Snakemake Module Vignette: Intro to Pipelines

The following documentation is to be used as an introduction to the understanding and development of Snakemake based pipeline modules which are run on a Sun Grid Engine (CentOS5) and support in Python (3.5.1).

Written By: Tim Boyarski, BSc

With assistance from: Lauren Chong, MSc

For: BC Cancer Agency – Lymphoid Cancer Research

Date of: March 28th, 2017

Contents

[I. Snakemake module organization 2](#_Toc479769562)

[II. Module Design Tutorial - Overview 3](#_Toc479769563)

[III. Single Module Design - Intro 4](#_Toc479769564)

[IV. Single Module Design - INCLUDE 5](#_Toc479769565)

[V. Single Module Design - PY 9](#_Toc479769566)

[VI. Single Module Design - README 13](#_Toc479769567)

[VII. Multiple Module Design - Intro 15](#_Toc479769568)

[VIII. Multiple Module Design - INCLUDE 16](#_Toc479769569)

[IX. Multiple Module Design - PY 18](#_Toc479769570)

[X. Multiple Module Design - README 18](#_Toc479769571)

[XI. References 18](#_Toc479769572)

[XII. Appendices 18](#_Toc479769573)

## Snakemake module organization

All Snakemake modules are to be stored within the modules directory. Inside the directory, modules are to be developed with the intent of high cohesion. This means that we must attempt to keep modules small and focused. Each module will have at minimum three files (.md, .py, and \_INCLUDE). An example structure is outlined in Figure 1 and Figure 2.

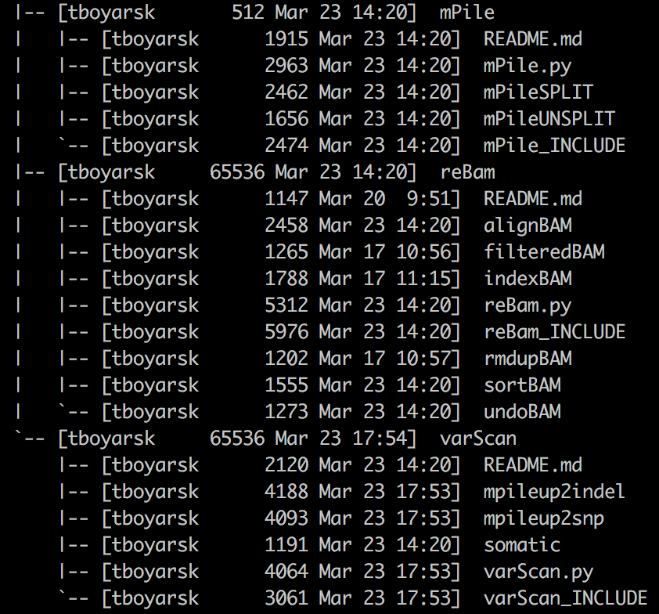


Figure . Tree structure of the module directory on Genesis

Snakemake/modules

/module0

/README.md

/module0.py

/module0\_INCLUDE

/module1

/README.md

/module1.py

/module1\_INCLUDE

/subModule1A

/subModule12B

…

/moduleN

/README.md

/moduleN.py

/moduleN\_INCLUDE

Figure . Hypothetical structure of modules directory

Naming: Modules should be named after their purpose. A *varScan* module uses varScan. A *reBAM* module re-assembles and processes ‘.BAM’ files. The modules themselves do not need to have the word ‘module’ in them, it’s just used in the example above by chance. Sub-modules are then named by what they specifically accomplish. *sortBAM* sorts BAM files, *indexBAM* indexes ‘.BAM’ files, etc.

Multi-Rule Composition: Complex modules are decomposed into smaller events. We highly encourage that these small events are separated into their own Snakefile. They are all included into the core ‘\_INCLUDE’ file. For multi-part composition, the ‘\_INCLUDE’ should only contain the ‘include’ statements and a description about the submodules included. *When using submodules, there should be no rules in the ‘\_INCLUDE’ file.*

Core files:

* **README.md** – A read-me file written in markdown format.
* **moduleN.py** – A python script which accepts the names of the ‘.YAML’ file, ‘JSON’ file, and Snakefile. It will act as a script to populate the ‘.YAML’ file, ‘JSON’ file, and Snakefile.
* **moduleN\_INCLUDE** – The Snakefile to be included into the pipeline.
  + - **Single-Rule**: Contains only 1 rule.
    - **Multi**-**Rule**: Contains no rules. Contains submodule ‘include’ statements and a significant description for each submodule.
* **subModuleA** – Used in multi-rule modules only; isolating each subModule file to have only 1 rule. Add subModules as needed.

## Module Design Tutorial - Overview

The following document has been designed to provide first-hand experience developing a module for the Snakemake pipeline system. The tutorial is to reproduce the existing production ready module called *mPile*. The recreated *mPile* module will be called *tPile*.

A completed copy of *tPile* will be stored inside the module’s Template/ directory for the following reasons:

* Provide quick reference to users when they are following this tutorial.
* Provide quick answers to users to utilize when they get stuck while following this tutorial.
* Prevents users from accidentally editing the original *mPile* module.

The *tPile* module directory will actually contain two directories. The reason *tPile* has sub-directories is a result of this tutorial. This is atypical and no other module directory will have sub-directories. The directories exist to exemplify the differences between single and multiple modules systems. As well, it also provides context for the transition to multiple modules. Because the single module system is also available as a multiple module system users can better compare the differences in template usage and file naming.

\*\* NOTE \*\* If you were wondering ‘t’ in *tPile* stands for tutorial. I don’t know what the ‘m’ is in *mPile*.

\*\* NOTE \*\* The directory *tPile* may not appear in all screenshots of the Template/ directory.

\*\* NOTE \*\* If developing non-tutorial modules and copying the Template/ directory, *tPile* can be deleted.

An example of the directory structure of the *tPile* module can be seen in Figure 3.

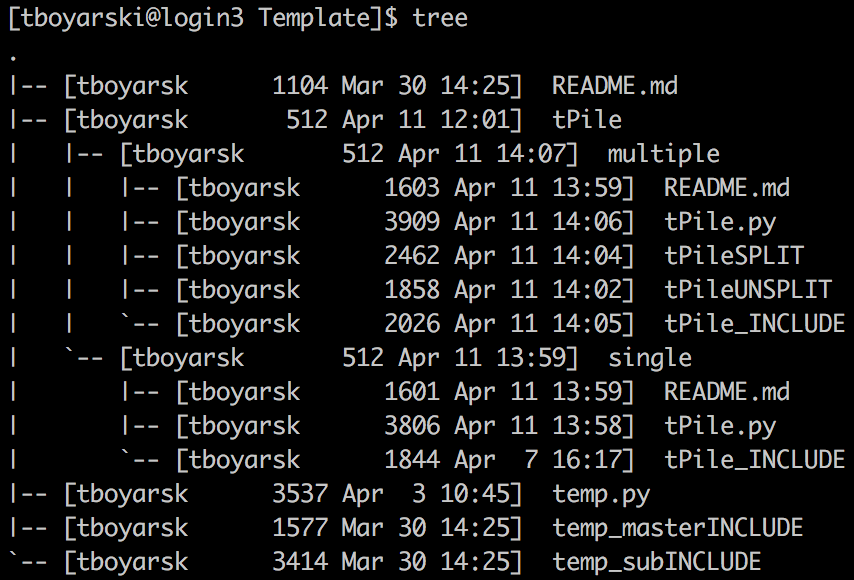


Figure . VIM screenshot of the current Template/ directory setup on Genesis. Note that the Template/ directory contains an answer key for this tutorial.

