parser = ArgumentParser(
    description="A pipeline to count the number of reads and file size. Accepts"
    " BAM, fastq, or fastq.gz files.")

# First, add standard arguments from Pypiper.
# groups="pypiper" will add all the arguments that pypiper uses,
# and adding "common" adds arguments for --input and --sample--name
# and "output_parent". You can read more about your options for standard
# arguments in the pypiper docs (section "command-line arguments")
parser = pypiper.add_pypiper_args(parser, groups=["pypiper", "common", "ngs"],
                                  args=["output-parent", "config"],
                                  required=['sample-name', 'output-parent'])

# Add any pipeline-specific arguments if you like here.

args = parser.parse_args()

if not args.input or not args.output_parent:
    parser.print_help()
    raise SystemExit

if args.single_or_paired == "paired":
    args.paired_end = True
else:
    args.paired_end = False

# args for `output_parent` and `sample_name` were added by the standard
# `add_pypiper_args` function.
# A good practice is to make an output folder for each sample, housed under
# the parent output folder, like this:
outfolder = os.path.abspath(os.path.join(args.output_parent, args.sample_name))

# Create a PipelineManager object and start the pipeline
pm = pypiper.PipelineManager(name="count",
                             outfolder=outfolder,
                             args=args)

# NGSTk is a "toolkit" that comes with pypiper, providing some functions
# for dealing with genome sequence data. You can read more about toolkits in the
# documentation

# Create a ngstk object
ngstk = pypiper.NGSTk(pm=pm)

raw_folder = os.path.join(outfolder, "raw/")
fastq_folder = os.path.join(outfolder, "fastq/")

# Merge/Link sample input and Fastq conversion
# These commands merge (if multiple) or link (if single) input files,
# then convert (if necessary, for bam, fastq, or gz format) files to fastq.
```

```
# We'll start with a timestamp that will provide a division for this section
# in the log file
pm.timestamp("### Merge/link and fastq conversion: ")

# Now we'll rely on 2 NGSTk functions that can handle inputs of various types
# and convert these to fastq files.

local_input_files = ngstk.merge_or_link(
                        [args.input, args.input2],
                        raw_folder,
                        args.sample_name)

cmd, out_fastq_pre, unaligned_fastq = ngstk.input_to_fastq(
                                            local_input_files,
                                            args.sample_name,
                                            args.paired_end,
                                            fastq_folder)


# Now we'll use another NGSTk function to grab the file size from the input files
#
pm.report_result("File_mb", ngstk.get_file_size(local_input_files))


# And then count the number of reads in the file

n_input_files = len(filter(bool, local_input_files))

raw_reads = sum([int(ngstk.count_reads(input_file, args.paired_end))
                 for input_file in local_input_files]) / n_input_files

# Finally, we use the report_result() function to print the output and
# log the key-value pair in the standard stats.tsv file
pm.report_result("Raw_reads", str(raw_reads))

# Cleanup
pm.stop_pipeline()
```

Once you've been through these simple tutorials, you may want to check out *real-world pipelines built using pypiper*.

1.8 Basic functions

Pypiper is simple, but powerful. You only need 3 functions to get started. `PipelineManager` can do:

start_pipeline([args, multi])	Initialize setup.
run(cmd[, target, lock_name, shell, nofail, ...])	The primary workhorse function of PipelineManager, this runs a command.
stop_pipeline([status])	Terminate the pipeline.

With that you can create a simple pipeline. You can click on each function to view the in-depth documentation for that function. There are quite a few optional parameters to the `run` function, which is where most of Pypiper's power comes from

When you've mastered the basics and are ready to get more powerful, add in a few new (optional) commands that make debugging and development easier:

timestamp([message, checkpoint, finished, ...])	Print message, time, and time elapsed, perhaps creating checkpoint.
report_result(key, value[, annotation])	Writes a string to self.pipeline_stats_file.
clean_add(regex[, conditional, manual])	Add files (or regexs) to a cleanup list, to delete when this pipeline completes successfully.
get_stat(key)	Returns a stat that was previously reported.

The complete documentation for these functions can be found in the *API*.

1.9 Command-line arguments

To take full advantage of Pypiper (make your pipeline recoverable, etc.), you may choose to add command-line options to your pipeline that pypiper understands. Pypiper uses the typical Python argparse module to define command-line arguments to your pipeline.

You can use an ArgumentParser as usual, adding whatever arguments you like. Then, you add Pypiper args to your parser with the function `add_pypiper_args()`, and pass the parser to your `PipelineManager`, like this:

```
import pypiper, os, argparse
parser = ArgumentParser(description='Write a short description here')

# add any custom args here
# e.g. parser.add_argument('--foo', help='foo help')

# once you've established all your custom arguments, we can add the default
# pypiper arguments to your parser like this:

parser = pypiper.add_pypiper_args(parser)

# Then, pass the args parsed along to the PipelineManger

args = parser.parse_args()

pipeline = pypiper.PipelineManager(name="my_pipeline", outfolder="out", \
                    args=args)
```

Once you've added pypiper arguments, your pipeline will then enable a few built-in arguments: `--recover`, `--follow`, and `--dirty`, for example. As a side bonus, all arguments (including any of your custom arguments) will be recorded in the log outputs.

That's the basics. But you can customize things for more efficiency using a simple set of pre-built args and groups of args in pypiper:

1.9.1 Customizing `add_pypiper_args()`

There are two ways to modulate the arguments added by `add_pypiper_args()` function: the `groups` argument, which lets you add argument groups; or the `args` argument, which lets you add arguments indvidually. By default, `add_pypiper_args()` add all arguments listed in the `pypiper` group. You may instead pass a list of one or more of these groups of arguments (to `groups`) or individual arguments (to `args`) to customize exactly the set of built-in options your pipeline implements.

For example, `parser.add_pypiper_args(parser, groups=['pypiper', 'common'])` will add all arguments listed under `pypiper` and `common` below:

1.9.2 Built-in arguments accessed with `add_pypiper_args()`

Individual arguments that are understood and used by pypiper:

- `-R, --recover`: for a failed pipeline run, start off at the last successful step.
- `-N, --new-start`: Just recreate everything, even if it exists.
- `-D, --dirty`: Disables automatic cleaning of temporary files, so all intermediate files will still exist after a pipeline run (either sucessful or failed). Useful for debugging a pipeline even if it succeeds.
- `-F, --follow`: Runs all `follow-functions`, regardless of whether the accompanying command is run.
- `-C, --config`: Pypiper pipeline config yaml file.

Individual arguments just provided for convenience and standardization:

- `-I, --input`: primary input file (e.g. read1)
- `-I2, --input2`: secondary input file (e.g. read2)
- `-O, --output-parent`: parent folder for pipeline results (the pipeline will use this as the parent directory for a folder named `sample-name`)
- `-P, --cores`: Number of cores to use
- `-M, --mem`: Amount of memory in megabytes
- `-G, --genome`: Reference genome assembly (e.g. `hg38`)
- `-Q, --simple-or-paired`: For sequencing data, is input single-end or paired-end?

1.9.3 Pre-built collections of arguments added via `groups`:

- pypiper: `recover`, `new-start`, `dirty`, `follow`
- common: `input`, `sample-name`
- config: `config`
- resource: `mem`, `cores`
- looper: `config`, `output-parent`, `mem`, `cores`
- ngs: `input`, `sample-name`, `input2`, `genome`, `single-or-paired`

1.9.4 Specifying required built-in arguments

If you're using the built-in arguments, you may want to module which are required and which are not. That way, you can piggyback on how `ArgumentParser` handles required arguments very nicely – if the user does not specify a required argument, the pipeline will automatically prompt with usage instructions.

By default, built-in arguments are not flagged as required, but you can pass a list of required built-ins to the `required` parameter, like `add_pypiper_args(parser, args=["sample-name"], required=["sample-name"])`.

1.9.5 Examples

