pyrpipe

Release 0.0.1

Jun 04, 2021

Contents

Author Urminder Singh

Date Jun 04, 2021

Version 0.0.5

CHAPTER 1

Introduction

pyrpipe (pronounced as “pyre-pipe”) is a python package to easily develop computational pipelines in pure python, in an object-oriented manner. pyrpipe provides an easy-to-use object oriented framework to import any UNIX executable command or tool in python. pyrpipe also provides flexible and easy handling of tool options and parameters and can automatically load the tool parameters from .yaml files. This framework minimizes the commands user has to write, rather the tools are available as objects and are fully re-usable. All commands executed via pyrpipe are automatically logged extensively and could be compiled into reports using the pyrpipe_diagnostic tool.

To enable easy and easy processing of RNA-Seq data, we have implemented specialized classes build on top of the pyrpipe framework. These classes provide high level APIs to many popular RNA-Seq tools for easier and faster development of RNA-Seq pipelines. These pipelines can be fully customizable and users can easily add/replace the tools using the pyrpipe framework. Finally, pyrpipe can be used on local computers or on HPC environments and pyrpipe scripts can be easily integrated into workflow management systems such as Snakemake and Nextflow.

Get started with the tutorial: *Tutorial* or look at real data examples.

1.1 Key Features

- Import any UNIX executable command/tool in python
- Dry-runnable pipelines to check dependencies and commands before execution
- Flexible and robust handling of tool arguments and parameters
- Auto load parameters from .yaml files
- Easily override threads and memory options using global values
- Extensive logging and reports with MultiQC reports for bioinformatic pipelines
- Specify GNU make like targets and verify the integrity of the targets
- Automatically resume pipelines/jobs from where interrupted
- Easily integrated into workflow managers like Snakemake and NextFlow

1.2 Installation

To install the latest stable release via conda:

```
conda install -c bioconda pyrpipe
```

To install the latest stable release via pip:

```
pip install pyrpipe --upgrade
```

Install latest development version :

```
git clone https://github.com/urmi-21/pyrpipe.git
pip install -r pyrpipe/requirements.txt
pip install -e pyrpipe
```

See the *Installation notes* for details.

CHAPTER 2

Examples and Case-Studies

Example usage and case-studies with real data is provided at GitHub.

2.1 Installation

Note: See *Create a new conda environment* in tutorial, to learn how to install pyrpipe and required tools in an conda environment.

Before installing pyrpipe, make sure conda channels are added in right order

```
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
```

pyrpipe is available through bioconda could be installed via:

```
conda install -c bioconda pyrpipe
```

pyrpipe is available on PyPI and could be installed via pip:

```
pip install pyrpipe
```

To install from source, clone the git repo:

```
git clone https://github.com/urmi-21/pyrpipe
```

Then install using pip:

```
pip install -r pyrpipe/requirements.txt
pip install -e pyrpipe
```

To run tests using pytest, from the pyrpipe root perform:

```
py.test
#or
py.test tests/<specific test file>
```

2.2 Tutorial

This tutorial provides an introduction to pyrpipe. New users can start here to get an idea of how to use pyrpipe APIs for RNA-Seq or other analysis. This tutorial is divided into smaller sections.

Section 1 describes the recommended way of installing pyrpipe and setting up conda environments.

Section 2 is an introduction to basic RNA-Seq API is provided via a simple RNA-Seq processing pipeline example.

Section 3 provides information on how to extend pipelines using third-party tools and integrate pyrpipe into snakemake for more scalable workflows.

Subsequent Sections detail pyrpipe_engine, pyrpipe_utils and pyrpipe_diagnostic, each of which provide helpful functionality.

2.2.1 Setting up the environment

First step is to setup the environment in a way such that reproducibility is ensured or maximized. In this tutorial we will use the conda environment manager to install python, pyrpipe and the required tools and dependencies into a single environment. **Note:** Conda must be installed on the system. For help with setting up conda, please see miniconda.

Create a new conda environment

To create a new conda environment with python 3.8 execute the following commands. We recommend sharing conda environment files with pipeline scripts to allow for reproducible analysis.

```
conda create -n pyrpipe python=3.8
```

Activate the newly created conda environment and install required tools

```
conda activate pyrpipe
conda install -c bioconda pyrpipe star=2.7.7a sra-tools=2.10.9 stringtie=2.1.4 trim-
↪galore=0.6.6 orfipy=0.0.3 salmon=1.4.0
```

If the above command fails, please try adding conda channels (see commands below) in the right order and then try again.

```
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
```

Using conda environment in yaml files

We have provided a yaml file containing the conda packages required to reproduce pyrpipe environment. Users can also use this file to create a conda environment and run pyrpipe. To create a conda environment, use the pyrpipe_environment.yaml:

```
conda env create -f pyrpipe_environment.yml
```

Users can easily export and share their own conda environment yaml files containing information about the conda environment. To export any conda environment as yaml, run the following command

```
conda env export | grep -v "^prefix: " > environment.yml
```

To recreate the conda environment in the *environment.yml*, use

```
conda env create -f environment.yml
```

Automated installation of required tools

We have also provided a utility to install required RNA-Seq tools via a single command:

```
pyrpipe_diagnostic build-tools
```

Note: Users must verify the versions of the tools installed in the conda environment.