## Single Module Design - Intro

A key focus of module design is cohesion, keeping modules small and focused.

* Primary calls, those which run programs, should all be separated into a different module.
* Secondary calls, those which support a primary call, include: print statements, directory creation, input redirection, parsing commands, and other administrative functions.

The following guide explains how to build the non-chromosome splitting mPile Snakemake module.

\*\*WARNING: Please note this module already exists, and has already been expanded beyond a single module design. Have a colleague assist in setting up the module workspace such that following this tutorial will not replace production ready files. This vignette is having you create a simplified version of the mPile, called “tPile”.

1. **In your local checkout of the modules directory, copy and rename the Template/ directory.**

Figure 4 shows the contents inside the modules directory at the time of writing this document. As indicated by the yellow box, copy the Template/, and name the new directory “tPile”.

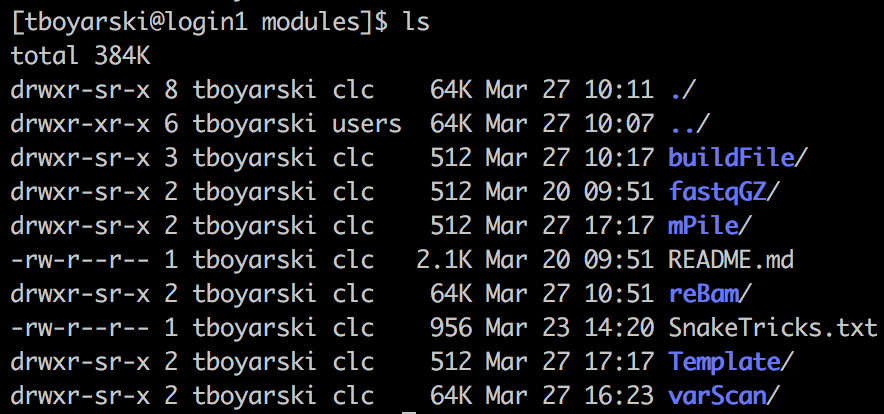


Figure . Genesis screenshot of the modules directory. Highlighted in the yellow box is the 'Template' directory users have been instructed to copy.

1. **Enter the renamed module directory and rename the files to reflect the module name.**

Only multiple module designs require the “temp\_masterINCLUDE” file, as such it can be deleted. Files containing a single rule, like the one we are about to write, use the template file “temp\_subINCLUDE”. For clarification in naming, please refer to Table 1. The python file and the rule-containing Snakefile should be named in accordance with the module directory name. This results in the files being named “tPile.py” and “tPile\_INCLUDE”, respectively.

|  |  |  |
| --- | --- | --- |
| Directory item | Action | Final Name |
| README.md | No action | README.md |
| temp\_masterINCLUDE | Deleted | N/A |
| temp.py | Renamed | tPile.py |
| temp\_subINCLUDE | Renamed | tPile\_INCLUDE |
| tPile/ | No action | ~ This is the answer key ~ |

Table . A conversion table to assist in when renaming files from their template names. Names were chosen in accordance with the development of the mPile module.

### Single Module Design - INCLUDE

The “\_INCLUDE” module is the file which is provided to the Snakefile pipeline. It gives the pipeline access to the rules of the module. In this case, for a single module, the rules are contained within this “\_INCLUDE” file. In any module directory, there should only ever be a single “\_INCLUDE” file.

1. **Using VIM, open the “\_INCLUDE” file.**

Open “tPile\_INCLUDE”.

1. **Highlight the fields to be edited by moving the cursor to “XXXXXXXX” on Line 13, and pressing “Shift + #”**

Red highlighting should appear as seen in Figure 5. These are the fields that will need to be edited. The following steps will refer to the line numbers in Figure 5. The coloured boxes below are to help clarify variable usage.

Rule’s Log or Parameter Variable

.YAML Configuration File Variable



Figure . Genesis screenshot of the starting "tPile\_INCLUDE" file, after performing Step 3. The highlighted fields are areas of text which will need to be replaced. The highlighting can be turned off using the VIM command ":noh".

1. **Go to Line 2 and Line 3, insert your name and the date.**
2. **Go to Line 5, edit the python call.**

Typically, this should just involve replacing the highlighted X’s with the module name, as seen in Figure 6.

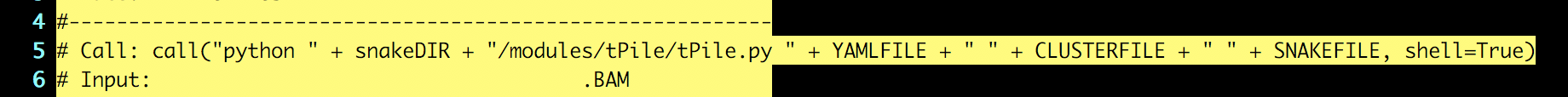


Figure . VIM screenshot for what Line 5 from the tPile\_INCLUDE file should look like. This will allow for easy access when needing to refer to this file while building a pipeline build file.

1. **Go to Line 6, Line 7, and Line 8, insert the input file type, output file type, and the module’s purpose.**

This module utilizes Samtools’s mpileup function to generate a .mpileup file from an input .bam file (http://www.htslib.org/doc/samtools.html). The resulting pileup file summarizes the sequencing depth, quality, and other features of each base position.

1. **Go to Line 13, edit the name of the rule to be the same as the file prefix.**

The file is named “tPile\_INCLUDE”. Hence the prefix (“tPile”) should be the name of this rule and the directory.

1. **Go to Line 15, edit the input for the module.**

A lot of the power in Snakemake and Make comes from their ability to specify starting input files, targeting with precision and ambiguity simultaneously. With this power comes a complexity.

VARIABLES

Named Wildcards: Indicated by double {} E.G. {{namedWildcard}}

Wildcards are automatically determined via regex parsing of the argument used to run your pipeline.

Inline Variables: Indicated by single {} E.G. {rndVAR}

Inline variables can be used as place holders when populate with data, like from the YAML config file.

They can be of the following formats:

Strings … rndVAR=“Strings are formatted in quotes”

Lists … rndVAR=[“Lists”, “follow”, “python”, “formatting”]

YAML Parameters … rndVAR=config[“paramNAME”]

FUNCTIONS VARIABLES

Input Function: Functions can be used in the input directive, provided they have a return value. Functions called from the input directive are automatically given the Snakemake Wildcard Object. As such, the method signature must accept a formal parameter. For consistency, please always name the first formal argument “wildcards”. This is not used in this module, as such, the concept will not be further explained nor will examples be given.

Expand Function: The expand function is the most commonly used Snakemake function. It is used to generate the sets of input and output files required. Variables passed into this function perform a full union. For more information on the expand function, please refer to the following link to the [Snakemake documentation](http://snakemake.readthedocs.io/en/latest/tutorial/basics.html?highlight=expand()#step-5-calling-genomic-variants).