```
import pypiper, os, argparse
parser = ArgumentParser(description='Write a short description here')

# add just arguments from group `pypiper`
parser = pypiper.add_pypiper_args(parser, groups=["pypiper"])

# add just arguments from group `common`
parser = pypiper.add_pypiper_args(parser, groups=["common"])

# add arguments from two groups
parser = pypiper.add_pypiper_args(parser, groups=["common", "resources"],
                                  required=["sample-name", "output-parent"])

# add individual argument
parser = pypiper.add_pypiper_args(parser, args=["genome"])

# add some groups and some individual arguments
parser = pypiper.add_pypiper_args(parser, args=["genome"], groups=["looper", "ngs"])
```

1.10 Pipeline config files

Optionally, you may choose to conform to our standard for parameterizing pipelines, which enables you to use some powerful features of the pypiper system.

If you write a pipeline config file in `yaml` format and name it the same thing as the pipeline (but ending in `.yaml` instead of `.py`), pypiper will automatically load and provide access to these configuration options, and make it possible to pass customized config files on the command line. This is very useful for tweaking a pipeline for a similar project with slightly different parameters, without having to re-write the pipeline.

It's easy: just load the `PipelineManager` with `args` (as described in *command-line arguments*), and you have access to the config file automatically in `pipeline.config`.

For example, in `myscript.py` you write:

```
parser = pypiper.add_pipeline_args(parser, args=["config"])
pipeline = pypiper.PipelineManager(name="my_pipeline", outfolder=outfolder, \
                                   args = parser)
```

And in the same folder, you include a `yaml` called `myscript.yaml`:

```
settings:
  setting1: True
  setting2: 15
```

Then you can access these settings automatically in your script using:

```
pipeline.config.settings.setting1
pipeline.config.settings.setting2
```

This `yaml` file is useful for any settings the pipeline needs that is not related to the input Sample (which should be passed on the command-line). By convention, for consistency across pipelines, we use sections called `tools`, `resources`, and `parameters`, but the developer has the freedom to add other sections/variables as needed.

Pipeline config files by default are named the same as the pipeline with the suffix `.yaml` and reside in the same directory as the pipeline code.

Example:

```
tools:
  # absolute paths to required tools
  java:  /home/user/.local/tools /home/user/.local/tools/java
  trimmomatic:  /home/user/.local/tools/trimmomatic.jar
  fastqc:  fastqc
  samtools:  samtools
  bsmap:  /home/user/.local/tools/bsmap
  split_reads:  /home/user/.local/tools/split_reads.py  # split_reads.py script;␣
↪distributed with this pipeline

resources:
  # paths to reference genomes, adapter files, and other required shared data
  resources: /data/groups/lab_bock/shared/resources
  genomes: /data/groups/lab_bock/shared/resources/genomes/
  adapters: /data/groups/lab_bock/shared/resources/adapters/

parameters:
  # parameters passed to bioinformatic tools, subclassed by tool

  trimmomatic:
    quality_encoding: "phred33"
    threads: 30
    illuminaclip:
      adapter_fasta: "/home/user/.local/tools/resources/cpgseq_adapter.fa"
      seed_mismatches: 2
      palindrome_clip_threshold: 40
      simple_clip_threshold: 7
    slidingwindow:
      window_size: 4
      required_quality: 15
    maxinfo:
      target_length: 17
      strictness: 0.5
    minlen:
      min_length: 17

  bsmap:
    seed_size: 12
    mismatches_allowed_for_background: 0.10
    mismatches_allowed_for_left_splitreads: 0.06
    mismatches_allowed_for_right_splitreads: 0.00
    equal_best_hits: 100
    quality_threshold: 15
    quality_encoding: 33
    max_number_of_Ns: 3
    processors: 8
    random_number_seed: 0
    map_to_strands: 0
```

1.11 The follow argument

The `PipelineManager.run` function has an optional argument named `follow` that is useful for checking or reporting results from a command. To the `follow` argument you must pass a python function (which may be either a defined function or a `lambda` function). These *follow functions* are then coupled to the command that is run; the follow function will be called by python **if and only if** the command is run.

Why is this useful? The major use cases are QC checks and reporting results. We use a folllow function to run a QC check to make sure processes did what we expect, and then to report that result to the `stats` file. We only need to check the result and report the statistic once, so it's best to put these kind of checks in a `follow` function. Often, you'd like to run a function to examine the result of a command, but you only want to run that once, *right after the command that produced the result*. For example, counting the number of lines in a file after producing it, or counting the number of reads that aligned right after an alignment step. You want the counting process coupled to the alignment process, and don't need to re-run the counting every time you restart the pipeline. Because pypiper is smart, it will not re-run the alignment once it has been run; so there is no need to re-count the result on every pipeline run!

Follow functions let you avoid running unnecessary processes repeatedly in the event that you restart your pipeline multiple times (for instance, while debugging later steps in the pipeline).

1.12 Reporting statistics

One of the most useful features of pypiper is the `report_result()` function. This function provides a way to record small-scale results, like summary statistics. It standardizes the output so that universal tools can be built to process all the pipeline results from any pipeline, because the results are all reported in the same way.

When you call `pm.report_result(key, value)`, pypiper simply writes the key-value pair to a `tsv` file (`stats.tsv`) in the pipeline output folder. These `stats.tsv` files can then later be read and aggregated systematically by other tools, such as `looper summarize`.

1.12.1 Re-using previously reported results

We frequently want to use the `report_result` capability in `follow` functions. It's a convenient place to do something like count or assess the result of a long-running command, and then report some summary statistic on it. One potential hangup with this strategy is dealing with secondary results after a pipeline is interrupted and restarted. By secondary result, I mean one that requires knowing the value of an earlier result. For example, if you want to compute the **percentage of reads that aligned**, you need to first know the **total reads** – but what if your pipeline got interrupted and calculation of **total reads** happened in an earlier pipeline run?

To solve this issue, Pypiper has a neat function called `get_stat` that lets you retrieve any value you've reported with `report_result` so you could use it to calculate statistics elsewhere in the pipeline. It will retrieve this either from memory, if the calculation of that result happened during the current pipeline run, or from the `stats.tsv` file, if the result was reported by an earlier run (or even another pipeline). So you could, in theory, calculate statistics based on results across pipelines.

An example for how to use this is how we handle calculating the alignment rate in an NGS pipeline:

