

# Snakemake Vignette: Intro to Modules

Written By: Tim Boyarski, BSc  
With assistance from: Lauren Chong, MSc  
For: BC Cancer Agency – Lymphoid Cancer Research  
Date of: August 31<sup>st</sup>, 2017

## Introduction

The following vignette is to be used as an introduction to the understanding and development of Snakemake modules for the newly designed LCR Snakemake Pipeline System. This is the second Snakemake vignette, and builds from knowledge developed in the previous document, “Intro to Pipelines”. This document provides a step-wise guide how to create and integrate a Snakemake module, called “tutorialGen”. The tutorial is to reproduce the existing production ready module called “mpileupGen” and is to provide a first-hand experience in developing a module for the Snakemake pipeline system.

The vignette is applicable for users regardless whether they are running the pipeline locally or on a high-performance computing cluster. It is intended for audiences who are savvy computer programmers and are also very familiar with the existing Snakemake Pipeline System. Basic syntax is not robustly covered within this documentation. An extended language description can be found in the [Snakemake documentation](#).

**\*\*WARNING\*\*** *This vignette is based-off of “mpileupGen”, a module that already exists. Do not refer to this module if you are stuck in this vignette. A sample key is provided, “tutorialGen”, which can be found inside of the “TEMPLATE” directory. This vignette follows the creation of a simplified version of the “bam2mpileup” file, which will be called “tPile”.*

A completed copy of “tutorialGen” is stored inside the module’s “Template/” directory in order to:

- 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 “mpileupGen” module.

Table 1. Documenting the changes to this vignette document

Date	Author	Contact	Changes
2017-08-31	tboyarski	tim.boyarski@gmail.com	Final Version

## Contents

Introduction.....	1
I. Terminology (Same as in “Intro to Pipelines).....	3
II. Pre-Conditions (Same as in “Intro to Pipelines) .....	4
III. Snakemake module organization (Same as in “Intro to Pipelines) .....	5
IV. TEMPLATE module Overview .....	6
V. Module Design – Rationale and Setup .....	7
VI. Module Design – Configuring “tutorialGen_INCLUDE” .....	8
VII. Module Design – Configuring “tPile” .....	10
VIII. Module Design – Configuring “tutorialGen.py” .....	15
IX. Module Design – Configuring the “README.md” .....	19
X. Multiple Module Design - Intro .....	<b>Error! Bookmark not defined.</b>
XI. References.....	20
XII. Appendices .....	20

## I. Terminology

- **Workspace:** This term refers to the current, or potential, directory where the pipeline is located.
- **Module Names:** Typically start with the format of the output file, then they include the purpose(s), and lastly if needed, the software used by all sub-modules.
  - A `vcfGenUtil_varScan` module uses is able to generate or utilize '.vcf' files, and all its operations are done using the processing functionality of VarScan.

The most commonly used categories used for naming are:

- Gen - Ability to output files which are of a format than the inputs.
  - Util - Ability to manipulate files without changing their format or summaries files.
  - Annotate - Ability to annotate existing files without changing their format.
  - Metrics - Ability to output files providing metrics on existing data.
- **Sub-modules:** Typically, an action verb and a file format, sub-modules are named by what they specifically accomplish. If two rules have the same functionality but accomplish it using different software, then the sub-modules can be suffix'ed with an '\_', and then the software name.
    - `mergeVCF` – merges '.vcf' files
    - `sortBAM_bioambam` – sorts '.bam' file using biobambam
    - `sortBAM_samtools` – sorts '.bam' file using samtools

**\*\*NOTE\*\*** *The alignment sub-modules contained within "bamGen" do not follow this nomenclature.*

Complex modules are decomposed into smaller, rule-containing, sub-modules. We highly encourage that these small rules are separated into their own file. Rules are to be collectively gathered into the module-specific '\_INCLUDE' file which should only contain comments, control flow of internal sub-modules, and the sub-module 'include:' statements. Control flow is achieved using conditionals and "ruleorder". This control flow allows for the simplistic swapping of equivalent rules running different software, and to set rule precedence when rules are ambiguous because they are capable of producing the same output file.

If not already available, the modules are to be copied from online from the "LCR-BCCRC/workflow\_exploration" private repository. Please download the entire Snakemake directory as it contains the modules directory, and this vignette expects "modules" to be a sub-directory. A direct link to the repository can be found below:

Private: [https://github.com/LCR-BCCRC/workflow\\_exploration](https://github.com/LCR-BCCRC/workflow_exploration)

The private repository is actively maintained. For the purpose of transparency and accessibility, a public facing repository has been made available. It will not be actively maintained, and it reflects the state of the private code base at the end of development by the initial author. For those with access, please utilize the private repository link above and get the most up-to-date vignettes. In the event you are not apart of the BC Cancer Agency and do not have access to the Lymphoid Cancer Research Departments private repository, please refer to this static copy of the repository and vignettes, which can be found a public repository hosted on the GitHub account of the initial author:

Public: <https://github.com/tboyarski/BCCRC-Snakemake>

## II. Pre-Conditions

- 1) This vignette was validated using the core software listed in Table 2 below. The vignette will likely also work with the most up-to-date versions of the software listed, however, this is not guaranteed.

*Table 2. Software versions utilized during the creation of this vignette. Newer versions are likely not to have redacted existing functionality, as such, it is highly likely that this vignette is compatible with the most up-to-date versions of the software listed.*

Software	Vignette Version	Theoretical Compatibility
CentOS	5	6
Python	3.5	$\geq 3.5$
Snakemake	3.10.1	$\geq 3.10$
DRMAA	0.7.6	$\geq 0.7.6$

- 2) The conda environment was created using the following '.YML' file. Except for the installation of Snakemake and Python, the dependencies listed are software required for the successful execution and processing of the finalized pipeline; however, they are not required for the generation of the Snakemake pipeline.

name: CentOS5-Compatible

channels:

- bioconda
- defaults

dependencies:

- biobambam=2.0.39=0
- bwa=0.7.13=0
- java-jdk=8.0.92=0
- picard=2.7.1=py35\_1
- samtools=0.1.18=0
- snakemake=3.12.1=py35\_1
- snpeff=4.1l=0
- varscan=2.4.0=0
- drmaa=0.7.6=py35\_0
- python=3.5.1=5

- 3) All white space formatting must be performed exclusively with spaces. The system requires that users do not use tab-indentation when editing or writing code. This can be established on a UNIX system by setting the following conditions in your '.vimrc' file.