Setting up NCBI SRA-Tools

After installing sra-tools, please configure prefetch to save the downloads the the **public user-repository**. This will ensure that the prefetch command will download the data to the user defined directory. To do this

- Type *vdb-config -i* command in terminal to open the NCBI SRA-Tools configuration editor.
- Under the TOOLS tab, set prefetch downloads option to **public user-repository**

Users can easily test if SRA-Tools has been setup properly by invoking the following command

```
pyrpipe_diagnostic test
```

2.2.2 Basic RNA-Seq processing

After setting up the environment, one can import pyrpipe modules in python and start using it. In this example we will use the *A. thaliana* Paired-end RNA-Seq run *SRR976159*.

Many other examples are available on github

Required files

We need to download the reference genome and annotation for *A. thalinia*. This can be done inside python script too. For simplicity we just download these using the *wget* command from the terminal.

Code for *A. thalinia* transcript assembly case study is available on github

```
wget ftp://ftp.ensemblgenomes.org/pub/release-46/plants/fasta/arabidopsis_thaliana/
↪dna/Arabidopsis_thaliana.TAIR10.dna.toplevel.fa.gz
gunzip Arabidopsis_thaliana.TAIR10.dna.toplevel.fa.gz
wget ftp://ftp.ensemblgenomes.org/pub/release-46/plants/gtf/arabidopsis_thaliana/
↪Arabidopsis_thaliana.TAIR10.46.gtf.gz
gunzip Arabidopsis_thaliana.TAIR10.46.gtf.gz
```

Simple pipeline

RNA-Seq processing is as easy as creating required objects and executing required functions. The following python script provides a basic example of using pyrpipe on publicly available RNA-Seq data.

```
1  from pyrpipe import sra,qc,mapping,assembly
2  #define some vaiables
3  run_id='SRR976159'
4  working_dir='example_output'
5  gen='Arabidopsis_thaliana.TAIR10.dna.toplevel.fa'
6  ann='Arabidopsis_thaliana.TAIR10.46.gtf'
7  star_index='star_index/athaliana'
8  #initialize objects
9  #creates a star object to use with threads
10 star=mapping.Star(index=star_index,genome=gen,threads=4)
11 #use trim_galore for trimming
12 trim_galore=qc.Trimgalore()
13 #Stringtie for assembly
14 stringtie=assembly.Stringtie(guide=ann)
15 #create SRA object; this will download fastq if doesnt exist
16 srr_object=sra.SRA(run_id,directory=working_dir)
17 #create a pipeline using the objects
18 srr_object.trim(trim_galore).align(star).assemble(stringtie)
19
20 #The assembled transcripts are in srr_object.gtf
21 print('Final result',srr_object.gtf)
```

The above code defines a simple pipeline (in a single line: Line 18) that:

- Downloads fastq files from NCBI-SRA
- Uses Trim Galore for trimming
- Uses Star for alignemnt to ref genome
- Uses Stringtie for assembly

A line by line explanation:

1. Imports the required pyrpipe modules
3. Lines 3 to 7 defines the variables for reference files, *output directory*, and star index. The *output directory* will be used to store the downloaded RNA-Seq data and will be the default directory for all the results.
10. Creates a Star object. It takes index and genome as parameters. It will automatically verify the index and if an index is not found, it will use the genome to build one and save it to the index path provided.
12. Creates a Trimgalore object
14. Creates a Stringtie object
16. Creates an SRA object. This represents the RNA-Seq data. If the raw data is not available on disk it auto-downloads it via fasterq-dump.
18. This is the pipeline which describes a series of operations. The SRA class implements the *trim()*, *align()* and *assemble()* methods.