```
x = myngstk.count_mapped_reads(bamfile, args.paired_end)
pm.report_result("Aligned_reads", x)
rr = float(pm.get_stat("Raw_reads"))
pm.report_result("Alignment_rate", round((rr * 100 / float(x), 3))
```

Here, we use `get_stat` to grab a result that we reported previously (with `report_result`), when we counted the number of `Raw_reads` (earlier in the pipeline). We need this after the alignment to calculate the alignment rate. Later, now that we've reported `Alignment_rate`, you could harvest this stat again for use with `pm.get_stat("Alignment_rate")`. This is useful because you could put this block of code in a `follow` statement so it may not be executed, but you can still grab a reported result like this even if the execution happened outside of the current pipeline run; you'd only have to do the calculation once.

1.13 Best practices

Here are some guidelines for how you can design the most effective pipelines.

- **Compartmentalize output into folders.** In your output, keep pipeline steps separate by organizing output into subfolders.
- **Use git for versioning.** If you develop your pipeline in a git repository, Pypiper will automatically record the commit hash when you run a pipeline, making it easy to figure out **exactly** what code version you ran.
- **Record stats as you go.** In other words, don't do all your stats (`report_result()`) and QC at the end, do it along the way. This makes it easy for you to monitor pipeline performance, and couples stats with how far the pipeline makes it, so you could make use of a partially completed (or even ultimately failed) pipelines.
- **Use looper args.** Even if you're not using looper at first, use `looper_args` and your pipeline will be looper-ready when it comes time to run 500 samples.
- **Use NGSTk early on.** NGSTk has lots of useful functions that you will probably need. We've worked hard to make these robust and universal. For example, using NGSTk, you can easily make your pipeline take flexible input formats (fastq or bam). Right now you may always have the same input type (fastq, for example), but later you may want your pipeline to be able to work from `bam` files. We've already written simple functions to handle single or multiple bam or fastq inputs; just use this infrastructure (in NGSTk) instead of writing your own, and you'll save yourself future headaches.
- **Make some important parameters in the pipeline config, instead of hardcoding them** Pypiper makes it painfully easy to use a config file to make your pipeline configurable. Typically you'll start by hard-coding in those parameters in your pipeline steps. But you can select a few important parameters and make them customizable in the pipeline config. Start from the very beginning by making a `yaml` pipeline config file. See an example of a *pipeline config file*

1.14 Advanced options

1.14.1 Python process types: shell vs direct

Since Pypiper runs all your commands from within python (using the *subprocess* python module), it's nice to be aware of the two types of processes that *subprocess* allows: **direct processes** and **shell processes**.

By default, Pypiper will try to guess what kind of process you want, so for most pipelines, it's probably not necessary to understand the details in this section. However, how you write your commands has some implications for memory tracking, and advanced pipeline authors may want to control the process types that Pypiper uses, so this section covers how these subprocesses work.

Direct process: A direct process is one that Python executes directly, from within python. Python retains control over the process completely. Wherever possible, you should use a direct subprocess – this has the advantage of enabling Python to monitor the memory use of the subprocess, because Python retains control over it. This the preferable way of running subprocesses in Python. The disadvantage of direct subprocesses is that you may not use shell-specific operators in a direct subprocess. For instance, if you use an asterisk (`*`) for wildcard expansion, or a bracket (`>`) for output redirection, or a pipe (`|`) to link processes – these are commands understood by a shell like Bash, and thus, cannot be run as direct subprocesses in Python.

Shell process: In a shell process, Python first spawns a shell, and then runs the command in that shell. The spawned shell is then controlled by Python, but processes done by the shell are not. This allows you to use shell operators (`*`, `|`, `>`), but at the cost of the ability to monitor memory high water mark, because Python does not have direct control over subprocesses run inside a subshell.

You **must** use a shell process if you are using shell operators in your command. You can force Pypiper to use one or the other by specifying `shell=True` or `shell=False` to the `run` function. By default Pypiper will try to guess: if your command contains any of the shell process characters (`*`, `|`, or `>`), it will be run in a shell. Otherwise, it will be run as a direct subprocess. So for most purposes, you do not need to worry about this at all, but you may want to write your commands to minimize shell operators if you are interested in the memory monitoring features of Pypiper.

1.15 What are toolkits?

Pypiper provides a limited set of functions that are all quite generic; they simple accept command-line commands and run them. You could use this to produce a pipeline in any domain.

To add to this, you may be interested in building some of your own convenience functions that make it easier for you to piece together commands. It's really easy to create your own library of python functions by creating a python package. Then, you just need to import your package in your pipeline script and make use of the common functions.

We refer to this type of package as a "toolkit", and Pypiper also includes an built-in toolkit called NGSTk (next-generation sequencing toolkit). NGSTk simply provides some convenient helper functions to create common shell commands, like converting from file formats (_e.g._ `bam_to_fastq()`), merging files (_e.g._ `merge_bams()`), counting reads, etc. These make it faster to design bioinformatics pipelines in Pypiper, but are entirely optional. Contributions of additional toolkits or functions to an existing toolkit are welcome.

More details about the NGSTk toolkit can be found on the next page under *NGSTk*.

1.16 NGSTk

An optional feature of pypiper is the accompanying toolkits, such as the next-gen sequencing toolkit, NGSTk, which simply provides some convenient helper functions to create common commands, like converting from file formats (*e.g.* bam to fastq), merging files (*e.g.* merge_bams), counting reads, etc. These make it faster to design bioinformatics pipelines in Pypiper, but are entirely optional.

Example:

```
import pypiper
pm = pypiper.PipelineManager(..., args = args)