To use have tabs automatically converted into spaces:

set expandtab

To set the width of the tabs which are automatically converted into spaces:

set shiftwidth=4

set tabstop=4

- 4) All software calls must exist as YAML variables to provide the option of dictating the programs used in the pipeline either implicitly via reliance on a managed conda environment, or directly via an absolute path. The YAML variables are to use the following naming convention formula:

varNAME = moduleNAME + "\_" + programNAME + "Prog"

### III. Snakemake module organization

All Snakemake modules are to be stored within the modules sub-directory of the Snakemake system. 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 four files. The directory structure is outlined in Figure 1, and below this is an explanation as to the purpose of each of the four file categories.

```
[tboyarski@login4 modules]$ tree -L 2
.
|-- [tboyarsk      2193 Jun 30 19:49] README.md
|-- [tboyarsk      512 Aug 29 11:45] TEMPLATE
|   |-- [tboyarsk    2278 Jul 19 18:36] README.md
|   |-- [tboyarsk     512 Jul  5 18:17] RegTestReadMeTemplate
|   |-- [tboyarsk   12602 Jun 16 13:09] SnakeTricks.md
|   |-- [tboyarsk    6263 Jul 10 20:40] temp.py
|   |-- [tboyarsk    1642 Jul 21 12:02] temp_masterINCLUDE
|   |-- [tboyarsk    6357 Jul 21 15:43] temp_subINCLUDE
|   `-- [tboyarsk     512 Apr 21 16:09] tutorialGen
|-- [tboyarsk      512 Aug 10 16:41] bamGen
|   |-- [tboyarsk    5036 Aug 10 16:41] README.md
|   |-- [tboyarsk     512 Jul 10 10:26] RegTestReadMe
|   |-- [tboyarsk    7649 Aug 10 16:41] bamALIGN_bwa
|   |-- [tboyarsk    6350 Aug 10 16:47] bamALIGN_star
|   |-- [tboyarsk   10106 Aug 10 16:56] bamGen.py
|   |-- [tboyarsk    2437 Aug 10 16:41] bamGen_INCLUDE
|   `-- [tboyarsk    2623 Aug 11 15:44] sam2BAM
|-- [tboyarsk   65536 Jul  7 17:01] bamMetrics
|   |-- [tboyarsk    2565 Jul  7 16:53] README.md
|   |-- [tboyarsk   65536 Jul  7 20:57] RegTestREADME
|   |-- [tboyarsk   12823 Aug 11 14:50] bamMetrics.py
|   |-- [tboyarsk    1800 Jun  9 18:01] bamMetrics_INCLUDE
|   |-- [tboyarsk    3225 Jul  7 16:44] collectGCBias
|   |-- [tboyarsk    7331 Jul 10 20:32] collectMERGE_ADAPTOR
|   |-- [tboyarsk    3959 Jul  7 16:48] collectMultMetrics
|   |-- [tboyarsk    4552 Jul 12 12:24] collectRNASeq
|   |-- [tboyarsk    2953 Jul  7 16:51] collectWGS
|   |-- [tboyarsk    2721 Jul  7 16:52] flagStats
|   `-- [tboyarsk    2624 Jul  7 16:53] readLen
```

Figure 1. Tree structure of the beginning portion of 'modules/', as of August 29th, 2017. Exemplifies that each module contains the four core files. Please note that 'TEMPLATE/' contains the 'tutorialGen/' directory for the module vignette.

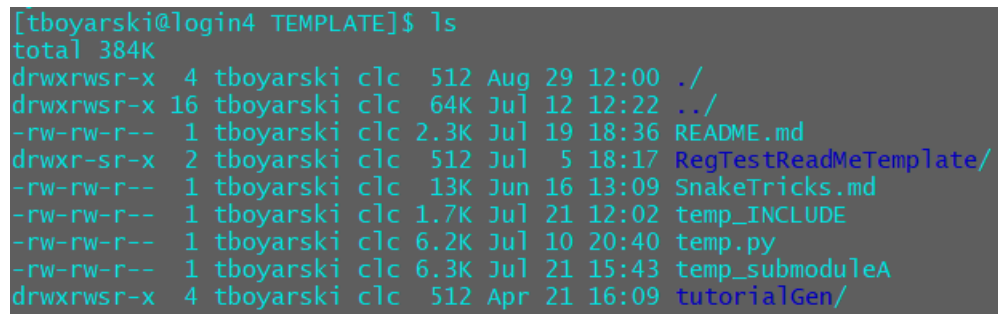
#### Module Core files:

- **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 Snakemake file to be included into the pipeline. It contains no rules; rather, it contains submodule 'include:' statements and a significant description for each submodule contained. It may also contain control flow in the form of python conditionals or Snakemake "ruleorder".
- **subModuleA** – Isolation of a single rule, with the potential to possess supporting functions for the Snakemake input directive or for python processing elsewhere during execution.

#### IV. TEMPLATE module Overview

An example of the directory structure of the “tutorialGen” module can be seen in Figure 2. This vignette builds from the files provided in the module “TEMPLATE”. This is how an experienced user would create a new module, as such, the same actions are performed in this vignette.