EXAMPLES VARIABLES

Using:

X = “output”

Y = “mutect”

Wildcard1 = “Pfeiffer”

Z = [“1”,”2”]

If we have: expand(“{var1}/{var2}/{{wildcard1}}\_{var3}.fa”, var1=X, var2=Y, var3=Z)

It becomes: output/mutect/Pfeiffer\_1.fa, output/mutect/Pfeiffer\_2.fa

In our tPile module the input is a .bam file, so the input argument should specify the location of this file for a given sample. The resultant tPile input call should be the same call as in mPile/mPileUNSPLIT; Both should look like Figure 7.

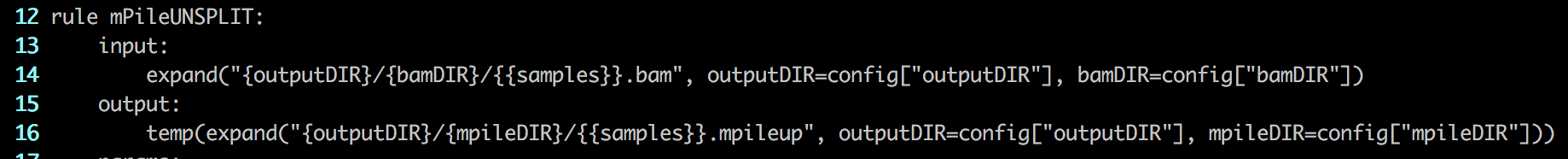


Figure . VIM screenshot of Line 14 of the mPileUNSPLIT module. This module is named differently from the vignette because it has since been converted to a submodule.

1. **Go to Line 17, edit the output for the module.**

Aside from two exceptions, the output directive is designed the same as the input directive.

Exception 1: The output directive cannot use functions

Exception 2: The output directive has 2 file protection tags

* + 1. temp(“filename”) – “filename” will be deleted when it is no longer required by the pipeline.
    2. protected(“filename”) – “filename” is protected from being deleted by the pipeline, even if it is just an intermediate.

In our tPile module, the output is a .mpileup file. The resultant tPile output call should be the same as Figure 8.

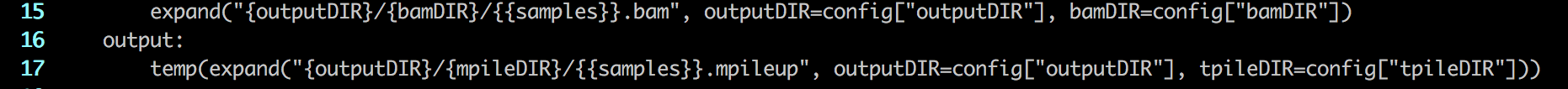


Figure . VIM screenshot of what lines 16-17 of tPile\_INCLUDE should look like.

1. **Go to Line 19, edit the name of the log file.**

This should be the same as the prefix to the file. It is the same term used to name the rule and the directory.

1. **Go to Lines 20, edit and add more parameters as needed.**

Any additionally required parameters which are not specifically listed in the ‘.YAML’ configuration file can be specified here. Python calls can be utilized here as well as references to the “.YAML” file variables. For ease of reading it is encouraged that users combine related arguments together, as seen in Figure 9. This is especially encouraged when there are five or more arguments to be passed to the program. In this module we will add one argument specifying the flag used in our BAM pre-filtering (bitFLAG), another argument summarizing the mpileup-specific arguments defined in our config file (tpilePARAMS), and another defining the location of our reference file (refLOC). Each argument must be separated by a comma.

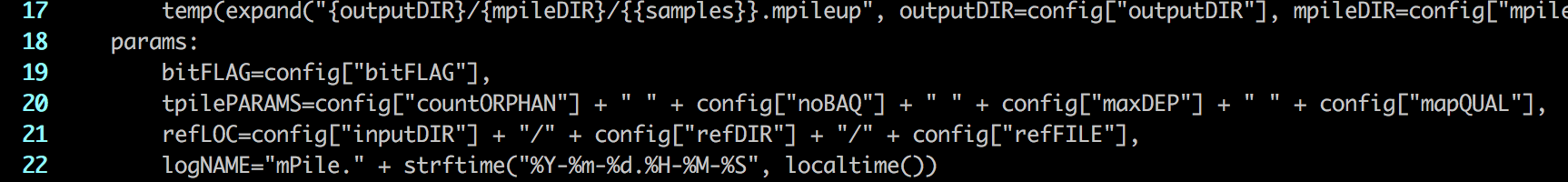


Figure . VIM screenshot of what lines 18-22 of tPile\_INCLUDE should look like. The variable tpilePARAMS is a union of all the parameters which will have to be passed to the program mpileup. This makes the resultant bash shell call short and therefore easier to read.

1. **Go to Line 24, edit the name of the log directory.**

This should be the same as the prefix to the file. It is the same term used to name the rule and the directory.

1. **Go to Line 26, add the run call(s).**

Run calls execute in python. As such, to make shell calls when using the “run” directive, you must pass them to the “call” function. The resulting call to execute mpileup should look like Figure 10.

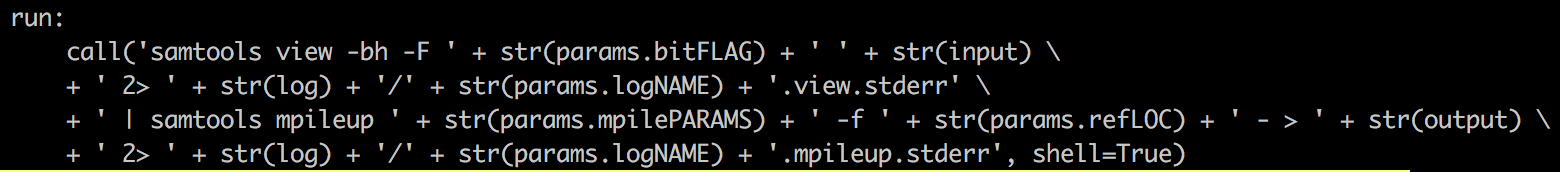
****

Figure . VIM screenshot of what the tPile\_INCLUDE run directive should look like. The standard error for both portions of the call is redirected to log files.

1. **Convert the names calls into YAML variables.**

This is to provide the option of dictating the programs used in the pipeline either via:

* Implicit program calls and a hard reliance on a managed conda environment.
* Direct program calls and a soft reliance on a managed conda environment. Meaning that all programs not provided direct calls will then be managed by the conda environment.

If your rule is named “tPile”, and the program being called is “samtools”, then the resultant YAML file configuration variable would be named “tPile\_samtoolsProg”. The YAML variables are to use the following naming convention formula:

varNAME = ruleNAME + “\_” + programNAME + “Prog”

After the transition of program names into YAML variables, the run directive should look like Figure 11.

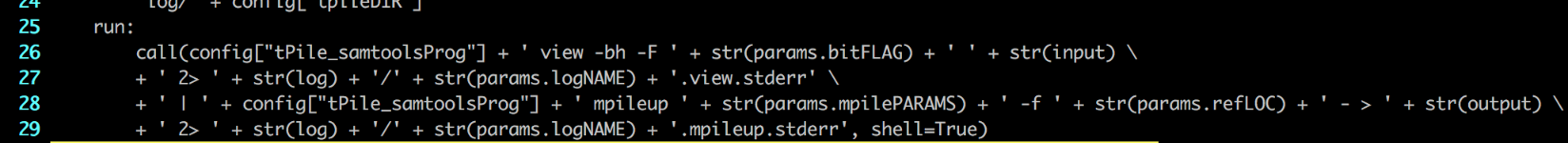
****

Figure . VIM screenshot of what the tPile\_INCLUDE run directive should look like after the ‘Step 13’ conversion of the hard-coded program calls into YAML configuration variables.