- *trim()* takes a qc type object, performs trimming via *qc.perform_qc* method and the trimmed fastq are updated in the SRA object
- *align()* takes a mappping type object, performs alignemnt via *mapping.perform_alignemnt* method. The resulting bam file is stored in *SRA.bam_path*

- *assemble()* takes a assembly type object, performs assembly via *mapping.perform_assembly* method. The resulting gtf file is stored in *SRA.gtf*

Executing the pipeline

To execute the pipeline defined above, save the python code in a file *pipeline.py*. The code can be executed as any python script using the python command:

```
python pipeline.py
```

Or it can be executed using the pyrpipe command and specifying the input script with the –in option

```
pyrpipe --in pipeline.py
```

One can specify pyrpipe specific options too

```
python pipeline.py --threads 10 --dry-run
   #OR
pyrpipe --in pipeline.py --threads 10 --dry-run
```

The above two commands are equivalent and specifies pyrpipe to use 10 threads. Thus 10 threads will be used for each of the tool except for STAR where we explicitly specied to use 4 threads during object creation.

The other option provided here is the *–dry-run* option and this option *turns off* the pyrpipe_engine and any command passed to the pyrpipe_engine is not actually executed but just displayed on screen and logged. During dry run the Runnable class also verifies the file dependencies (if any). More details are provided in the later chapters of the tutorial.

We recommend using the dry-run option before actually starting a job to make sure all parameters/dependencies are correct.

Specifying tool parameters

pyrpipe supports auto-loading of tool parameters specified in .yaml files. The .yaml files must be stored is a directory and can be specified using the *–param-dir* option. The default value is *./params*. The files must be named as *<tool>.yaml*, for example star.yaml for STAR parameters. These parameters are loaded during object creation and user can easily override these during execution.

Create a directory *params* in the current directory and make a file *star.yaml* inside *params*. Add the following to *star.yaml* and rerun *pipeline.py* using the dry run option.

```
--outSAMtype: BAM Unsorted SortedByCoordinate
--outSAMunmapped: Within
--genomeLoad: NoSharedMemory
--chimSegmentMin: 15
--outSAMattributes: NH HI AS nM NM ch
--outSAMattrRGline: ID:rg1 SM:sm1
```

If you did everything correctly, you wil notice that now the STAR commands contain these specified parameters.

Updating parameters dynamically

The parameters specified in the yaml will be replaced by any parameters provided during object creation. For example, consider the star.yaml specifying –runThreadN as 20

```
--outSAMtype: BAM Unsorted SortedByCoordinate
--outSAMunmapped: Within
--genomeLoad: NoSharedMemory
--runThreadN: 20
```

Now, consider creating a star object in the following scenarios

```
star1=Star(index='index') #will use 20 threads as mentioned in the yaml file
star2=Star(index='index',threads=5) #will use 5 threads
star3=Star(index='index',**{'--runThreadN':'10'}) #will use 10 threads
star4=Star(index='index') #initialized with 20 threads
star4.run(...,**{'--runThreadN':'10'}) #will use 10 threads for this particular run;
↪'--runThreadN':'20' remains in the star4 object
```

2.2.3 Importing tools into python

pyrpipe's *Runnable* can be used to import any Unix command into python. The *Runnable* class implements the *run()* method which checks required dependencies, monitors execution, execute commands via pyrpipe_engine, and verify target files. We will first download the *E. coli* genome file to use as input to orfipy.

```
wget ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/005/845/GCF_000005845.2_ASM584v2/
↪GCF_000005845.2_ASM584v2_genomic.fna.gz
gunzip GCF_000005845.2_ASM584v2_genomic.fna.gz
```

Basic example

In this tutorial we will consider the tool *orfipy*, a tool for fast and flexible ORFs, and import it in python.

```
from pyrpipe.runnable import Runnable
infile='GCF_000005845.2_ASM584v2_genomic.fna'
#create a Runnable object
orfipy=Runnable(command='orfipy')
#specify orfipy options; these can be specified into orfipy.yaml too
param={'--outdir':'orfipy_out','--procs':'3','--dna':'orfs.fa'}
orfipy.run(infile,**param)
```

Save the above script in *example.py* and execute it using *python example.py –dry-run*. pyrpipe should generate the orfipy command and display on screen during dry-run. Running it without *–dry-run* flag will generate the *orfipy_out/orfs.fa* file.

Requirements and targets

One can specify required dependencies and expected target files in the run() method Replacing the call to *run()* with the following will verify the required files and the target files. If command is interrupted, pyrpipe will scan for *Locked* taget files and resume from where the pipeline was interrupted.

```
orfipy.run(infile,requires=infile,target='orfipy_out/orfs.fa',**param)
```

2.2.4 Building APIs

Users can extend the Runnable class and specify classes dedicated to tools. Extra functionalities can be added by defining custom behaviour of classes. The RNA-Seq API built using pyrpipe the framework.