# Create a ngstk object (pass the PipelineManager as an argument)
ngstk = pypiper.NGSTk(pm = pm)

# Now you use use ngstk functions
ngstk.index_bam("sample.bam")
```

A list of available functions can be found in the *API* or in the source code for NGSTk.

Contributions of additional toolkits or functions in an existing toolkit are welcome.

1.17 API

1.17.1 pypiper.PipelineManager

Pypiper is a python module with two components: 1) the PipelineManager class, and 2) other toolkits (currently just NGSTk) with functions for more specific pipeline use-cases. The PipelineManager class can be used to create a procedural pipeline in python.

class pypiper.manager.**PipelineManager**(*name*, *outfolder*, *version=None*, *args=None*, *multi=False*, *manual_clean=False*, *recover=False*, *fresh=False*, *force_follow=False*, *cores=1*, *mem='1000'*, *config_file=None*, *output_parent=None*, *overwrite_checkpoints=False*, ***kwargs*)

Base class for instantiating a PipelineManager object, the main class of Pypiper.

Parameters

- **name** (*str*) – Choose a name for your pipeline; it's used to name the output files, flags, etc.
- **outfolder** (*str*) – Folder in which to store the results.
- **args** (*argparse.Namespace*) – Optional args object from ArgumentParser; Pypiper will simply record these arguments from your script
- **multi** (*bool*) – Enables running multiple pipelines in one script or for interactive use. It simply disables the tee of the output, so you won't get output logged to a file.
- **manual_clean** (*bool*) – Overrides the pipeline's clean_add() manual parameters, to *never* clean up intermediate files automatically. Useful for debugging; all cleanup files are added to manual cleanup script.
- **recover** (*bool*) – Specify recover mode, to overwrite lock files. If pypiper encounters a locked target, it will ignore the lock and recompute this step. Useful to restart a failed pipeline.
- **fresh** (*bool*) – NOT IMPLEMENTED
- **force_follow** (*bool*) – Force run all follow functions even if the preceding command is not run. By default, following functions are only run if the preceding command is run.
- **cores** (*int*) – number of processors to use, default 1
- **mem** (*str*) – amount of memory to use, in Mb
- **config_file** (*str*) – path to pipeline configuration file, optional
- **output_parent** (*str*) – path to folder in which output folder will live
- **overwrite_checkpoints** (*bool*) – Whether to override the stage-skipping logic provided by the checkpointing system. This is useful if the calls to this manager's run() method will be coming from a class that implements pypiper.Pipeline, as such a class will handle checkpointing logic automatically, and will set this to True to protect from a case in which a restart begins upstream of a stage for which a checkpoint file already exists, but that depends on the upstream stage and thus should be rerun if it's "parent" is rerun.

Raises **TypeError** – if start or stop point(s) are provided both directly and via args namespace, or if both stopping types (exclusive/prospective and inclusive/retrospective) are provided.

atexit_register(**args*)

Convenience alias to register exit functions without having to import atexit in the pipeline.

callprint(*cmd*, *shell='guess'*, *nofail=False*, *container=None*, *lock_name=None*, *errmsg=None*)

Prints the command, and then executes it, then prints the memory use and return code of the command.

Uses python's subprocess.Popen() to execute the given command. The shell argument is simply passed along to Popen(). You should use shell=False (default) where possible, because this enables memory profiling. You should use shell=True if you require shell functions like redirects (>) or pipes (|), but this will prevent the script from monitoring memory use.

cmd can also be a series (a dict object) of multiple commands, which will be run in succession.

Parameters

- **cmd** (*str or list*) – Bash command(s) to be run.
- **shell** (*bool*) – If command is required to be run in its own shell. Optional. Default: "guess", which will make a best guess on whether it should run in a shell or not, based on presence of shell utils, like asterisks, pipes, or output redirects. Force one way or another by specifying True or False
- **nofail** (*bool*) – Should the pipeline bail on a nonzero return from a process? Default: False Nofail can be used to implement non-essential parts of the pipeline; if these processes fail, they will not cause the pipeline to bail out.
- **container** – Named Docker container in which to execute.
- **container** – str
- **lock_name** (*str*) – Name of the relevant lock file.
- **errmsg** (*str*) – Message to print if there's an error.

checkprint(*cmd*, *shell='guess'*, *nofail=False*, *errmsg=None*)

Just like callprint, but checks output – so you can get a variable in python corresponding to the return value of the command you call. This is equivalent to running subprocess.check_output() instead of subprocess.call(). :param cmd: Bash command(s) to be run. :type cmd: str or list :param shell: If command requires should be run in its own shell. Optional. Default: "guess" – *run()* will try to guess if the command should be run in a shell (based on the presence of a pipe (|) or redirect (>), To force a process to run as a direct subprocess, set *shell* to False; to force a shell, set True. :type shell: bool :param nofail: Should the pipeline bail on a nonzero return from a process? Default: False Nofail can be used to implement non-essential parts of the pipeline; if these processes fail, they will not cause the pipeline to bail out. :type nofail: bool :param errmsg: Message to print if there's an error. :type errmsg: str

clean_add(*regex*, *conditional=False*, *manual=False*)

Add files (or regexs) to a cleanup list, to delete when this pipeline completes successfully. When making a call with run that produces intermediate files that should be deleted after the pipeline completes, you flag these files for deletion with this command. Files added with clean_add will only be deleted upon success of the pipeline.

Parameters

- **regex** (*str*) – A unix-style regular expression that matches files to delete (can also be a file name).
- **conditional** (*bool*) – True means the files will only be deleted if no other pipelines are currently running; otherwise they are added to a manual cleanup script called {pipeline_name}_cleanup.sh
- **manual** (*bool*) – True means the files will just be added to a manual cleanup script.

complete()

Stop a completely finished pipeline.

completed

Is the managed pipeline in a completed state?

Return bool Whether the managed pipeline is in a completed state.

fail_pipeline(*e*, *dynamic_recover=False*)

If the pipeline does not complete, this function will stop the pipeline gracefully. It sets the status flag to failed and skips the normal success completion procedure.

Parameters

- **e** (*Exception*) – Exception to raise.
- **dynamic_recover** (*bool*) – Whether to recover e.g. for job termination.

failed
Is the managed pipeline in a failed state?

Return bool Whether the managed pipeline is in a failed state.

flag_file_path(*status=None*)
Create path to flag file based on indicated or current status.

Internal variables used are the pipeline name and the designated pipeline output folder path.

Parameters **status** (*str*) – flag file type to create, default to current status

Return str path to flag file of indicated or current status.

get_stat(*key*)