1. **Delete the extra hints at the bottom.**

At the time of reading this, they may or may not exist. An example of a finished tPile\_INCLUDE file is available in Figure 12.

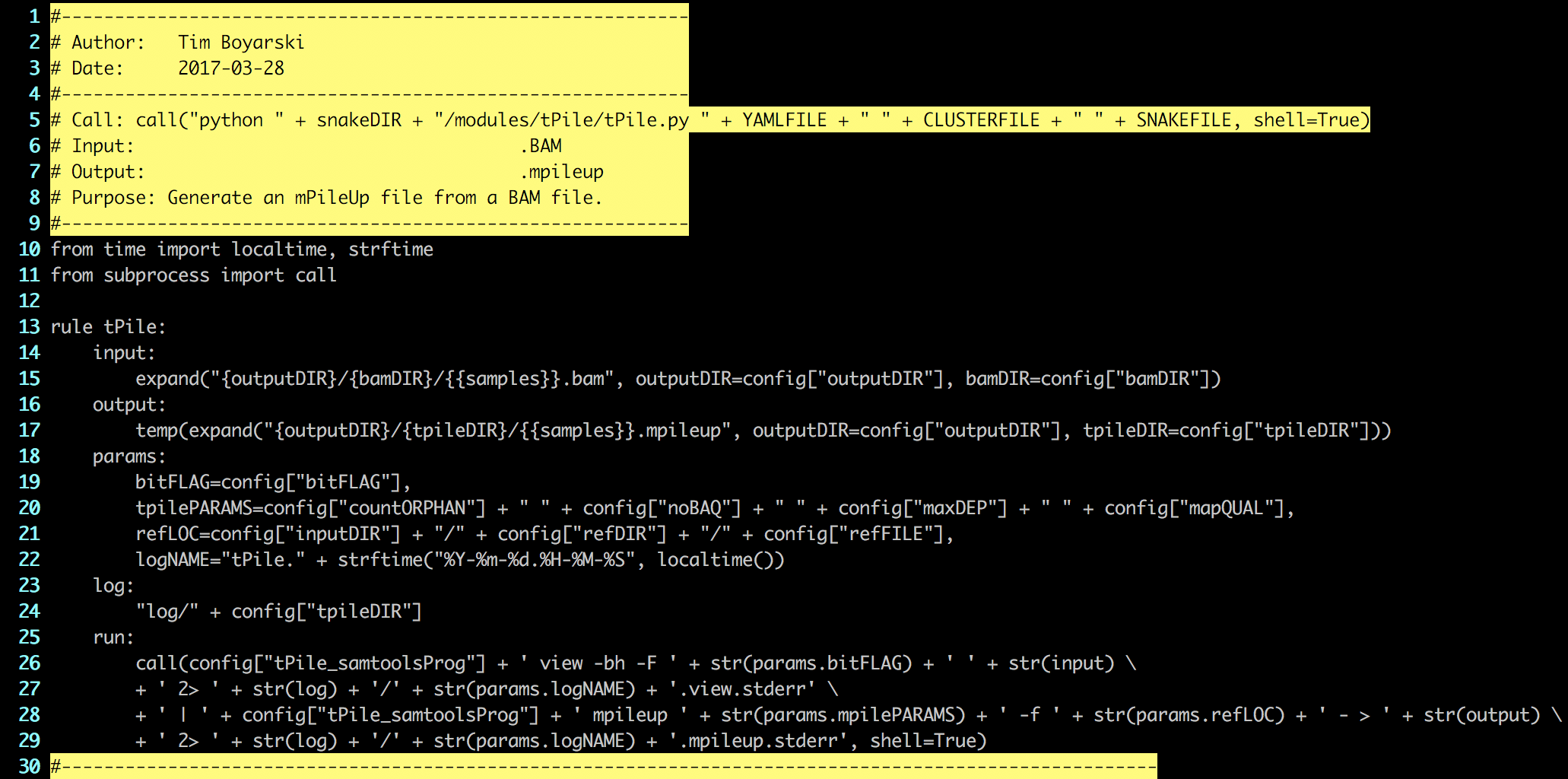


Figure . VIM screenshot of the finished tPile\_INCLUDE file should look like.

### Single Module Design - PY

The “.py” python script is the file which helps auto-populate the constructed Snakefile, ‘.YAML’, and ‘.JSON’ files. It provides the rules, calls, and the default arguments for the calls. There should only ever be one “.py” file.

1. **Using VIM, open “tPile.py”, formerly “temp.py”.**
2. **Go to Line 2 and Line 3, insert your name and the date.**
3. **Go to Line 5, edit the python call.**

Typically, this should just involve replacing the highlighted X’s with the module name, as seen in Figure 13.

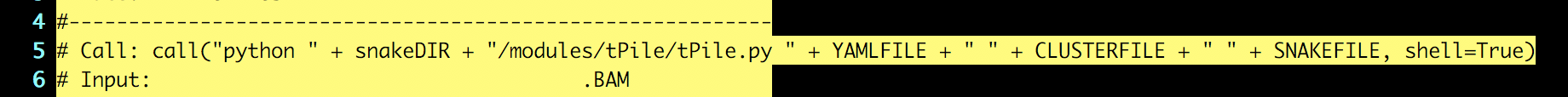


Figure . VIM screenshot for what Line 5 from the tPile.py file should look like. This will allow for easy access when needing to refer to this file while building a pipeline build file.

1. **Go to Line 6, Line 7, and Line 8, insert the input file type, output file type, and the module’s purpose.**

Typically, the purpose can be left unmodified, the default description is sufficient. Editing it may alter the line numbers being referenced.

1. **Highlight the fields to be edited by moving the cursor to “XXXXXXXX” on Line 12, and pressing “Shift + #”.**

To remove this highlighting, type “:noh” into VIM, or, eliminate it by changing all of the highlighted fields.

1. **Go to Line 19, edit the value of the variable to reflect the module name.**

This should be the same as the filename without the “.py”. It is the same term used to name the rule and the directory. The module name is used multiple times throughout when populating the Snakefile, the ‘.YAML’, and the ‘.JSON, files. The name is used for user reporting. When the python script is run, the outputs describe the actions being performed.

1. **To preserve line number references, will start from the bottom.**

Certain sections will require multiple lines of code to be organize, as such, this will push down the lines below. By starting from the bottom, we will not affect the lines until we start altering that line’s location.

1. **Go to line 79, edit the call used for this module.**

The statements required for all rules should be listed here. Use the output statements for the rules as templates. Convert the output statement {{samples}} from a wild card. It should become {samples} and have an

accompanying declaration at the end, ‘samples=config[“sample”]’. The template is shown in Figure 14, it is converted to the proper call, as seen in Figure 15.