A small example is presented here to build a class for orfipy tool

```python
from pyrpipe import Runnable

from pyrpipe.runnable import Runnable
from pyrpipe import pyrpipe_utils as pu
from pyrpipe import sra
from pyrpipe import _threads,_mem,_force
import os

class Orfipy(Runnable):
    """
    Extends Runnable class
    Attributes
    ----------

    """
    def __init__(self,*args,threads=None,mem=None,**kwargs):
        """
        init an Orfipy object

        Parameters
        ----------
        *args : tuple
            Positional arguements to orfipy
        threads : int, optional
            Threads to use for orfipy. This will override the global --threads␣
↪parameter supplied to pyrpipe. The default is None.
        mem : int, optional
            Maximum memory to use in MB. The default is None.
        **kwargs : dict
            options for orfipy
        Returns
        -------
        None.

        """
        super().__init__(*args,command='orfipy',**kwargs)
        self._deps=[self._command]
        self._param_yaml='orfipy.yaml'
        #valid arguments for orfipy
        self._valid_args=['--min','--between-stops','--include-stop','--dna','--pep',
↪'--bed','--bed12','--procs','--chunk-size','--outdir'] #valid arguments for orfipy

        #resolve threads to use
        """
        orfipy parameter for threads is --procs
        if threads is passed in __init__() it will be used
        else if --procs is found in orfipy.yaml that will be used
        else if --procs is found in the passed **kwargs in __init__() it will be used
        else the default value i.e. _threads will be used
        if default value is None nothing will be done
        after the function, --procs and its value will be stored in self._kwargs, and␣
↪_threads variable will be stored in the Orfipy object.
        """
        self.resolve_parameter("--procs",threads,_threads,'_threads')
        #resolve memory to use
        """
```

(continues on next page)

(continued from previous page)

```python
        default value is None--> if mem is not supplied don't make the self._mem␣
↪variable
        """
        self.resolve_parameter("--chunk-size",mem,None,'_mem')

    ##now we write a custom function that can be used with an SRA object
    def find_orfs(self,sra_object):
        out_dir=sra_object.directory

        out_file=os.path.join(out_dir,sra_object.srr_accession+"_ORFs.bed")

        if not _force and pu.check_files_exist(out_file):
            pu.print_green('Target files {} already exist.'.format(out_file))
            return out_file

        #In this example use orfipy on only first fastq file
        internal_args=(sra_object.fastq_path,)
        internal_kwargs={"--bed":sra_object.srr_accession+"_ORFs.bed","--outdir":out_
↪dir}

        #call run
        status=self.run(*internal_args,objectid=sra_object.srr_accession,target=out_
↪file,**internal_kwargs)

        if status:
            return out_file

        return ""
```

The above class, Orfipy, we created can be used directly with SRA type objects via the find_orfs function. We can also still use the *run()* method and provide any input to orfipy.

```python
#create object
orfipy=Orfipy()
#use run()
orfipy.run('test.fa',**{'--dna':'d.fa','--outdir':'of_out'},requires='test.fa',target=
↪'of_out/d.fa')

#use the api function to work with SRA
srr=sra.SRA('SRR9257212')
orfipy.find_orfs(srr)
```

Now try passing orfipy parameters from orfipy.yaml file. Create *params/orfipy.yaml* and add the following options into it.

```yaml
--min: 36
--between-stops: True
--include-stop: True
```

Now re-run the python code and it will automatically read orfipy options from *./params/orfipy.yaml*.

2.2.5 Snakemake example

Since Snakemake direclty supports python, pyrpipe libraries can be driectly imorted into snakemake. Advantage of using a workflow manager like Snakemake is that it can handle parallel job-scheduling and scale jobs on clusters.

Basic RNA-Seq example

A basic example of directly using pyrpipe with Snakemake is provided here. This example uses yeast RNA-Seq samples from the GEO accession GSE132425.

First run the following bash script to download the yeast reference genome from Ensembl. The last command also generated a star index with the downloaded genome under the *refdatayeast_index* directory.

```
#!/bin/bash
mkdir -p refdata
wget ftp://ftp.ensemblgenomes.org/pub/fungi/release-49/fasta/saccharomyces_cerevisiae/
↪dna/Saccharomyces_cerevisiae.R64-1-1.dna.toplevel.fa.gz -O refdata/Saccharomyces_
↪cerevisiae.R64-1-1.dna.toplevel.fa.gz
wget ftp://ftp.ensemblgenomes.org/pub/fungi/release-49/gff3/saccharomyces_cerevisiae/
↪Saccharomyces_cerevisiae.R64-1-1.49.gff3.gz -O refdata/Saccharomyces_cerevisiae.R64-
↪1-1.49.gff3.gz
cd refdata
gunzip -f *.gz
#run star index
mkdir -p yeast_index
STAR --runThreadN 4 --runMode genomeGenerate --genomeDir yeast_index --
↪genomeFastaFiles Saccharomyces_cerevisiae.R64-1-1.dna.toplevel.fa --
↪genomeSAindexNbases 10
```