**\*\*WARNING\*\*** This vignette is based-off of “mpileupGen”, a module that already exists. Do not refer to this module if you are stuck in this vignette. A sample key is provided, “tutorialGen”, which can be found inside of the copied directory. This vignette is having you create a simplified version of the “bam2mpileup”, called “tPile”.



```
[tboyarski@login4 TEMPLATE]$ ls
total 384K
drwxrwsr-x  4 tboyarski clc  512 Aug 29 12:00 ./
drwxrwsr-x 16 tboyarski clc  64K Jul 12 12:22 ../
-rw-rw-r--  1 tboyarski clc 2.3K Jul 19 18:36 README.md
drwxr-sr-x  2 tboyarski clc  512 Jul  5 18:17 RegTestReadMeTemplate/
-rw-rw-r--  1 tboyarski clc  13K Jun 16 13:09 SnakeTricks.md
-rw-rw-r--  1 tboyarski clc  1.7K Jul 21 12:02 temp_INCLUDE
-rw-rw-r--  1 tboyarski clc  6.2K Jul 10 20:40 temp.py
-rw-rw-r--  1 tboyarski clc  6.3K Jul 21 15:43 temp_submoduleA
drwxrwsr-x  4 tboyarski clc  512 Apr 21 16:09 tutorialGen/
```

Figure 2. VIM screenshot of the current Template/ directory setup on Genesis. Note that the Template/ directory contains an answer key for this tutorial.

#### TEMPLATE Module files:

- **README.md** – A read-me file written in markdown format.
- **RegTestReadMeTemplate** – Directory of template documents to fill in when regression testing a new module. It structures the required information to replicate the modules functionality.
- **SnakeTricks.md** – A collection of thoughts and discoveries made during the development of the pipeline system. Talks about proactive solutions to avoiding cyclical dependencies and potential design hurdles.
- **temp\_INCLUDE** – The Snakemake file to be included into the pipeline. It contains no rules; rather, it contains submodule ‘include:’ statements and a significant description for each submodule contained. It may also contain control flow in the form of python conditionals or Snakemake “ruleorder”.
- **temp.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.
- **temp\_submoduleA** – Isolation of a single rule, with the potential to possess supporting functions for the Snakemake input directive or for python processing elsewhere during execution.
- **tutorialGen** – Directory containing the expected result for this tutorial.

## V. Module Design – Rationale and Setup

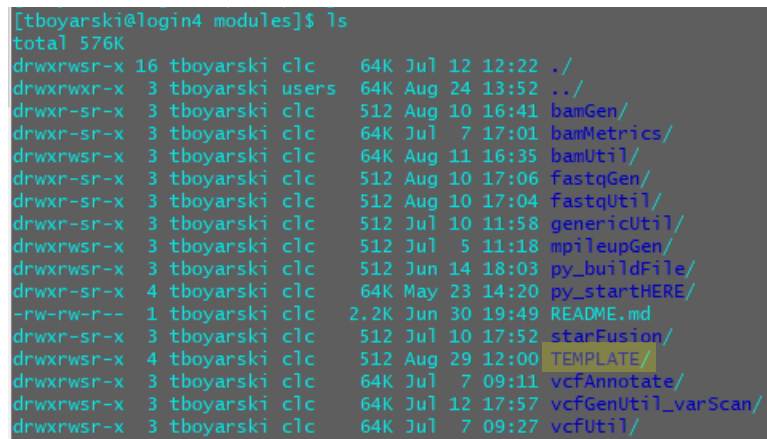
We will be creating a module which accepts a ‘.bam’ file, and produces a ‘.mpileup’ file. The files it produces may be at chromosomal granularity if required. The rule will require the use of [Samtools](#). A key focus of module design is cohesion, keeping modules small and focused. They should contain only one primary call, and a limited number of secondary calls.

- 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, can be contained within a rule but must be named uniquely as to prevent conflict with future supporting functions. The current approach is to prefix, when possible, using the rule name (E.g. a parsing function called “tutorialGen\_parseName”).

### 1) In your local checkout of the modules directory, copy and rename “TEMPLATE/” to “tutorialGen”.

Figure 4 shows the contents inside the modules directory at the time of writing this document. As indicated by the yellow box, copy the TEMPALTE/, and name the new directory “tutorialGen”. Copy the folder using:

```
$cp -rf TEMPLATE/ tutorialGen
```



```
[tboyarski@login4 modules]$ ls
total 576K
drwxrwsr-x 16 tboyarski c1c    64K Jul 12 12:22 ./
drwxrwsr-x  3 tboyarski users  64K Aug 24 13:52 ../
drwxr-sr-x  3 tboyarski c1c    512 Aug 10 16:41 bamGen/
drwxr-sr-x  3 tboyarski c1c    64K Jul  7 17:01 bamMetrics/
drwxrwsr-x  3 tboyarski c1c    64K Aug 11 16:35 bamUtil/
drwxr-sr-x  3 tboyarski c1c    512 Aug 10 17:06 fastqGen/
drwxr-sr-x  3 tboyarski c1c    512 Aug 10 17:04 fastqUtil/
drwxr-sr-x  3 tboyarski c1c    512 Jul 10 11:58 genericUtil/
drwxrwsr-x  3 tboyarski c1c    512 Jul  5 11:18 mpileupGen/
drwxrwsr-x  3 tboyarski c1c    512 Jun 14 18:03 py_buildFile/
drwxr-sr-x  4 tboyarski c1c    64K May 23 14:20 py_startHERE/
-rw-rw-r--  1 tboyarski c1c    2.2K Jun 30 19:49 README.md
drwxr-sr-x  3 tboyarski c1c    512 Jul 10 17:52 starFusion/
drwxrwsr-x  4 tboyarski c1c    512 Aug 29 12:00 TEMPLATE/
drwxrwsr-x  3 tboyarski c1c    64K Jul  7 09:11 vcfAnnotate/
drwxrwsr-x  3 tboyarski c1c    64K Jul 12 17:57 vcfGenUtil_varScan/
drwxrwsr-x  3 tboyarski c1c    64K Jul  7 09:27 vcfUtil/
```

Figure 3. Genesis screenshot of the modules directory. Highlighted in the yellow box is the 'Template' directory users have been instructed to copy.

### 2) Enter the renamed module directory and rename the files to reflect the module name.

For clarification in naming, please refer to Table 1. The python file and the “\_INCLUDE” file should be named in accordance with the module directory name. This results in the files being named “turotialGen.py” and “tutorialGen\_INCLUDE”, respectively. The submodule “tPile” we are creating will mimic the real submodule “bam2mpileup” inside of “mpileupGen”.