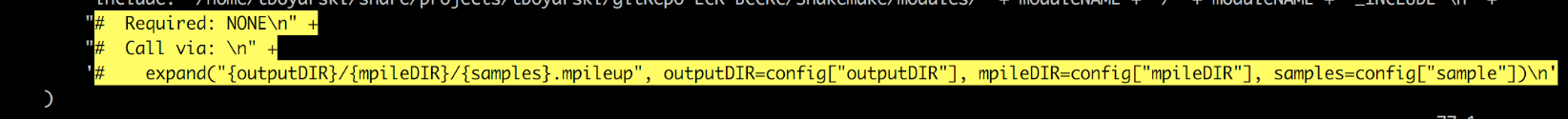


Figure . VIM screenshot of the code which replaced the template text for mPile.py

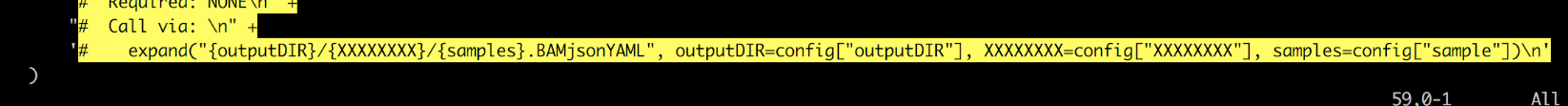


Figure . VIM screenshot of the code to be replaced inside the python file being built.

1. **Go to line Lines 74 and 75, reflect here what was listed in Lines 5 and 6 of this file.**

This output is used to provide context to what the module does when being viewed from the Snakefile.

1. **Go to Line 61, if using submodules edit this, otherwise, do not change.**

To make variable population easier in a single main module, everything is named the same as seen in Figure 16. Otherwise, if there are submodules to this unit, add the names of the rules within each submodule, as seen in Figure 17.

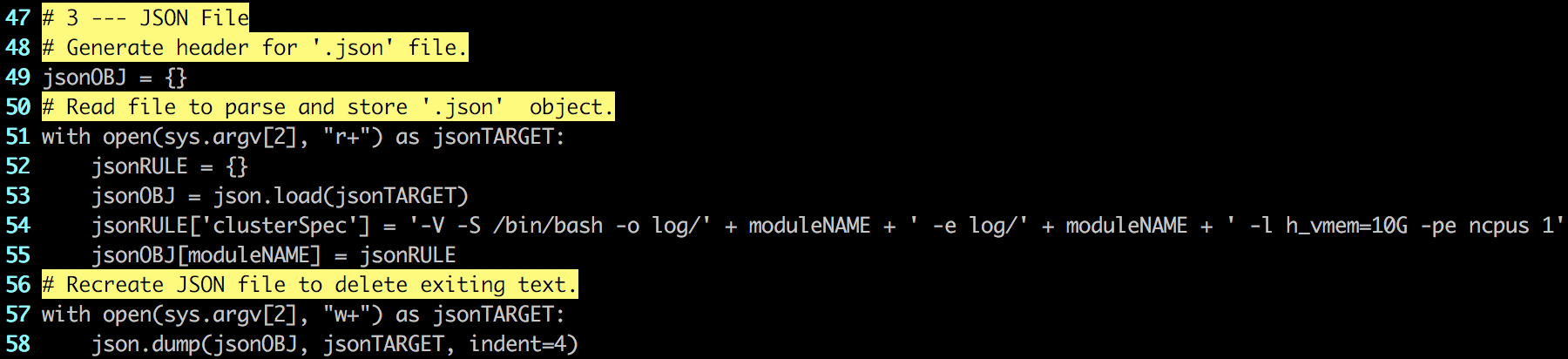


Figure . Genesis screenshot of the template structure provided. If only a single module is used, nothing needs to be changed as the "moduleNAME" variable listed at the top of the file will populate all the necessary fields. Line 61 is indicated by the green arrow.

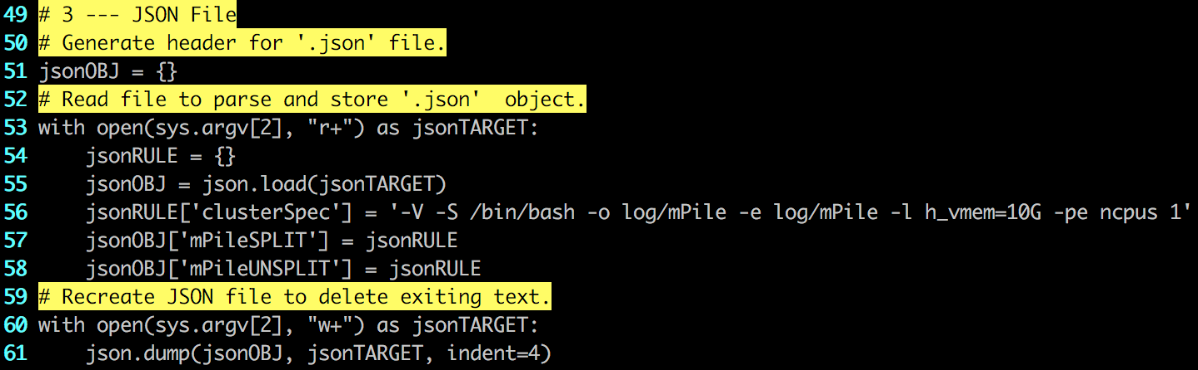


Figure . Genesis screenshot of the JSON File section from the production ready module mPile. mPile has two submodules, mPileSPLIT and mPileUNSPLIT. The green arrow highlights the listed submodules.

1. **Go to Lines 43-50, list the parameters and write them to the target YAML file.**

The parameters listed here should follow the format of line 43, and should be added to the Line 48 “.write()” statement. Add all the parameters that were used in the coding of the modules, as seen in Figure 18. This python script will populate a file in “.YAML” format, so all entries added must follow YAML syntax (see line 43).

When coding the tPile rule in our tPile\_INCLUDE file, we added the “tPile\_samtoolsProg” software argument to define which version of samtools to use (see step 13). We also defined a new output directory tpileDIR (step 8), and variables bitFLAG, countORPHAN, noBAQ, maxDEP, and mapQUAL (step 10). The other config variables we reference in tPile\_INCLUDE are global variables, and thus do not need to be re-explained here. Existing examples can be see below.

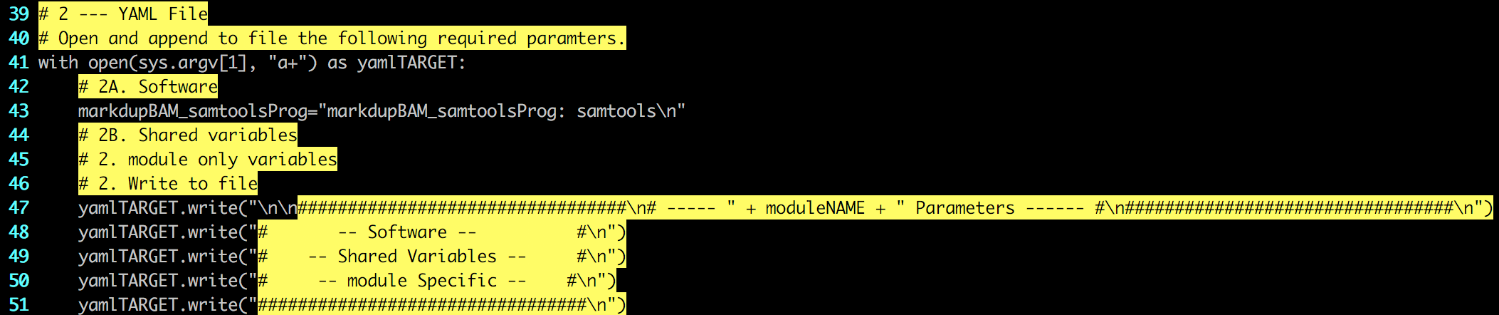


Figure . VIM Screenshot of the "temp.py" file in the Template directory. Shows the section of code that is to be edited.