Next, we create the following *snakefile*.

```
import yaml
import sys
import os
from pyrpipe import sra,qc,mapping,assembly

####Read config#####
configfile: "config.yaml"
DIR = config['DIR']
THREADS=config['THREADS']
##check required files
GENOME= config['genome']
GTF=config['gtf']
#####Read SRR ids######
with open ("runids.txt") as f:
    SRR=f.read().splitlines()

#Create pyrpipe objects
#parameters defined in ./params will be automatically loaded, threads will be␣
↪replaced with the supplied value
tg=qc.Trimgalore(threads=THREADS)
star=mapping.Star(threads=THREADS)
st=assembly.Stringtie(threads=THREADS)

rule all:
    input:
            expand("{wd}/{sample}/Aligned.sortedByCoord.out_star_stringtie.gtf",
↪sample=SRR,wd=DIR),

rule process:
    output:
            gtf="{wd}/{sample}/Aligned.sortedByCoord.out_star_stringtie.gtf"
    run:
```

(continues on next page)

(continued from previous page)

```
            gtffile=str(output.gtf)
            srrid=gtffile.split("/")[1]
            sra.SRA(srrid,directory=DIR).trim(tg).align(star).assemble(st)
```

The above snakefile requires some additional files. A config.yaml file contains path to data and threads information.

```
DIR: "results"
THREADS: 5
genome: "refdata/Saccharomyces_cerevisiae.R64-1-1.dna.toplevel.fa"
gtf: "refdata/Saccharomyces_cerevisiae.R64-1-1.49.gff3"
```

Next the snakefile reads a file, *runids.txt* containing SRR accessions.

```
SRR9257163
SRR9257164
SRR9257165
```

Finally, we need to provide tool parameters for pyrpipe. Create a file *./params/star.yaml* and specify the index in it

```
--genomeDir: ./refdata/yeast_index/
```

Now the snakefile could be run using the snakemake command e.g *snakemake -j 8*

pyrpipe_conf.yaml

Users can create a yaml file, *pyrpipe_conf.yaml*, to specify pyrpipe parameters, instead of directly passing them as command line arguments. When the *pyrpipe_conf.yaml* is found the pyrpipe specific arguments passed via the command-line are ignored. An example of *pyrpipe_conf.yaml* is shown below with the pyrpipe default values

```
dry: False           # Only print pyrpipe's commands and not execute anything through␣
↪pyrpipe_engine module
threads: None        # Set the number of threads to use
force: False         # Force execution of commands if their target files already exist
params_dir: ./params # Directory containing parameters
logs: True           # Enable or disable pyrpipe logs
logs_dir: ./pyrpipe_logs     # Directory to save logs
verbose: False       # Display pyrpipe messages
memory: None         # Set memory to use (in GB)
safe: False          # Disable file deletion via pyrpipe commands
multiqc: False       # Automatically run multiqc after analysis
```

2.2.6 RNA-Seq API

pyrpipe implements specialized classes for RNA-Seq processing. These classes are defined into different modules, each designed to capture steps integral to analysis of RNA-Seq data –from downloading raw data to trimming, alignment and assembly or quantification. Each of these module implements classes coressponding to a RNA-Seq tool. These classes extend the *Runnable* class. Specialized functions are implemented such that analysis of RNA-Seq data is intuitive and easy to code. The following table provide details about the pyrpipe's RNA-Seq related modules.

Module	Class	Purpose
assembly	Assembly	Abstract class to represent Assebler type
assembly	Stringtie	API to Stringtie
assembly	Cufflinks	API to Cufflinks
mapping	Aligner	Abstract class for Aligner type
mapping	Star	API to Star
mapping	Bowtie2	API to Bowtie2
mapping	Hisat2	API to Hisat2
qc	RNASeqQC	Abstract class for RNASeqQC type (quality control and trimming)
qc	Trimgalore	API to Trim Galore
qc	BBmap	API to bbduk.sh
quant	Quant	Abstract class for Quantification type
quant	Salmon	API to Salmon
quant	Kallisto	API to Kallisto
sra	SRA	Class to represent RNA-Seq data and API to NCBI SRA-Tools
tools	Samtools	API to Samtools and other commonly used tools

The SRA class

The SRA class contained in the sra module captures RNA-Seq data. It can automatically download RNA-Seq data from the NCBI SRA servers via the prefetch command. The SRA constructor can take the SRR accession, path to fastq or sra file as arguments.

The main attributes and functions are defined the following table.