Directory item	Action	Final Name
README.md	No action	README.md
temp_INCLUDE	Renamed	tutorialGen_INCLUDE
temp.py	Renamed	tutorialGen.py
temp_submoduleA	Renamed	tPile
turotialGen/	No action	~ This is the answer key ~
Snaketricks.md	Delete	**Not required by new modules**
RegTestReadMeTemplate	Ignore	**Used to Test Module**

Table 3. A conversion table to assist in when renaming files from their template names. Names were chosen in accordance with the development of the “mpileupGen” module.

## VI. Module Design – Configuring “tutorialGen\_INCLUDE”

The “\_INCLUDE” file provides users the ability to introduced conditionals to fine-tune control flow, and to introduce “ruleorder” to resolve ambiguity between rules. A module will always contain an “\_INCLUDE” file.

1) **Using VIM, open “tutorialGen\_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 “XXXXXXXX” with “tutorialGen”.

4) **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.

5) **Go to Line 27/28, delete this commented code.**

We will only be adding one module in this vignette, so the other include statement is not needed, nor is the module description.

6) **Go to Line 25 and write a brief description of what the submodule does.**

Provide a brief description of the purpose of the submodule.

7) **Go to Line 24 and replace “moduleNAME1” with the actual module name “tPile.**

This is where a submodule is included, using an “include:” statement and then the path to the rule file.

8) ***\*\*Optional\*\** Go to Line 19, uncomment “ruleorder” and resolve ambiguity between rules.**

This is only applicable to modules containing multiple submodules.

9) ***\*\*Optional\*\** Go to Line 16, write directions on how the modules must be executed.**

This is only applicable to modules containing multiple submodules. Use Lines 13-15 for discussion topics.

10) **Go to Line 7, describe the purpose of the module.**

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

11) **When finished, “tPile\_INCLUDE” should look similar to Figure 4.**



```

1 #-----
2 # Author:   Tim Boyarski
3 # Date:     2017-06-14
4 #-----
5 # Call: call("python " + snakeDIR + "/modules/tutorialGen/tutorialGen.py " + YAMLFILE + " " + CLUSTERFILE + " " + SNAKEFILE, shell=True)
6 #
7 # Purpose: Vignette file with no control flow..
8 #-----
9
10 #####
11 #                               Usage Guidelines                               #
12 #####
13 # Only needed if there is a specific order the modules must be executed or used in.
14 #     (E.g. processBam uses this because of its specific requirements on the first two and
15 #         last two modules which need to be called)
16
17
18 # Eliminate ambiguity as both are able to produce same output. Want to sort before merging
19 #ruleorder: moduleNAME1 > moduleNAME2
20 #####
21 #                               Include all required Snakefiles.                               #
22 #####
23
24 include: "tPile"
25 # tPile - A module used in a vignette for users new to writing modules. Serves no functional purpose.
26
27 #####

```

Figure 4. VIM screenshot of what the resultant "tutorialGen\_INCLUDE" file should look like.

## VII. Module Design – Configuring “tPile”

The submodules are typically the hardest to write and configure. There is core structure which is shared across all rules; however, often rules will require specific functionality and optional structure to optimally accomplish what is required, all while keeping system coupling low. Areas of conflict most often revolve around wildcard conflicts; as such, it is highly recommended to use “wildcard\_constraints” as needed.

Snakemake Tutorial: Constraining wildcards

[http://snakemake.readthedocs.io/en/latest/tutorial/additional\\_features.html#constraining-wildcards](http://snakemake.readthedocs.io/en/latest/tutorial/additional_features.html#constraining-wildcards)

- 1) **Using VIM, open “tPile”.**
- 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 “tutorialGen”.
- 4) **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.
- 5) **\*\*Optional\*\* Read and delete lines below the “Snakemake Rule” section, Lines 75+.**  
This is highly recommended. The information is provided or users who may need to utilize similar calls which already existing for other rules.
- 6) **We will start by editing the Snakemake rule in the “Snakemake Rule” section, on line 47.**  
Within this section we will reference rules by there directive, and when necessary but further sub-criteria (E.g. variable name, sub-section). A detailed explanation of Snakemake directives can be found [here](#).

### Directives:

- **input** – Lists rule’s expected input files.
- **output** – Lists rule’s expected output files.
- **wildcard\_constraints** – Lists regular expression constraints for rule-specific wildcards.
- **params** – Allows for the easy establishment of rule-specific variables.
- **log** – Allows for the easy establishment of rule-specific logging.
- **run** – Allows for python-based commands to be executed upon rule execution.

Within the input directive, there are three basic input types, strings, expand statements, and inline functions. A brief description is provided below, a further description can be found in the [Snakemake documentation](#).

Designing the input and the output directives are the most difficult. This is largely due to the use of wildcards, and the greedy expression matching that they utilize by default.

**\*\*NOTE\*\*** *It is important to explicitly list all required input files required, but not created, by Snakeamake (E.g. reference files) so that Snakemake make can check for them at the beginning of the pipeline’s execution.*

**\*\*NOTE\*\*** *Except for the run directive, directive listings must be separated by a comma.*

A further explanation of the input directive and some examples when using it are provided below:

**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. The method signature only accepts a single formal parameter, “wildcards”. The return of string values which have wildcards which need to be evaluated by Snakemake must be unpacked using the [“unpack\(\)” function](#).

**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](#).

**Strings:** These provide basic functionality to utilize Snakemake process dependency without the ability to specify additional variables. Considered to be the most generalizable of the three input types.

**Wildcards:** Input Expand - Indicated by double {} E.G. `expand("{}{namedWildcard}")`  
Input String - Indicated by single {} E.G. `"{}{namedWildcard}"`  
Wildcards are automatically determined via regex parsing of the argument used to run your pipeline.