When writing to file, the variable names are concatenated to the string as one would normally expect. Please ensure all variables are written with a new-line character at the end of the string. The newline character is to ensure proper formatting and parsing of the YAML file being written.

var1=”var1: X\n”

var1=”var2: X\n”

yamlTARGET.write(“ … … #\n” + var1 + var2)

When determining software program variable names, use the rule, the program, and the string “Prog”.

|  |  |  |
| --- | --- | --- |
| Rule | Program | String |
| tPile | samtools | Prog |
| Result | tPile\_samtoolsProg | |

Table . Depiction of how to create a named variable for a program within the rule.

1. **When finished, the YAML file section should look like Figure 19.**

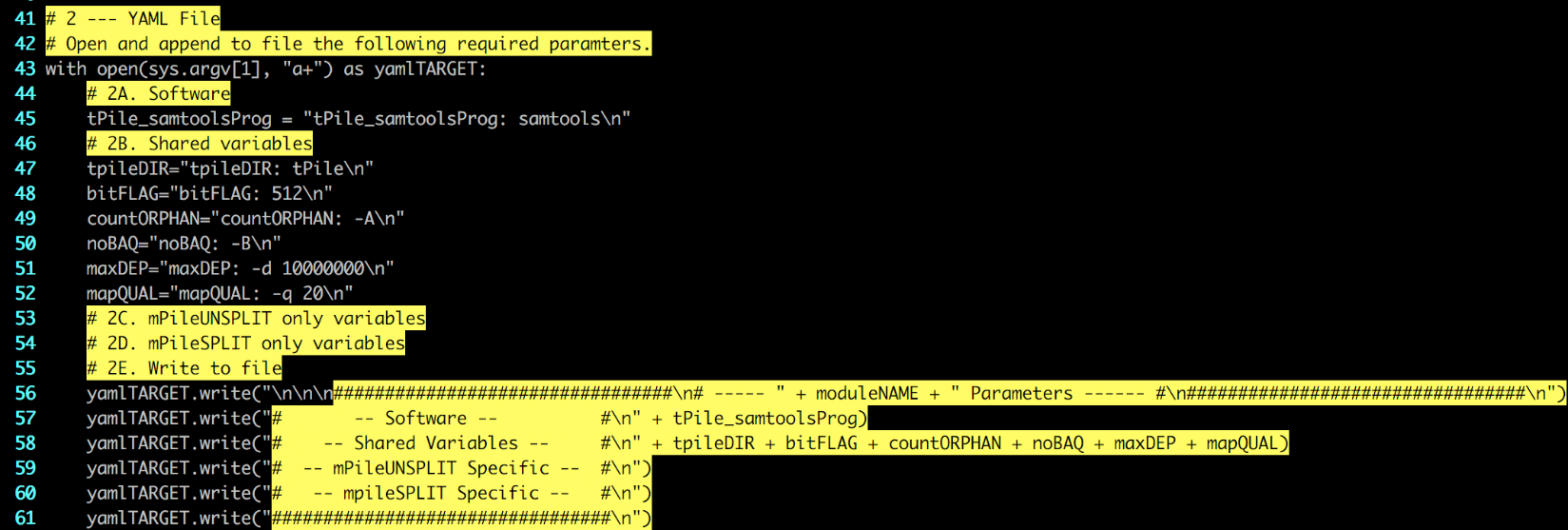


Figure . VIM Screenshot of the YAML file section within "tPile.py".

1. **When finished, “tPile.py” should be the same as Figure 20.**

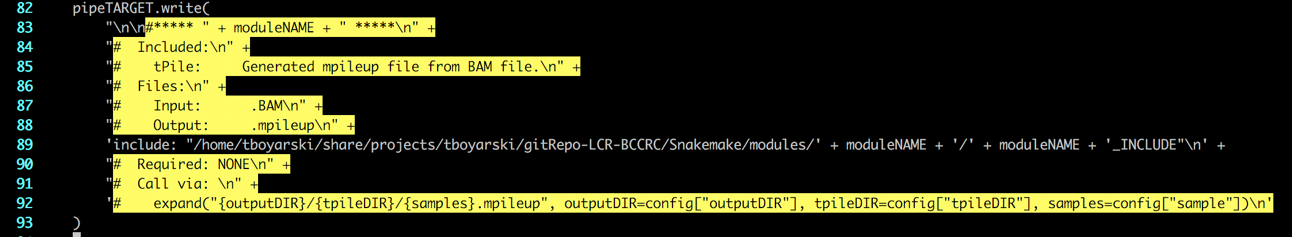


Figure . VIM Screenshot of the production ready “tPile.py” script.

### Single Module Design - README

The README.md file is used as a point of reference for users on Genesis, and when viewing the repository on GitHub. The comments can be used as guidance on what to write.

1. **Module (Snakemake)**

Provide the name of the module and the program language in which its written in. This description describes the type of module, not what it does.

1. **Modules**

List each of the modules and provide a quick description as to the purpose of each. Full length descriptions should be contained in the specific module file.

1. **Logging**

Just change the directory in which it is being stored. The directory will be named the same as this module’s.

1. **Global Directories**

All of the global directories and their descriptions are already listed, for ease of access. Delete the ones that this module will not require.

1. **Global Parameters**

All of the global parameters and their descriptions are already listed, for ease of access. Delete the ones that this module will not require. Note the four-column table layout, as exemplified in Figure 21 and Figure 22.

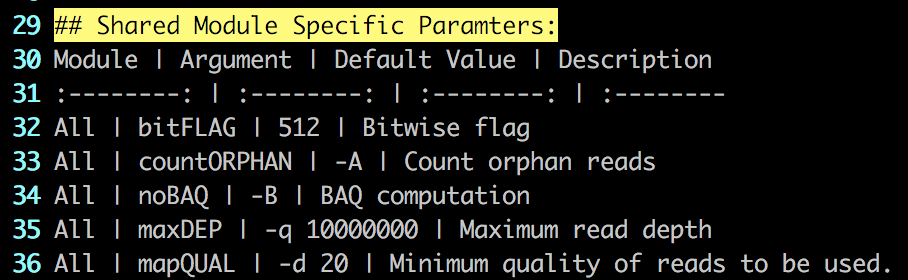


Figure . Genesis screenshot of the mPile module README.md. This code shows structure of the four-column markdown table. Column positions are determined by the colon's in Line 31.

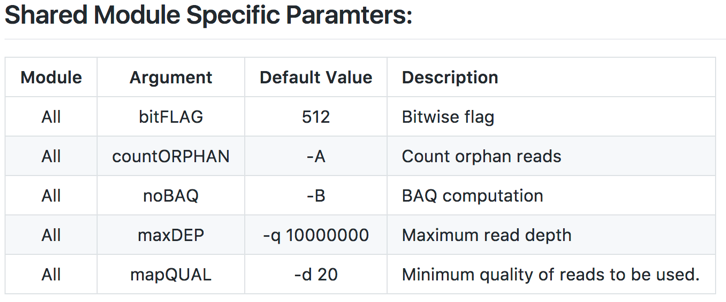


Figure . GitHub screenshot of the mPile module README.md. The image shows how the table format code is represented visually.