Attribute	Description
fastq_path	Path to the fastq file. If single end this is the only fastq file.
fastq2_path	Path to the second fastq file for paired end data.
sra_path	Path to the sra file
srr_accession	The SRR accession for RNA-Seq run
layout	RNA-Seq layout, auto determined by SRA class.
bam_path	Path to bam file after running the align() function
gtf	Path to the gtf file after running assemble() function

Function	Description
__init__()	This is the constructor. It can take SRR accession, path to fastq files, or sra file as input. If accession if provided as input the files are downloaded via prefetch if they aren't preset on disk. It will automatically handle single-end and paired-end data.
download_sra()	This function downloads the sra file via prefetch.
download_fastq()	This function runs fasterq-dump on the sra file downloaded via prefetch.
sra_exists()	Check if fastq sra files are present
fastq_exists()	Check if fastq file exists
delete_sra()	Delete the sra file
delete_fastq()	Delete the fastq files
trim()	This function takes a RNASeqQC type object and performs trimming. The trimmed fastq files are then stored in fastq_path and fastq2_path.
align()	This function takes an Aligner type object and performs read alignemnt. The BAM file returned is stored in bam_path attribute.
assemble()	This function takes an Assembly type object and performs transcript assembly. The result is soted on the SRA object as gtf attributes
quant()	This function takes a Quantification type object and performs quant.

The RNASeqQC class

The RNASeqQC is an abstract class defined in the qc module. RNASeqQC class extends the Runnable class and thus has all the attributes as in the Runnable class. Classes Trimgalore and BBmap extends RNASeqQC class and share following attributes and functions.

Attribute	Description
_category	Represents the type: "RNASeqQC"

Function	Description
__init__()	The constructor function
perform_qc()	Takes a SRA object, performs qc and returns path to resultant fastq files

The Aligner class

The Aligner is an abstract class defined in the mapping module. Aligner class extends the Runnable class and thus has all the attributes as in the Runnable class. Classes Star, Hisat2 and Bowtie2 extends the Aligner class and share following attributes and functions.

Attribute	Description
_category	Represents the type: "Aligner"
index	Index used by the aligner tool
genome	Reference genome used by the tool

Function	Description
__init__()	The constructor function
build_index()	Build an index for the aligner tool using the *genome*.
check_index()	Checks if the index is valid
perform_alignment()	Takes a sra object, performs alignemnt and returns path to the bam file

The Assembly class

The Assembly is an abstract class defined in the assembly module. Assembly class extends the Runnable class and thus has all the attributes as in the Runnable class. Classes Stringtie and Cufflinks extends the Assembly class and share following attributes and functions.

Attribute	Description
_category	Represents the type: "Assembler"

Function	Description
__init__()	The constructor function
perform_assembly()	Takes a SRA object, performs transcript assembly and returns path to resultant gtf/gff files

The Quant class

The Quant is an abstract class defined in the quant module. Quant class extends the Runnable class and thus has all the attributes as in the Runnable class. Classes Salmon and Kallisto extends the Quant class and share following attributes and functions.

Attribute	Description
_category	Represents the type: "Quantification"
index	Index used by the aligner tool
transcriptome	Reference transcriptome used by the tool

Function	Description
__init__()	The constructor function
build_index()	Build an index for the quantification tool using the *transcriptome*.
check_index()	Checks if the index is valid
perform_quant()	Takes a sra object, performs quantification and returns path to the quantification results file

2.2.7 pyrpipe_engine module

The pyrpipe_engine module is the key module responsible for handling execution of all the Unix commands. The Runnable class calls the execute_comand() function in the pyrpipe_engine module. All the commands executed via the pyrpipe_engine module are automatically logged. All the functions responsible for executing Unix command are "decorated" with the dryable method, which allows using the –dry-run flag on any function using the pyrpipe_engine module.

The pyrpipe_engine can be directly used to execute Unix command or to import output of a Unix command in python. The functions defined in the pyrpipe_engine module are described below.

Function	Description
dryable()	A decorater to make Unix functions dry when –dry-run is specified
parse_cmd()	Parse a Unix command and return command as string
get_shell_output()	Execute a command and return return code and output. These commands are not logged
get_return_status()	Execute command and return status of the command
exe-cute_commandRealtime()	Execute a command and print output in realtime
execute_command()	Execute a command and log the status. This is the function used by the Runnable class.
is_paired()	Check is sra file paired or single end. Uses fastq-dump
get_program_path()	Return path to a Unix command
check_dependencies()	Check a list of dependencies are present in Unix path
delete_files()	Delete files, rm command
move_file()	Moves files, mv command

2.2.8 pyrpipe_utils module

The pyrpipe_utils module define multiple helpful functions. Functions defined in the pyrpipe_utils modules are extensively used throughout pyrpipe modules. Users can directly utilize these fuctions in their code and expedite development. A description of these functions is provided below.