**\*\*NOTE\*\*** Wildcards should be *UNIQUELY* named; suffix a rule's wildcards with an acronym for the rule.

E.G.    samples in cleanBAM    ==    samplesCB

E.G. chr in bamp2mpileup == chrB2M

Variables:      Input Expand - Indicated by single {}      E.G.      example("{rndVAR}", rndVAR=otherVar)  
                          Input String - Not Available

Inline variables can be used to data, and 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"]
```

$$Z = ["1", "2"]$$

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
```

7) **Go to the “input” directive, edit the input for the module.**

In “tPile” the input is a ‘.bam’ file, so the input argument should specify the location of this file for a given sample. The resultant input call should be the same call as in “mpileupGen/bam2pileup”, as seen in Figure 5.

```
input:
  inputBAM = expand("{outputDIR}/{bamDIR}/{sampleB2M}.bam", outputDIR=config["outputDIR"], bamDIR=config["bamDIR"],
  inputBAI = expand("{outputDIR}/{bamDIR}/{sampleB2M}.bam.bai", outputDIR=config["outputDIR"], bamDIR=config["bamDIR"],
  refFile = config["refFILE"]
```

Figure 5. VIM screenshot of the input directive for the submodule “bam2mpileup” of the module “mpileupGen”. The input directive explicitly lists all the files required by the rule.

8) **Go to the “output” directive, edit the output for the module.**

Aside from two exceptions, the output directive is designed the same as the input directive. The output directive cannot use functions; however, the output directive has optional file tags; further documentation [here](#). The resultant output call should be the same call as in “mpileupGen/bam2pileup”, as seen in Figure 6.

```
output:
  outputMpileup = temp(expand("{outputDIR}/{mpileupDIR}/{sampleB2M}/{chrB2M}.mpileup", outputDIR=config["outputDIR"], mpileupDIR=config["mpileupDIR"])))
```

Figure 6. VIM screenshot of the input directive for the submodule “bam2mpileup” of the module “mpileupGen”. The input directive explicitly lists all the files required by the rule.

File Tags:

- **temp()** – “filename” will be deleted when it is no longer required by the pipeline.
- **protected()** – “filename” is protected from being deleted by the pipeline, (E.g. it is reasonable to protect a ‘.bam’ file due to the resource costs of creating one).

9) **Go to the “wildcard\_constraints” directive, potentially include and restrict all wildcards.**

Wildcard constraints can be used to prevent cyclical dependencies; however, they also greatly reduce the generalizability of a rule. Use “wildcard\_constraints” sparingly, and refer to a global regular expression to ensure consistent regular expression restrictions throughout the system. The resultant “wildcard\_constraints” directive should be the same as in “mpileupGen/bam2pileup”, as seen in Figure 7.

```
wildcard_constraints:
  # Constrain the wildcard to not start with '-', '_' or '\', must be comprised of only letters and numbers
  sampleB2M = config["sampleREGEX"],
  # May or may not start with _, then consists of any number of letter, number or _, and must finish with a '.' AND a digit, or no '.' AND a digit
  chrB2M = config["chrREGEX"]
```

Figure 7. VIM screenshot of the input directive for the submodule “bam2mpileup” of the module “mpileupGen”. The wildcard\_constraints directive restricts the regular expressions used when Snakemake infers wildcards.

10) **Go to the “params” directive, 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. This is especially encouraged when there are five or more arguments to be passed to the program. The resultant “params” directive should be the same as in “mpileupGen/bam2pileup”, as seen in Figure 8.

```
params:
  mpileupARGS=config["countOrphan"] + " " + config["noBaq"] + " " + config["maxDepth"] + " " + config["mapQuality"] + " " + config["bedFILE"]
```

Figure 8. VIM screenshot of the input directive for the submodule “bam2mpileup” of the module “mpileupGen”. The params directive makes internal rule-specific parameters more flexible and accessible.

11) Go to “log” directive, edit the name of the log directory.

This should be the same as the module name. It is the same name used for the python and “\_INCLUDE” files. The resultant “log” directive should be the same as in “mpileupGen/bam2mpileup”, as seen in Figure 8.

```
log:
  stderr_mpileup = "log/" + config["mpileupGenDIR"] + '/bam2mpileup/bam2mpileup_{sampleB2M}. ' + strftime("%Y-%m-%d.%H-%M-%S", localtime()) + '.mpileup.stderr',
  stderr_view = "log/" + config["mpileupGenDIR"] + '/bam2mpileup/bam2mpileup_{sampleB2M}. ' + strftime("%Y-%m-%d.%H-%M-%S", localtime()) + '.view.stderr'
```

Figure 9. VIM screenshot of the log directive for the submodule “bam2mpileup” of the module “mpileupGen”. The log directive describes where to store log files for the rule. It is utilized by the scheduler if run on a cluster.

12) Go to the “run” directive, add the python execution.

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 resultant “run” directive should be the same as in “mpileupGen/bam2mpileup”, as seen in Figure 10.

**\*\*NOTE\*\*** Separate statements which produce error responses. This is important to ensure that the logging is properly captured on both cluster and non-cluster processing environments.

**\*\*NOTE\*\*** It is important to include off cluster error log piping in the event the cluster schedule is not used to generate log files. This is exemplified in sub-sections 1.B and 1.D of the run directive.

**\*\*NOTE\*\*** It is important to record the python execution call just prior to execution. This is done in sub-section 2.A of the run directive. By utilizing the string pathing created in out log directive, we can use a truncated portion as the descriptive portion of the execution call which are written to the file named by the “shellCallFile” variable.

**\*\*NOTE\*\*** Convert the software calls into YAML variables, as instructed in the pre-conditions.

```
run:
  # 1.A - View input via Samtools.
  callString=config["mpileup_samtoolsProg"] + ' view -bh -F ' + str(config["bitFlag"]) + ' ' + str(input.inputBAM) + ' ' + wildcards.chrB2M[1:]

  # 1.B - Differing output redirection as when not on cluster we cannot use the cluster config file.
  if config["offCluster"]:
    callString += ' 2> ' + str(log.stderr_view)

  # 1.C - Process into mpileup.
  callString += ' | ' + config["mpileup_samtoolsProg"] + ' mpileup ' + str(params.mpileupARGS) + ' -f ' + str(input.refFile) + ' -> ' + str(output.outputMpileup)

  # 1.D - Differing output redirection as when not on cluster we cannot use the cluster config file.
  if config["offCluster"]:
    callString += ' 2> ' + str(log.stderr_mpileup)

  # 2.A - Printing system calls to a local file, and then executing them.
  call('echo "#" + str(log.stderr_view)[-12] + ':\n#' + callString + '\n' >> ' + config["shellCallFile"], shell=True)
  call(callString, shell=True)
```

Figure 10. VIM screenshot of the run directive for the submodule “bam2mpileup” of the module “mpileupGen”. It contains rule’s functionality.