Returns a stat that was previously reported. This is necessary for reporting new stats that are derived from two stats, one of which may have been reported by an earlier run. For example, if you first use report_result to report (number of trimmed reads), and then in a later stage want to report alignment rate, then this second stat (alignment rate) will require knowing the first stat (number of trimmed reads); however, that may not have been calculated in the current pipeline run, so we must retrieve it from the stats.tsv output file. This command will retrieve such previously reported stats if they were not already calculated in the current pipeline run. :param key: key of stat to retrieve

halt(*checkpoint=None*, *finished=False*, *raise_error=True*)
Stop the pipeline before completion point.

Parameters

- **checkpoint** (*str*) – Name of stage just reached or just completed.
- **finished** (*bool*) – Whether the indicated stage was just finished (True), or just reached (False)
- **raise_error** (*bool*) – Whether to raise an exception to truly halt execution.

halted
Is the managed pipeline in a paused/halted state?

Return bool Whether the managed pipeline is in a paused/halted state.

has_exit_status
Has the managed pipeline been safely stopped?

Return bool Whether the managed pipeline's status indicates that it has been safely stopped.

is_running
Is the managed pipeline running?

Return bool Whether the managed pipeline is running.

make_sure_path_exists(*path*)
Creates all directories in a path if it does not exist.

Parameters **path** (*str*) – Path to create.

Raises **Exception** – if the path creation attempt hits an error with a code indicating a cause other than pre-existence.

report_figure(*key*, *filename*, *annotation=None*)
Writes a string to self.pipeline_figures_file.

Parameters

- **key** (*str*) – name (key) of the figure
- **filename** (*str*) – relative path to the file (relative to parent output dir)
- **annotation** (*str*) – By default, the figures will be annotated with the pipeline name, so you can tell which pipeline records which figures. If you want, you can change this.

report_result(*key*, *value*, *annotation=None*)
Writes a string to self.pipeline_stats_file.

Parameters

- **key** (*str*) – name (key) of the stat
- **annotation** (*str*) – By default, the stats will be annotated with the pipeline name, so you can tell which pipeline records which stats. If you want, you can change this; use annotation='shared' if you need the stat to be used by another pipeline (using get_stat()).

run(*cmd*, *target=None*, *lock_name=None*, *shell='guess'*, *nofail=False*, *errmsg=None*, *clean=False*, *follow=None*, *container=None*)
The primary workhorse function of PipelineManager, this runs a command.

This is the command execution function, which enforces race-free file-locking, enables restartability, and multiple pipelines can produce/use the same files. The function will wait for the file lock if it exists, and not produce new output (by default) if the target output file already exists. If the output is to be created, it will first create a lock file to prevent other calls to run (for example, in parallel pipelines) from touching the file while it is being created. It also records the memory of the process and provides some logging output.

Parameters

- **cmd** (*str or list*) – Shell command(s) to be run.
- **target** (*str or None*) – Output file to be produced. Optional.
- **lock_name** (*str or None*) – Name of lock file. Optional.
- **shell** (*bool*) – If command requires should be run in its own shell. Optional. Default: "guess" –will try to determine whether the command requires a shell.
- **nofail** (*bool*) – Whether the pipeline proceed past a nonzero return from a process, default False; nofail can be used to implement non-essential parts of the pipeline; if a 'nofail' command fails, the pipeline is free to continue execution.
- **errmsg** (*str*) – Message to print if there's an error.
- **clean** (*bool*) – True means the target file will be automatically added to a auto cleanup list. Optional.
- **follow** (*callable*) – Function to call after executing (each) command.
- **container** (*str*) – Name for Docker container in which to run commands.

Returns Return code of process. If a list of commands is passed, this is the maximum of all return codes for all commands.

Return type int

set_status_flag(*status*)
Configure state and files on disk to match current processing status.

Parameters **status** (*str*) – Name of new status designation for pipeline.

start_pipeline(*args=None*, *multi=False*)
Initialize setup. Do some setup, like tee output, print some diagnostics, create temp files. You provide only the output directory (used for pipeline stats, log, and status flag files).

stop_pipeline(*status='completed'*)
Terminate the pipeline.

This is the "healthy" pipeline completion function. The normal pipeline completion function, to be run by the pipeline at the end of the script. It sets status flag to completed and records some time and memory statistics to the log file.

time_elapsed(*time_since*)
Returns the number of seconds that have elapsed since the time_since parameter.

Parameters **time_since** (*float*) – Time as a float given by time.time().

timestamp(*message=''*, *checkpoint=None*, *finished=False*, *raise_error=True*)
Print message, time, and time elapsed, perhaps creating checkpoint.

This prints your given message, along with the current time, and time elapsed since the previous timestamp() call. If you specify a HEADING by beginning the message with "###", it surrounds the message with newlines for easier readability in the log file. If a checkpoint is designated, an empty file is created corresponding to the name given. Depending on how this manager's been configured, the value of the checkpoint, and whether this timestamp indicates initiation or completion of a group of pipeline steps, this call may stop the pipeline's execution.

Parameters

- **message** (*str*) – Message to timestamp.
- **checkpoint** (*str, optional*) – Name of checkpoint; this tends to be something that reflects the processing logic about to be or having just been completed. Provision of an argument to this parameter means that a checkpoint file will be created, facilitating arbitrary starting and stopping point for the pipeline as desired.
- **finished** (*bool, default False*) – Whether this call represents the completion of a conceptual unit of a pipeline's processing

:param raise_error [Whether to raise exception if] checkpoint or current state indicates that a halt should occur.

1.17.2 pypiper.NGSTk

class pypiper.ngstk.**NGSTk**(*config_file=None*, *pm=None*)
Class to hold functions to build command strings used during pipeline runs. Object can be instantiated with a string of a path to a yaml *pipeline config file*. Since NGSTk inherits from *AttributeDict*, the passed config file and its elements will be accessible through the NGSTk object as attributes under *config* (e.g. *NGSTk.tools.java*). In case no *config_file* argument is passed, all commands will be returned assuming the tool is in the user's $PATH.

Parameters

- **config_file** (*str*) – Path to pipeline yaml config file (optional).
- **pm** (*pypiper.PipelineManager*) – A PipelineManager with which to associate this toolkit instance; that is, essentially a source from which to grab paths to tools, resources, etc.

Example from pypiper.ngstk import NGSTk as tk tk = NGSTk() tk.samtools_index("sample.bam") # returns: samtools index sample.bam