1. **Module Specific Parameters**

The module specific parameters and their descriptions are to be listed here. Follow the formatting of Step 5.

1. **When finished, “README.md” should look similar to Figure 23.**

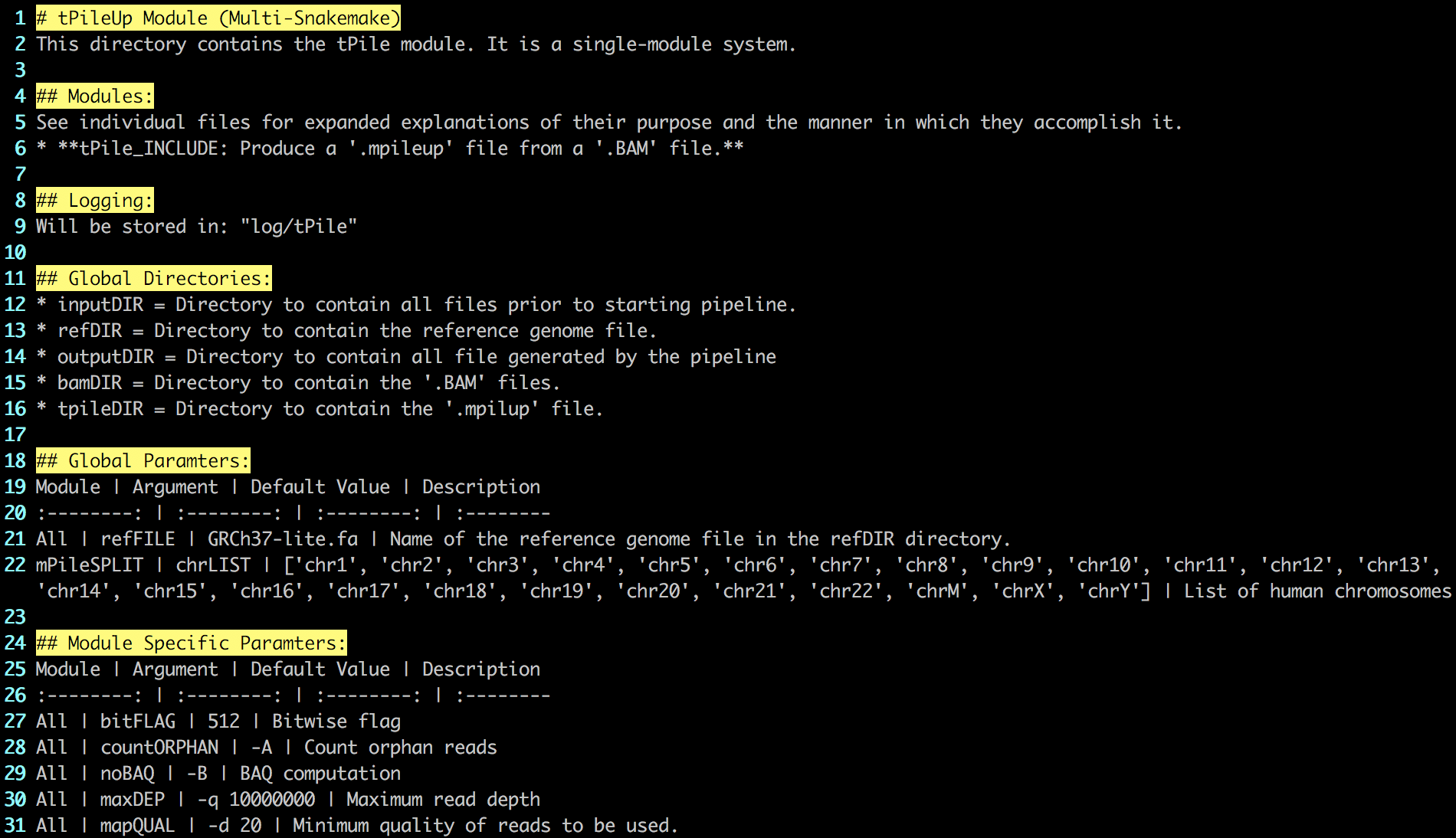


Figure . VIM screenshot of the README.md file for the tPile module.

## Multiple Module Design - Intro

A key focus of module design is cohesion, keeping modules small and focused.

* Primary calls should all be separated into a different module and are those which run programs.
* Secondary calls support a primary call include: print statements, directory creation, input redirection, parsing commands, and other administrative functions.

The following section explains the file setup for a multi-module system.

1. **In the modules directory, copy “Template/” and name the copy with the module name.**

Figure 24 shows the contents inside the modules directory at the time of writing this document. The Template directory to be copied has been indicated by a yellow box.

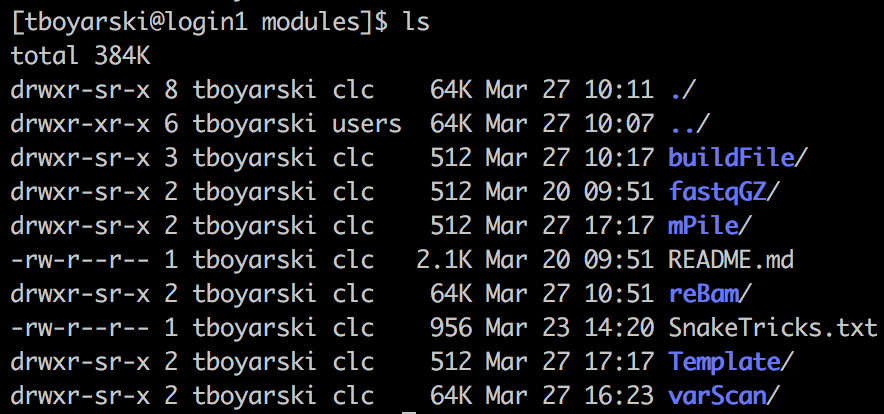
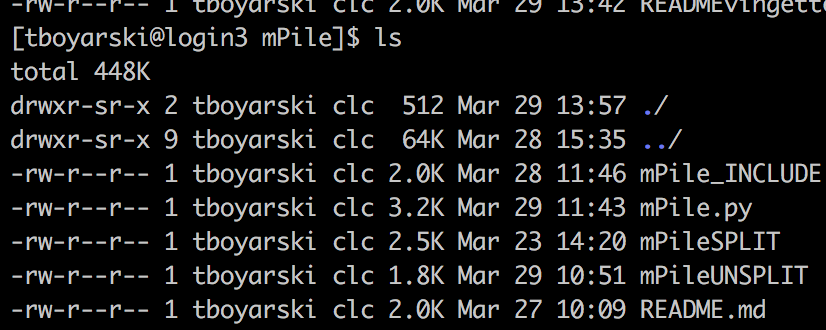


Figure . Genesis screenshot of the modules directory. Highlighted in the yellow box is the 'Template' directory users have been instructed to copy.