13) Go to Line 10, write the purpose for this rule.

14) Go to Lines 6 and 8, list the functional inputs and outputs for this rule.

15) Go to Lines 27-42, delete the formatted template for a function declaration.

16) When finished, “tPile” should look similar to Figure 11.

**\*\*NOTE\*\*** The usage of external YAML variables is very important when designing a rule. The configuration of these external variables is covered in the section “Module Design – Configuring ‘tutorialGen.py’”.

```
#-----
# Author:   Tim Boyarski
# Date:    2017-06-31
#-----
# Call: call("python " + snakeDIR + "/modules/tutorialGen/tutorialGen.py " + YAMFILE + " " + CLUSTERFILE + " " + SNAKEFILE, shell=True)
# Input:    .bam
# Output:   .mpileup
# Purpose:  Generate an '.mpileup' file from a '.bam' file.
#
# **NOTE** Input files are always kept. No way via this
# script to automate their destruction. This was done to
# avoid the risk of deleting the processed '.bam' files.
#
# **NOTE** Wildcard constraints may prevent some sample names
# from being used.
#-----
# PYTHON PACKAGES #
#-----
# Used for system calls.
from subprocess import call

# Used for timestamping the log files.
from time import localtime, strftime
#-----

#-----
# SNAKEMAKE RULE #
#-----
rule tPile:
    input:
        inputBAM = expand("{outputDIR}/{bamDIR}/{sampleTP}.bam", outputDIR=config["outputDIR"], bamDIR=config["bamDIR"]),
        inputBAI = expand("{outputDIR}/{bamDIR}/{sampleTP}.bam.bai", outputDIR=config["outputDIR"], bamDIR=config["bamDIR"]),
        refFile = config["refFILE"]
    output:
        tPileOutput = temp(expand("{outputDIR}/{tutorialGenDIR}/{sampleTP}/{chrTP}.mpileup", outputDIR=config["outputDIR"], tutorialGenDIR=config["tutorialGenDIR"]))
    wildcard_constraints:
        # Constrain the wildcard to not start with '-' '_' or '\', must be comprised of only letters and numbers
        sampleTP = config["sampleREGEX"],
        # May or may not start with _, then consists of any number of letter, number or _, and must finish with a '.' AND a digit, or no '.' AND a digit
        chrTP = config["chrREGEX"]
    params:
        mpileupARGS=config["countOrphan"] + " " + config["noBaq"] + " " + config["maxDepth"] + " " + config["mapQuality"] + " " + config["bedFILE"]
    log:
        stderr_tPile = "log/" + config["tutorialGenDIR"] + '/tPile/tPile_{sampleTP}.log' + strftime("%Y-%m-%d.%H-%M-%S", localtime()) + '.tPile.stderr',
        stderr_view = "log/" + config["tutorialGenDIR"] + '/tPile/tPile_{sampleTP}.log' + strftime("%Y-%m-%d.%H-%M-%S", localtime()) + '.view.stderr'
    run:
        # 1.A - View input via Samtools.
        callString=config["tutorialGen_samtoolsProg"] + ' view -bh -F ' + str(config["bitFlag"]) + ' ' + str(input.inputBAM) + ' ' + wildcards.chrTP[1:]

        # 1.B - Differing output redirection as when not on cluster we cannot use the cluster config file.
        if config["offCluster"]:
            callString += ' 2> ' + str(log.stderr_view)

        # 1.C - Process into mpileup.
        callString += ' | ' + config["tutorialGen_samtoolsProg"] + ' mpileup ' + str(params.mpileupARGS) + ' -f ' + str(input.refFile) + ' -> ' + str(output.tPileOutput)

        # 1.D - Differing output redirection as when not on cluster we cannot use the cluster config file.
        if config["offCluster"]:
            callString += ' 2> ' + str(log.stderr_tPile)

        # 2.A - Printing system calls to a local file, and then executing them.
        call('echo "#' + str(log.stderr_view)[-12] + ':\n#' + callString + '\n>> ' + config["shellCallFile"], shell=True)
        call(callString, shell=True)
#-----
```

Figure 11. VIM screenshot of the submodule “tPile” of the module “tutorialGen”.

## VIII. Module Design – Configuring “tutorialGen.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 “tutorialGen.py”.**

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 “XXXXXXXX” with “tutorialGen”.

4) **Go to Line 8, edit the module’s purpose.**

Typically, the purpose can be left unmodified, the default description is sufficient.

5) **\*\*Optional\*\* Highlight the “XXXXXXXX” fields to be edited.**

The fields can be highlighted by searching for them using VIM’s “/”, or you can move the cursor over one of the fields and highlight it by pressing “Shift + #”. To remove this highlighting, type “:noh” into VIM, or, eliminate it by changing all of the highlighted fields.

6) **Go to Line 22, 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.

7) **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.

8) **Edit subsection 4: Snakefile (~Lines 92-109).**

This is covered in steps 9.