Using a configuration file (custom executable location): from pypiper.ngstk import NGSTk tk = NGSTk("pipeline_config_file.yaml") tk.samtools_index("sample.bam") # returns: /home/.local/samtools/bin/samtools index sample.bam

bam2fastq(*input_bam*, *output_fastq*, *output_fastq2=None*, *unpaired_fastq=None*)
Create command to convert BAM(s) to FASTQ(s).

Parameters

- **input_bam** (*str*) – Path to sequencing reads file to convert
- **output_fastq** – Path to FASTQ to write
- **output_fastq2** – Path to (R2) FASTQ to write
- **unpaired_fastq** – Path to unpaired FASTQ to write

Return str Command to convert BAM(s) to FASTQ(s)

bam_conversions(*bam_file*, *depth=True*)
Sort and index bam files for later use. :param depth: also calculate coverage over each position

bam_to_bigwig(*input_bam*, *output_bigwig*, *genome_sizes*, *genome*, *tagmented=False*, *normalize=False*, *norm_factor=1000*)
Convert a BAM file to a bigWig file.

Parameters

- **input_bam** (*str*) – path to BAM file to convert
- **output_bigwig** (*str*) – path to which to write file in bigwig format
- **genome_sizes** (*str*) – path to file with chromosome size information
- **genome** (*str*) – name of genomic assembly
- **tagmented** (*bool*) – flag related to read-generating protocol
- **normalize** (*bool*) – whether to normalize coverage
- **norm_factor** (*int*) – number of bases to use for normalization

Return list[str] sequence of commands to execute

bam_to_fastq(*bam_file*, *out_fastq_pre*, *paired_end*)
Build command to convert BAM file to FASTQ file(s) (R1/R2).

Parameters

- **bam_file** (*str*) – path to BAM file with sequencing reads
- **out_fastq_pre** (*str*) – path prefix for output FASTQ file(s)
- **paired_end** (*bool*) – whether the given file contains paired-end or single-end sequencing reads

Return str file conversion command, ready to run

bam_to_fastq_awk(*bam_file*, *out_fastq_pre*, *paired_end*)
This converts bam file to fastq files, but using awk. As of 2016, this is much faster than the standard way of doing this using Picard, and also much faster than the bedtools implementation as well; however, it does no sanity checks and assumes the reads (for paired data) are all paired (no singletons), in the correct order.

bam_to_fastq_bedtools(*bam_file*, *out_fastq_pre*, *paired_end*)
Converts bam to fastq; A version using bedtools

calc_frip(*input_bam*, *input_bed*, *threads=4*)
Calculate fraction of reads in peaks.

A file of with a pool of sequencing reads and a file with peak call regions define the operation that will be performed. Thread count for samtools can be specified as well.

Parameters

- **input_bam** (*str*) – sequencing reads file
- **input_bed** (*str*) – file with called peak regions
- **threads** (*int*) – number of threads samtools may use

Returns fraction of reads in peaks defined in the given peaks file

Return type float

check_command(*command*)
Check if command can be called.

check_fastq(*input_files*, *output_files*, *paired_end*)
Returns a follow sanity-check function to be run after a fastq conversion. Run following a command that will produce the fastq files.

This function will make sure any input files have the same number of reads as the output files.

check_trim(*trimmed_fastq*, *paired_end*, *trimmed_fastq_R2=None*, *fastqc_folder=None*)
Build function to evaluate read trimming, and optionally run fastqc.

This is useful to construct an argument for the 'follow' parameter of a PipelineManager's 'run' method.

Parameters

- **trimmed_fastq** (*str*) – Path to trimmed reads file.
- **paired_end** (*bool*) – Whether the processing is being done with paired-end sequencing data.
- **trimmed_fastq_R2** (*str*) – Path to read 2 file for the paired-end case.
- **fastqc_folder** – Path to folder within which to place fastqc output files; if unspecified, fastqc will not be run.

Returns Function to evaluate read trimming and possibly run fastqc.

Return type callable

count_fail_reads(*file_name*, *paired_end*)
Counts the number of reads that failed platform/vendor quality checks. :param paired_end: This parameter is ignored; samtools automatically correctly responds depending on the data in the bamfile. We leave the option here just for consistency, since all the other counting functions require the parameter. This makes it easier to swap counting functions during pipeline development.

count_flag_reads(*file_name*, *flag*, *paired_end*)
Counts the number of reads with the specified flag.

Parameters

- **file_name** (*str*) – name of reads file
- **flag** (*str*) – sam flag value to be read
- **paired_end** – This parameter is ignored; samtools automatically correctly responds depending

on the data in the bamfile. We leave the option here just for consistency, since all the other counting functions require the parameter. This makes it easier to swap counting functions during pipeline development. :type paired_end: bool

count_lines(*file_name*)
Uses the command-line utility wc to count the number of lines in a file.

Parameters **file_name** (*str*) – name of file whose lines are to be counted

count_lines_zip(*file_name*)
Uses the command-line utility wc to count the number of lines in a file. For compressed files. :param file: file_name

count_mapped_reads(*file_name*, *paired_end*)
Mapped_reads are not in fastq format, so this one doesn't need to accommodate fastq, and therefore, doesn't require a paired-end parameter because it only uses samtools view. Therefore, it's ok that it has a default parameter, since this is discarded.

Parameters

- **file_name** (*str*) – File for which to count mapped reads.
- **paired_end** – This parameter is ignored; samtools automatically correctly responds depending

on the data in the bamfile. We leave the option here just for consistency, since all the other counting functions require the parameter. This makes it easier to swap counting functions during pipeline development. :type paired_end: bool :return: Either return code from samtools view command, or -1 to

indicate an error state.

Return type int

count_multimapping_reads(*file_name*, *paired_end*)
Counts the number of reads that mapped to multiple locations. Warning: currently, if the alignment software includes the reads at multiple locations, this function will count those more than once. This function is for software that randomly assigns, but flags reads as multimappers.

Parameters

- **file_name** (*str*) – name of reads file
- **paired_end** – This parameter is ignored; samtools automatically correctly responds depending

on the data in the bamfile. We leave the option here just for consistency, since all the other counting functions require the parameter. This makes it easier to swap counting functions during pipeline development.

count_reads(*file_name*, *paired_end*)
Count reads in a file.

Paired-end reads count as 2 in this function. For paired-end reads, this function assumes that the reads are split into 2 files, so it divides line count by 2 instead of 4. This will thus give an incorrect result if your paired-end fastq files are in only a single file (you must divide by 2 again).

Parameters

- **file_name** (*str*) – Name/path of file whose reads are to be counted.
- **paired_end** (*bool*) – Whether the file contains paired-end reads.

count_unique_mapped_reads(*file_name*, *paired_end*)
For a bam or sam file with paired or or single-end reads, returns the number of mapped reads, counting each read only once, even if it appears mapped at multiple locations.

Parameters