Function	Description
pyrpipe_print()	Prints in color
get_timestamp()	Return current timestamp
check_paths_exist()	Return true is paths are valid
check_files_exist()	Return True if files are valid
check_hisatindex()	Verify valid Hisat2 index
check_kallistoindex()	Verify kallisto index
check_salmonindex()	Verify salmon index
check_starindex()	Verify STAR index
check_bowtie2index()	Verify Bowtie2 index
get_file_size()	Return file size in human readable format
parse_java_args()	Parse tool options in JAVA style format
parse_unix_args()	Parse tool options in Unix style format
get_file_directory()	Return a file’s directory
get_filename()	Return filename with extension
get_fileext()	Return file extension
get_file_basename()	Return filename, without extension
mkdir()	Create a directory
get_union()	Return union of lists
find_files()	Search files using regex patterns
get_mdf()	Compute and return MD5 checksum of a file

2.2.9 pyrpipe_diagnostic

The pyrpipe_diagnostic command allows users to easily examine pyrpipe logs. It provides several options.

1. report: This command can generate a summary or detailed report of the analysis.
2. shell: This command creates a bash file containing all the commands executed via pyrpipe

3. benchmark: This command can generate benchmarks to compare walltimes of the different commands in the pipeline.
4. multiqc: This command uses the MultiQC tool to generate a report using pyrpipe logs and other logs geenrated by the pipeline.

2.3 Cookbook

Contents

2.3.1 Using SRA objects

```python
from pyrpipe.sra import SRA #imports the SRA class

#create an SRA object using a valid run accession
"""
this checks if fastq files already exist in the directory,
otherwise downloads the fastq files and stores the path in the object
"""
myob=SRA('SRR1168424',directory='./results')

#create an SRA object using fastq paths
myob=SRA('SRR1168424',fastq='./fastq/SRR1168424_1.fastq',fastq2='./fastq/SRR1168424_2.
↪fastq')

#create an SRA object using sra path
myob=SRA('SRR1168424',sra='./sra/SRR1168424.sra')

#accessing fastq files
print(myob.fastq,myob.fastq2)

#check if fastq files are present
print (myob.fastq_exists())

#check sra file
```

(continues on next page)

(continued from previous page)

```
print (myob.sra_exists())

#delete fastq
myob.delete_fastq()

#delete sra
myob.delete_sra()

#download fastq
myob.download_fastq()

#trim fastq files
myob.trim(qcobject)
```

2.3.2 Using RNASeqQC objects

RNASeqQC objects can be used for quality control and trimming. These are defined in the *qc* module. Following example uses Trimgalore class but is applicable to any class extending the *RNASeqQC* class

```
from pyrpipe.qc import Trimgalore

#create trimgalore object
tgalore=Trimgalore()
#print category
print(tgalore._category) #should print Aligner

#use with SRA object
"""
Following will trim fastq and update fastq paths in the sraobject
"""
sraobject.trim(tgalore)
#following trims and returns qc
fq1,fq2=tgalore.perform_qc(sraobject)

#run trimgalore using user arguments; provide any arguments that Runnable.run() can␣
↪take
tgalore.run(*args,**kwargs)
```

2.3.3 Using Aligner objects

Aligner objects from the *mapping* module can be used to perform alignment tasks.

```
from pyrpipe.mapping import Star

#create a star object
star=Star(index='path_to_star_index')

#print category
print(star._category) #should print Aligner

#perform alignment using SRA object
bam=star.perform_alignment(sraobject)
#or
```

(continues on next page)

(continued from previous page)

```
sraobject.align(star)
bam=sraobject.bam_path

#execute STAR with any arguments and parameters
kwargs={'--outFilterType' : 'BySJout',
        '--runThreadN': '6',
        '--outSAMtype': 'BAM SortedByCoordinate',
        '--readFilesIn': 'SRR3098744_1.fastq SRR3098744_2.fastq'
        }
star.run(**kwargs)
```

2.3.4 Using Assembler objects

Assembler objects are defined the the assembly module and can be used for transcript assembly.

```
from pyrpipe.assembly import Stringtie

#create a stringtie object
stringtie=Stringtie(guide='path_to_ref_gtf')

#perform assembly using SRA object
"""
Note: following first runs star to perform alignment. After alignment the Sorted
BAM file is stored in the sraobject.bam_path attribute and returns the modified␣
↪sraobject
The assemble function requires a valid bam_path attribute to work.
"""
sraobject.align(star).assemble(stringtie)
#Or manually set bam_path
sraobject.bam_path='/path/to/sorted.bam'
sraobject.assemble(stringtie)

#use perform_assembly function
result_gtf=stringtie.perform_assembly('/path/to/sorted.bam')

#run stringtie with user arguments
stringtie.run(verbose=True, **kwargs)
```