1. **Enter the renamed module directory and rename the files to reflect the module name.**

In a multi-rule module, each file containing a rule should be created using the template file “temp\_subINCLUDE”. In this tutorial, we will create two submodules that run mpileup either on the whole genome (tPileUNSPLIT) or separated by chromosome (tPileSPLIT). The module-level “\_INCLUDE” file will be based on template file “temp\_masterINCLUDE”. The python file and the “\_INCLUDE” Snakefile should be named in accordance with the module directory name; this results in the files being named “tPile.py” and “tPile\_INCLUDE”, respectively. For clarification in naming, please refer to Table 1. Figure 25 shows the starting structure of the template directory. Figure 26 shows the file structure of the deployed mPile module (analogous to what we will create for tPile).



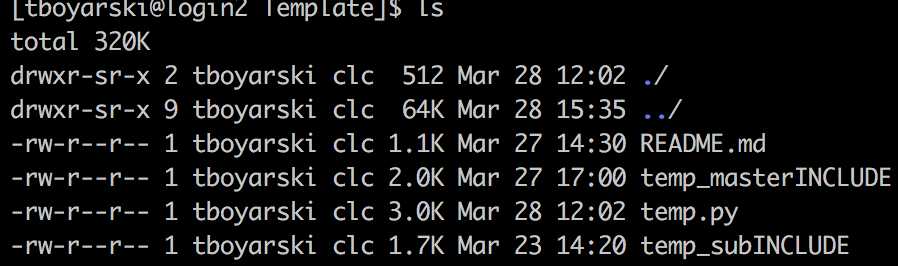


Figure . Genesis screenshot of the contents of the Template directory.

Figure . Genesis screenshot of the contents of the mPile directory after the files have been named correctly.

|  |  |  |
| --- | --- | --- |
| Template Name | Action | Final Name |
| README.md | No action | README.md |
| temp\_masterINCLUDE | Renamed | tPile\_INCLUDE |
| temp.py | Renamed | tPile.py |
| temp\_subINCLUDE | Copied | tPileUNSPLIT |
| temp\_subINCLUDE | Renamed | tPileSPLIT |

Table . An example of the naming convention which could be used when expanding the tutorial module tPile.

### Multiple Module Design - INCLUDE

Submodules can be built using the “Single Module Design – INCLUDE” guide. The only difference is the name of the file and the contained rule, as per Table 3 in the previous section.

In this example, the two submodule names will be tPileUNSPLIT and tPileSPLIT. The directory name is reserved to always carry the “\_INCLUDE” tag. Submodules will not carry the “\_INCLUDE” suffix.

1. **Using VIM, open “tPile\_INLCUDE”.**
2. **Go to Line 2 and 3, add your name and the date.**
3. **Go to Line 5, edit the path to reach the python file.**

The module should be inside a directory with the same name. Replace “XXXXXXX” with the name of the module.

1. **We will now start from the bottom, as to preserve number lines referenced.**

Certain sections will require multiple lines of code to be organize, as such, this will push down the lines below. By starting from the bottom, we will not affect the lines until we start altering that line’s location.

1. **Go to Line 20/21 and Line 25/26, and replace “moduleNAME1” and “moduleNAME2” with the actual module names “tPileUNSPLIT” and “tPileSPLIT”, respectively. Remove lines 30-34 since we will only have two submodules for tPile.**
2. **Go to Line 22/23 and Line 27/28, list the submodule-specific variables.**

Provide a brief description of what the parameter represents. If there are no parameters, list None.

1. **Figure 27 demonstrates a finished two-submodule include section.**

If additional modules were needed, repeat the format and structure of the two modules just created. For readability, separate by a blank line.

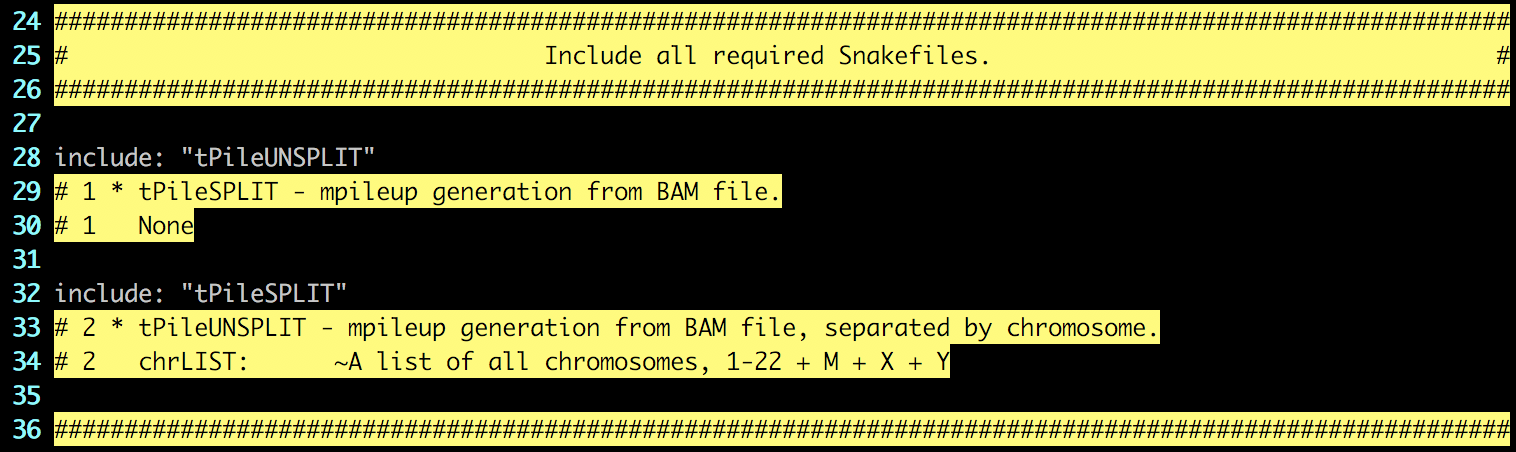


Figure . Genesis screenshot of the tPile\_INCLUDE file. The module tPileUNSPLIT, included on Line 28, does not have any submodule-specific variables. The module tPileSPLIT, included on line 32, is the only submodule which requires the list of chromosomes, chrLIST.

1. **Go to Line 12, list all parameters required by the module.**

These are the ‘.YAML’ configuration file global and shared parameters.

1. **Go to Lines 6-8, update the input, output and purpose.**

This should describe the starting and ending points of the module and a very high level.

1. **When finished, “tPile\_INCLUDE” should look similar to Figure 28.**



Figure . VIM screenshot of the finalized "tPile\_INCLUDE".

### Multiple Module Design - PY

Same as Single Module Design – PY. Make sure to add jsonOBJ entries for each submodule (step 10).

### Multiple Module Design - README

Same as Single Module Design – README.

## References

Köster, J., & Rahmann, S. (2012). Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, *28*(19), 2520-2522. [Available at: https://pypi.python.org/pypi/snakemake]

Python Software Foundation (2017) Python. [Available at: https://www.python.org/]

## Appendices

Clean.sh

This file is used when developing modules. It assists in cleaning out the working environment of all produced files and outputs so that both the python build file and the Snakemake calls can be performed again. This file is a convenience function and otherwise serves no purpose for this pipeline.

#!/bin/bash

rm -rf output/

rm -rf log/

rm -rf .snakemake/

rm Snakefile

rm input/config.yaml

rm input/config.json