9) **Go to Line 97, edit what will be written to the Snakefile when this module’s python script is called.**

The statements required to execute any of the submodules should be listed here. Use the output statements from each rule as a template. Convert the output statement {{samples}} from a wildcard format, to a variable format by removing a set of {}. Then, add a declaration at the end, ‘samples=config[“sample”]’. The resultant code should be the same as in “mpileupGen/mpileupGen.py”, as seen in Figure 12.



```
with open(argv[3], "a") as pipeTARGET:
    pipeTARGET.write(
        """
        ##### " + moduleName + " #####
        "# Included:
        "# mpileupUNSPPLIT: Generated 'mpileup' file from 'bam' file.
        "# mpileupSPLIT: Generated 'mpileup' file from 'bam' file.
        'include: " + path.dirname(path.realpath(__file__)) + '/' + moduleName + '_INCLUDE'
        "# Required: NONE
        "# Call via:
        'bam2mpileup: expand("{outputDIR}/{mpileupDIR}/{samples}.mpileup", outputDIR=config["outputDIR"], mpileupDIR=config["mpileupDIR"], samples=config["sample"]),
        'bam2mpileupCHR: expand("{outputDIR}/{mpileupDIR}/{samples}_{chrLIST}.mpileup", outputDIR=config["outputDIR"], mpileupDIR=config["mpileupDIR"], samples=config["sample"],
        chrLIST=config["chrLIST"]),
        """
    )
```

Figure 12. VIM screenshot of the python script for module “mpileupGen”. Provides output calls for requests at the chromosome and the genome level.

10) **Edit subsection 3: JSON File (~Lines 79-92).**

This is covered in steps 11-14.

11) **Go to line Line 85, and add the rule name for JSON configuration.**

Add the name of the rule “tPile” within each submodule.

12) **Go to line Line 86, edit the logging directory for the rule.**

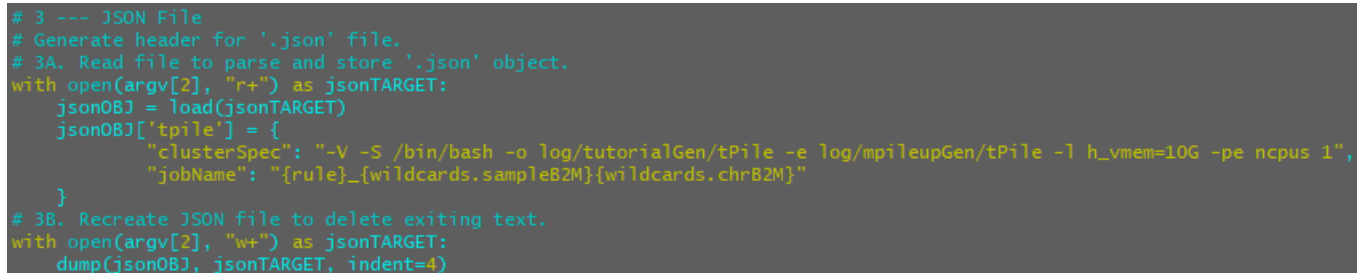
The directory structure follows “log/moduleName/ruleName”.

13) **Go to line Line 87, edit the job name to reasonable include all wildcards.**

Try to following the existing job naming convention of “rule\_sampleName\_firstWildCard\_secodWildCard”.

**\*\*NOTE\*\*** When listing chromosomal wildcards, keep in mind they already contain an underscore.

14) **The result of steps 9, 10 and 11 should look similar to Figure 13.**

A VIM screenshot showing a Python script for module 'tutorialGen'. The script is editing a JSON file. It defines a 'clusterSpec' and a 'jobName' for a rule named 'tPile'. The 'jobName' is constructed using wildcards for rule, sample, and chromosome. The script also includes comments about generating headers and reading/writing JSON files.

```
# 3 --- JSON File
# Generate header for '.json' file.
# 3A. Read file to parse and store '.json' object.
with open(argv[2], "r+") as jsonTARGET:
    jsonObj = load(jsonTARGET)
    jsonObj['tPile'] = {
        "clusterSpec": "-V -S /bin/bash -o log/tutorialGen/tPile -e log/mpileupGen/tPile -l h_vmem=10G -pe ncpus 1",
        "jobName": "{rule}_{wildcards.sampleB2M}{wildcards.chr82M}"
    }
# 3B. Recreate JSON file to delete exiting text.
with open(argv[2], "w+") as jsonTARGET:
    dump(jsonObj, jsonTARGET, indent=4)
```

Figure 13. VIM screenshot of the JSON section of the python script for module "tutorialGen". Provides default specifications on job names, logging, and hardware requests for this module.

15) **Edit the subsection 2: YAML File (~Lines 55-79).**

This is covered in steps 16-17.

16) **Go to Lines 60-64, list the parameters by category.**

The parameters listed here should fall under three category types: 2A. Software, 2B. Shared variables, and 2C. submodule variables. All submodule headings used 2C. The subsection 2D is wh

17) **Go to the line below “yamlTARGET.write(“ , edit the string to have all variables written to YAML file.**

This subsection of the python script will populate the YAML file, so all entries added must follow YAML syntax. Each category is to be provided a line heading. Category variables can be listed on the same line; max 150 characters per line. Be sure to add all the parameters that were used in the coding of the module “tPile”.

**\*\*NOTE\*\*** Some of the config variables in tPile\_INCLUDE are global variables, and do not need to be included.

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. Please follow the provided format.

```
var1="var1: X\n"
var1="var2: X\n"
yamlTARGET.write("` ... .. #\n" + var1 + var2)
```



**18) Edit the subsection 1: Log Files (~Lines 38-55).**

This is covered in step 19.

**19) Go to Line 49 and 50, edit the “ruleList” variable to contain active module “tPile”.**

This must stay as a list, even if it is only rule.

**20) When finished, the YAML file section should reflect Figure 14 and Figure 15.**

Figure 14: Header, Python Packages, Python Script: Validation and Log Files.