2.3.5 Using Quantification objects

Quantification type objects can perform quantification and are defined inside the *quant* module.

```
from pyrpipe.assembly import Salmon

#create salmon object
"""
A valid salmon idex is required. If index is not found it is built using the provided␣
↪transcriptome
"""
salmon=Salmon(index='path/to/index',transcriptome='path/to/tr')

#directly quantify using SRA object
sraobject.quant(salmon)
```

(continues on next page)

(continued from previous page)

```
#or trim reads before quant
sraobject.trim(tgalore).quant(salmon)
print('Result file',sraobject.abundance)

#use perform quant function
abundance_file=salmon.perform_quant(sraobject)

#use salmon with user defined arguments
salmon.run(**kwargs)
```

2.3.6 Using RNASeqTools objects

The *RNASeqTools* type is defined in *tools* module. This contains various tools used routinely for RNA-Seq data processing/analysis

```
from pyrpipe.tools import Samtools

#create samtools object
samtools=Samtools(threads=6)

#convert sam to sorted bam
bam=samtools.sam_sorted_bam('sam_file')

#merge bam files
mergedbam=samtools.merge_bam(bamfiles_list)


#run samtools with used defined arguments
"""
NOTE: the Runnable.run() method accepts a subcommand argument that allows user to␣
↪procide a subcommand like samtools index or samtools merge
"""
samtools.run(*args,subcommand='index',**kwargs)
```

2.3.7 Using Runnable objects

The *Runnable* class, defined inside the *runnable* module, is the main parent class in *pyrpipe* i.e. all other classes borrows its functionality. User can directly create *Runnable* objects to define their own tools/commands.

A full example to build APIs is here: *Building APIs*

```
from pyrpipe.runnable import Runnable

#say you want to use the tool orfipy
orfipy=Runnable(command='orfipy')

#execute orfipy as
orfipy.run(*args,**kwargs)

#another example using Unix grep
grep=Runnable(command='grep')
grep.run('query1','file1.txt',verbose=True)
grep.run('query2','file2.txt',verbose=True)
```

(continues on next page)

(continued from previous page)

```python

#extend Runnable to build more complex APIs that fit with each other
"""
One can create classes extending the Runnable class.
Full example is given in the tutorial
"""
class Orfipy(Runnable):
    def __init__(self,*args,threads=None,mem=None,**kwargs):
        super().__init__(*args,command='orfipy',**kwargs)
        self._deps=[self._command]
        self._param_yaml='orfipy.yaml'

    #create special API functions that can work with other objects
    def find_orfs(self,sra_object):
        #define logic here and gather command options and parameters

        #call the self.run() function and check values

        #return a useful value
```

2.3.8 Using pyrpipe_engine module

The *pyrpipe_engine* module contain functions that creates new processes and enable executing commands. User can directly import *pyrpipe_engine* module and start using the functions. This is very useful for quickly executing commands without having to create a Runnable object. A table describing functions implements in *pyrpipe_engine* is provided in the tutorial *pyrpipe_engine module*

```python
import pyrpipe_engine as pe

#execute_command: Runs a command, logs the status and returns the status (True or␣
↪False)
pe.execute_command(['ls', '-l'],logs=False,verbose=True)

#get_shell_output Runs a command and returns a tuple (returncode, stdout and stderr)
"""
NOTE: only this function supports shell=True
"""
result=pe.get_shell_output(['head','sample_file'])
#result contains return code, stdout, stderr
print(result)

#make a function dry-run compatible
"""
when --dry-run flag is used this function will be skipped and the first parameter 'cmd
↪' will be printed to screen
"""
@pe.dryable
def func(cmd,...)
    #function logic here

"""
Other way to work with dry-run flag is to directly import _dryrun flag
"""

from from pyrpipe import _threads,_force,_dryrun
```

(continues on next page)

(continued from previous page)

```python
def myfunction(...):
    if _dryrun:
        print('This is a dry run')
        return

    #real code here...
```

2.3.9 Using pyrpipe_utils module

The *pyrpipe_utils* module contains several helpful functions that one needs to frequently access for typical computational pipelines. A table describing functions implements in *pyrpipe_utils* is provided in the tutorial *pyrpipe_utils module*

```python
import pyrpipe_utils as pu

#check if files exist
pu.check_files_exist('path/f1','path/f2','path/f3') #returns bool

#get filename without extention
pu.get_file_basename('path/to/file.ext')

#get file directory
pu.get_file_directory('path/to/file.ext')

#create a directory
pu.mkdir('path/to/dir')

#Search .txt files in a directly
pu.find_files('path/to/directory','.*\.txt$',recursive=False,verbose=False)
```

Contents:

- genindex
- modindex
- search