- **file_name** (*str*) – name of reads file
- **paired_end** (*bool*) – True/False paired end data

Returns Number of uniquely mapped reads.

Return type int

count_unique_reads (*file_name*, *paired_end*)
Sometimes alignment software puts multiple locations for a single read; if you just count those reads, you will get an inaccurate count. This is _not_ the same as multimapping reads, which may or may not be actually duplicated in the bam file (depending on the alignment software). This function counts each read only once. This accounts for paired end or not for free because pairs have the same read name. In this function, a paired-end read would count as 2 reads.

count_uniquelymapping_reads (*file_name*, *paired_end*)
Counts the number of reads that mapped to a unique position.

Parameters

- **file_name** (*str*) – name of reads file
- **paired_end** (*bool*) – This parameter is ignored.

fastqc (*file*, *output_dir*)
Create command to run fastqc on a BAM file (or FASTQ file, right?)

Parameters

- **file** (*str*) – Path to file with sequencing reads
- **output_dir** (*str*) – Path to folder in which to place output

Return str Command with which to run fastqc

fastqc_rename (*input_bam*, *output_dir*, *sample_name*)
Create pair of commands to run fastqc and organize files.

The first command returned is the one that actually runs fastqc when it's executed; the second moves the output files to the output folder for the sample indicated.

Parameters

- **input_bam** (*str*) – Path to file for which to run fastqc.
- **output_dir** (*str*) – Path to folder in which fastqc output will be written, and within which the sample's output folder lives.
- **sample_name** (*str*) – Sample name, which determines subfolder within output_dir for the fastqc files.

Returns Pair of commands, to run fastqc and then move the files to their intended destination based on sample name.

Return type list[str]

filter_reads (*input_bam*, *output_bam*, *metrics_file*, *paired=False*, *cpus=16*, *Q=30*)
Remove duplicates, filter for >Q, remove multiple mapping reads. For paired-end reads, keep only proper pairs.

get_chrs_from_bam (*file_name*)
Uses samtools to grab the chromosomes from the header that are contained in this bam file.

get_file_size (*filenames*)
Get size of all files in string (space-separated) in megabytes (Mb).

Parameters **filenames** (*str*) – a space-separated string of filenames

get_frip(*sample*)
Calculates the fraction of reads in peaks for a given sample. :param sample: A Sample object with the "peaks" attribute. :type sample: pipelines.Sample

get_input_ext(*input_file*)
Get the extension of the input_file. Assumes you're using either .bam or .fastq/.fq or .fastq.gz/.fq.gz.

get_mitochondrial_reads(*bam_file*, *output*, *cpus=4*)

get_peak_number(*sample*)
Counts number of peaks from a sample's peak file. :param sample: A Sample object with the "peaks" attribute. :type sample: pipelines.Sample

get_read_type(*bam_file*, *n=10*)
Gets the read type (single, paired) and length of bam file. :param bam_file: Bam file to determine read attributes. :type bam_file: str :param n: Number of lines to read from bam file. :type n: int :returns: tuple of (read_type=string, read_length=int). :rtype: tuple

input_to_fastq(*input_file*, *sample_name*, *paired_end*, *fastq_folder*, *output_file=None*, *multiclass=False*)
Builds a command to convert input file to fastq, for various inputs.

Takes either .bam, .fastq.gz, or .fastq input and returns commands that will create the .fastq file, regardless of input type. This is useful to made your pipeline easily accept any of these input types seamlessly, standardizing you to the fastq which is still the most common format for adapter trimmers, etc.

It will place the output fastq file in given *fastq_folder*.

Parameters **input_file** (*string*) – filename of the input you want to convert to fastq

Returns A command (to be run with PipelineManager) that will ensure your fastq file exists.

macs2_call_peaks(*treatment_bams*, *output_dir*, *sample_name*, *genome*, *control_bams=None*, *broad=False*, *paired=False*, *pvalue=None*, *qvalue=None*, *include_significance=None*)
Use MACS2 to call peaks.

Parameters

- **treatment_bams** (*str | Iterable[str]*) – Paths to files with data to regard as treatment.
- **output_dir** (*str*) – Path to output folder.
- **sample_name** (*str*) – Name for the sample involved.
- **genome** (*str*) – Name of the genome assembly to use.
- **control_bams** (*str | Iterable[str]*) – Paths to files with data to regard as control
- **broad** (*bool*) – Whether to do broad peak calling.
- **paired** (*bool*) – Whether reads are paired-end
- **pvalue** (*NoneType | float*) – Statistical significance measure to pass as –pvalue to peak calling with MACS
- **qvalue** (*NoneType | float*) – Statistical significance measure to pass as –qvalue to peak calling with MACS
- **include_significance** (*NoneType | bool*) – Whether to pass a statistical significance argument to peak calling with MACS; if omitted, this will be True if the peak

calling is broad or if either p-value or q-value is specified; default significance specification is a p-value of 0.001 if a significance is to be specified but no value is provided for p-value or q-value.

Returns Command to run.

Return type str

make_dir(*path*)
Forge path to directory, creating intermediates as needed.

Parameters **path** (*str*) – Path to create.

make_sure_path_exists(*path*)
Alias for make_dir

merge_bams(*input_bams*, *merged_bam*, *in_sorted='TRUE'*, *tmp_dir=None*)
Combine multiple files into one.

The tmp_dir parameter is important because on poorly configured systems, the default can sometimes fill up.

Parameters

- **input_bams** (*Iterable[str]*) – Paths to files to combine
- **merged_bam** (*str*) – Path to which to write combined result.
- **in_sorted** (*bool | str*) – Whether the inputs are sorted
- **tmp_dir** (*str*) – Path to temporary directory.

merge_fastq(*inputs*, *output*, *run=False*, *remove_inputs=False*)
Merge FASTQ files (zipped or not) into one.

Parameters

- **inputs** (*Iterable[str]*) – Collection of paths to files to merge.
- **output** (*str*) – Path to single output file.
- **run** (*bool*) – Whether to run the command.
- **remove_inputs** (*bool*) – Whether to keep the original files.

Return NoneType | str Null if running the command, otherwise the command itself

Raises **ValueError** – Raise ValueError if the call is such that inputs are to be deleted but command is not run.

merge_or_link(*input_args*, *raw_folder*, *local_base='sample'*)
This function standardizes various input possibilities by converting either .bam, .fastq, or .fastq.gz files into a local file; merging those if multiple files given.

Parameters

- **input_args** (*list*) – This is a list of arguments, each one is a class of inputs (which can in turn be a string or a list). Typically, input_args is a list with 2 elements: first a list of read1 files; second an (optional!) list of read2 files.
- **raw_folder** (*str*) – Name/path of folder for the merge/link.
- **local_base** (*str*) – Usually the sample name. This (plus file extension) will be the name of the local file linked (or merged) by this function.

parse_bowtie_stats(*stats_file*)
Parses Bowtie2 stats file, returns series with values. :param stats_file: Bowtie2 output file with alignment statistics. :type stats_file: str

parse_duplicate_stats(*stats_file*)
Parses sambamba markdup output, returns series with values. :param stats_file: sambamba output file with duplicate statistics. :type stats_file: str