Figure 15: YAML File, JSON File, Snakefile

```
#-----
# Author:   Tim Boyarski
# Date:    2017-08-31
#-----
# Call: call("python " + snakeDIR + "/modules/tutorialGen/tutorialGen.py " + YAMLFILE + " " + CLUSTERFILE + " " + SNAKEFILE, shell=True)
# Input:                                     .bam

# Output:                                     .mpileup

# Purpose: Automate the population of user's pipeline
#   Snakefile, '.YAML', and '.JSON' files.
#-----
# PYTHON PACKAGES #
#-----
# Request sys so be able to use CLI arguments.
from sys import argv

# Request json to be able to load and write to the config.json file.
from json import load, dump

# Request os permissions to be able to create directories for the log files.
from os import path, mkdir

# Global variable used for reporting of the module name.
moduleName = "tutorialGen"
#-----

#-----
# PYTHON SCRIPT #
#-----
# 0 --- Validate number of user arguments.
if len(argv) != 4:
    print("Please provide arguments as follows:")
    print("python " + moduleName + ".py yaml json snake")
    print("\t-yaml = 'path/name' of the yaml file to write the pipeline parameters")
    print("\t-json = 'path/name' of the json file we write the cluster config to")
    print("\t-snake = 'path/name' of snakefile we are building")
    quit()
#-----

# 1 --- Log Files
# Check if directories exist for logging, as the DRMAA caller cannot create directories.
if (path.isdir("log")) != True:
    mkdir("log")
    print(moduleName + ".py \tCreating: log/")
if (path.isdir("log/" + moduleName) != True):
    # Maintain this list of active submodules.
    ruleLIST = ['tPile']
    # 1A. Create module directories
    mkdir("log/" + moduleName)
    print(moduleName + ".py \tCreating: log/" + moduleName)
    # 1B. Report on directories created.
    for rule in ruleLIST:
        mkdir("log/" + moduleName + "/" + rule)
        print(moduleName + ".py \tCreating: log/" + moduleName + "/" + rule + "/")
#-----
```

Figure 14. VIM screenshot of the Header, Python Packages, Python Script: Validation and Log Files section of the python script for module "tutorialGen". Provides general purpose information for the file, user error feedback, and default specifications on logging.

```

# 2 --- YAML File
# Open and append to file the following required parameters.
with open(argv[1], "a+") as yamlTARGET:
    # 2A. Software
    tutorialGen_samtoolsProg = "tutorialGen_samtoolsProg: samtools"
    # 2B. Shared variables
    tutorialGenDIR="tutorialGenDIR: tutorialGen"
    bitFlag="bitFlag: 512"
    countOrphan="countOrphan: -A"
    noBaq="noBaq: -B"
    maxDepth="maxDepth: -d 10000000"
    mapQuality="mapQuality: -q 20"
    bedFILE="bedFILE: ""
    # 2C. bam2mpileup
    # 2D. Write to file
    yamlTARGET.write(
        "#####\n"
        "#####\n"
        "# " + moduleName + " Parameters\n"
        "#####\n"
        "----- *Software* -----\n"
        tutorialGen_samtoolsProg +
        "----- *Shared Variables* -----\n"
        tutorialGen + bitFlag + countOrphan + noBaq + maxDepth + mapQuality + bedFILE +
        "----- bam2mpileup -----\n"
        "#####\n"
    )

# 3 --- JSON File
# Generate header for '.json' file.
# 3A. Read file to parse and store '.json' object.
with open(argv[2], "r+") as jsonTARGET:
    jsonObj = load(jsonTARGET)
    jsonObj['tPile'] = {
        "clusterSpec": "-V -S /bin/bash -o log/tutorialGen/tPile -e log/tutorialGen/tPile -l h_vmem=10G -pe ncpus 1",
        "jobName": "{rule}_{wildcards.sampleTP}{wildcards.chrTP}"
    }

# 3B. Recreate JSON file to delete exiting text.
with open(argv[2], "w") as jsonTARGET:
    dump(jsonObj, jsonTARGET, indent=4)

# 4 --- Snakefile
# Open and append o file a descriptin and the last rule call.
with open(argv[3], "a+") as pipeTARGET:
    pipeTARGET.write(
        "#####\n"
        "# " + moduleName + " *****\n"
        "# Included: \n"
        "# tPile: Fancy generated description.\n"
        'include: "' + path.dirname(path.realpath(__file__)) + '/' + moduleName + '_INCLUDE"\n'
        "# Required: NONE\n"
        "# Call via: \n"
        '# expand("{outputDIR}/{tutorialGenDIR}/{samples}.mpileup", outputDIR=config["outputDIR"], tutorialGenDIR=config["tutorialGenDIR"], samples=config["sample"])\n'
        "#####\n"
    )

```

Figure 15. VIM screenshot of the YAML File, JSON File, Snakefile section of the python script for module "tutorialGen". Provides rule specific content for the YAML, JSON, and Snakefile of any pipeline using this rule.

IX. Module Design – Configuring the “README.md”

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.
- 2) **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.
- 3) **Regression Testing**  
List all regression testing that was performed and when.
- 4) **Logging**  
Just change the directory in which it is being stored. The directory will be named the same as this module’s.
- 5) **Control flow**  
Describe the control implemented within the module. Often a discussion of the “\_INCLUDE” file.
- 6) **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.
- 7) **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.
- 8) **Module Specific Software**  
The module specific software variables should all be prefixed by the module name.
- 9) **Module Specific Parameters**  
The module specific parameters and their descriptions are to be listed here. Follow the formatting provided. The resultant table should look similar to Figure 16 in file, and Figure 17 when viewed from a GitHub.

```
30 Module | Argument | Default Value | Description
31 :-----: | :-----: | :-----: | :-----:
32 All | bitFLAG | 512 | Bitwise flag
33 All | countORPHAN | -A | Count orphan reads
34 All | noBAQ | -B | BAQ computation
35 All | maxDEP | -q 10000000 | Maximum read depth
36 All | mapQUAL | -d 20 | Minimum quality of reads to be used.
```

Figure 16. 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.

Module	Argument	Default Value	Description
All	bitFLAG	512	Bitwise flag
All	countORPHAN	-A	Count orphan reads
All	noBAQ	-B	BAQ computation
All	maxDEP	-q 10000000	Maximum read depth
All	mapQUAL	-d 20	Minimum quality of reads to be used.

Figure 17. GitHub screenshot of the mPile module README.md. The image shows how the table format code is represented visually.

- 10) **When finished, “README.md” should look the actively maintained production versions.**
  - [bamUtil ReadMe](#)

- [fastqUtil ReadMe](#)
- [mpileupGen ReadMe](#)
- [vcfGenUtil varScan ReadMe](#)

## X. 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/>]

## XI. 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
```