parse_qc(*qc_file*)
Parse phantompeakqualtools (spp) QC table and return quality metrics.

Parameters **qc_file** (*str*) – Path to phantompeakqualtools output file, which contains sample quality measurements.

plot_atacseq_insert_sizes(*bam*, *plot*, *output_csv*, *max_insert=1500*, *smallest_insert=30*)
Heavy inspiration from here: https://github.com/dbrg77/ATAC/blob/master/ATAC_seq_read_length_curve_fitting.ipynb

run_spp(*input_bam*, *output*, *plot*, *cpus*)
Run the SPP read peak analysis tool.

Parameters

- **input_bam** (*str*) – Path to reads file
- **output** (*str*) – Path to output file
- **plot** (*str*) – Path to plot file
- **cpus** (*int*) – Number of processors to use

Returns Command with which to run SPP

Return type str

sam_conversions(*sam_file*, *depth=True*)
Convert sam files to bam files, then sort and index them for later use. :param depth: also calculate coverage over each position

samtools_index(*bam_file*)
Index a bam file.

samtools_view(*file_name*, *param*, *postpend=''*)
Run samtools view, with flexible parameters and post-processing.

This is used internally to implement the various count_reads functions.

Parameters

- **file_name** (*str*) – file_name
- **param** (*str*) – String of parameters to pass to samtools view
- **postpend** (*str*) – String to append to the samtools command; useful to add cut, sort, wc operations to the samtools view output.

skewer(*input_fastq1*, *output_prefix*, *output_fastq1*, *log*, *cpus*, *adapters*, *input_fastq2=None*, *output_fastq2=None*)
Create commands with which to run skewer.

Parameters

- **input_fastq1** (*str*) – Path to input (read 1) FASTQ file
- **output_prefix** (*str*) – Prefix for output FASTQ file names
- **output_fastq1** (*str*) – Path to (read 1) output FASTQ file

- **log** (*str*) – Path to file to which to write logging information
- **| str cpus** (*int*) – Number of processing cores to allow
- **adapters** (*str*) – Path to file with sequencing adapters
- **input_fastq2** (*str*) – Path to read 2 input FASTQ file
- **output_fastq2** (*str*) – Path to read 2 output FASTQ file

Return list[str] Sequence of commands to run to trim reads with skewer and rename files as desired.

spp_call_peaks(*treatment_bam*, *control_bam*, *treatment_name*, *control_name*, *output_dir*, *broad*, *cpus*, *qvalue=None*)

Build command for R script to call peaks with SPP.

Parameters

- **treatment_bam** (*str*) – Path to file with data for treatment sample.
- **control_bam** (*str*) – Path to file with data for control sample.
- **treatment_name** (*str*) – Name for the treatment sample.
- **control_name** (*str*) – Name for the control sample.
- **output_dir** (*str*) – Path to folder for output.
- **broad** (*str | bool*) – Whether to specify broad peak calling mode.
- **cpus** (*int*) – Number of cores the script may use.
- **qvalue** (*float*) – FDR, as decimal value

Returns Command to run.

Return type str

validate_bam(*input_bam*)

Wrapper for Picard's ValidateSamFile.

Parameters **input_bam** (*str*) – Path to file to validate.

Returns Command to run for the validation.

Return type str

1.18 FAQ

- **How can I run my pipeline on more than 1 sample?** Pypiper only handles individual-sample pipelines. To run it on multiple samples, write a loop, or use Looper. Dividing multi-sample handling from individual sample handling is a conceptual advantage that allows us to write a nice, universal, generic sample-handler that you only have to learn once.
- **What cluster resources can pypiper use?** Pypiper is compute-agnostic. You run it wherever you want; If you want a nice way to submit pipelines for samples any cluster manager, check out Looper.
- **What does it mean for a sample to be in the "waiting" state?** Waiting means it encountered a file lock, but no recovery flag. So the pipeline thinks a process (from another run or another process) is currently writing that file. It periodically checks for the lock file to disappear, and assumes that the other process will unlock the file when finished. If the lock was left by a previous failed run, then it will just wait forever. This is what recover mode (`-R`) is intended for, if pipelines fail.

- **What is the 'elapsed time' in output?** The "elapsed" time is referring to the amount of time since the preceding timestamp, not since the start of the pipeline. Timestamps are all displayed with a flag: `_TIME_`. The total cumulative time for the pipeline is displayed only at the end.
- **How should I run a QC step to check results of one of my commands?** Usually, you only want to run a QC step if the result was created in the same pipeline run. There's no need to re-run that step if you have to restart the pipeline due to an error later on. If you use `run()` for these steps, then they'll need to run each time the pipeline runs. Instead, this is exactly why we created *the follow argument*. This option lets you couple a QC step to a `run()` call, so it only gets excecuted when it is required.

1.19 Changelog

- **v0.7.0** (*2017-12-12*):
  - Standardize `NGSTk` function naming.
  - Introduce `Stage` as a model for a logically related set of pipeline processing steps.
  - Introduce `Pipeline` framework for automated processing phase execution and checkpointing.
  - Add ability to start and/or stop a pipeline at arbitrary checkpoints.
  - Introduce new state for a paused/halted pipeline.
  - Improve spawned process shutdown to avoid zombie processes.
- **v0.6** (*2017-08-24*):
  - Adds 'dynamic recovery' capability. For jobs that are terminated by an interrupt, such as a SIGINT or SIGTERM (as opposed to a failed command), pypiper will now set a dynamic recovery flags. These jobs, when restarted, will automatically pick up where they left off, without requiring any user intervention. Previously, the user would have to specify recover mode (`-R`). Now, recover mode forces a recover regardless of failure type, but interrupted pipelines will auto-recover.
  - Pypiper now appropriately adds cleanup files intermediate files for failed runs. It adds them to the cleanup script.
  - Improves error messages so only a single exception is raised with a more direct relevance to the user/
  - Pypiper will automatically remove existing flags when the run starts, eliminating the earlier issue of confusion due to multiple flags present on runs that were restarted.
  - Fixes a bug that caused a pipeline to continue if a SIGTERM is given during a process that was marked `nofail`.
  - Pypiper now can handle multiple SIGTERMs without one canceling the shutdown procedure begun by the other.
  - Major improvements to documentation and tutorials.
  - Adds `report_figure` function.
- **v0.5** (*2017-07-21*):
  - Adds preliminary support for handling docker containers
  - Updates docs, adds Hello World example
  - Adds 'waiting' flag
  - Eliminates extra spaces in reported results
  - Pypiper module is version aware

  - Updates Success time format to eliminate space
  - Improves efficiency in some ngstk merging functions
- **v0.4** (*2017-01-23*):
  - First major public release!
  - Revamps pypiper args
  - Adds parallel compression/decompression with pigz
  - Various small bug fixes and speed improvements

1.20 Support

1.20.1 Bug Reports

If you find a bug or want request a feature, open an issue at https://github.com/epigen/pypiper/issues.

1.20.2 Contributing

We welcome contributions in the form of pull requests.

1.21 Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

N

P

R

S

T

V