Script of Scripts

Release

April 23, 2016

Specification of SoS format version 1.0.

Terminology & Grammar

- **Script**: A SoS script that defines one or more workflows.
- **Workflow**: A sequence of processes that can be executed to complete certain task.
- **Step**: A step of a workflow that perform one piece of the workflow.
- **Step options**: Options of the step that assist the definition of the workflow.
- **Step input**: Specifies the input files of the step.
- **Step output**: Specifies the output files of the step.
- **Step dependencies**: Specifies the files that are required by the step.
- **Step process**: The process that a step executes to complete specified work, specified as one or more Python statements.
- **Action**: SoS or user-defined Python functions. They differ from regular Python functions in that they may behave differently in different running mode of SoS (e.g. ignore when executed in dryrun mode).

The SoS syntax obeys the following grammar, given in extended Backus-Naur form (EBNF):

```
` Script = {comment}, {statement}, [parameter_step], {step}; comment =
"#", text, NEWLINE parameter_step = "[parameters]", NEWLINE, {[comment],
assignment} assignment = name, "=", expression, NEWLINE `
```

That is to say, a SoS script contains **comments**, **statements**, an optional **parameter step** with multiple assignment statements, and one or more SoS **steps**. Here *name*, *expression* and *statement* are arbitrary [Python](http://www.python.org) names, expression and statements with added SoS features. **SoS requires Python 3 and does not support Python 2.x specific syntax**

‘‘‘ step = step_header,

{comment}, {{statement}, [input | output | depends]}, [process, NEWLINE, {script}]

step_header = “[”, names, [”:”, names | options], “]”, NEWLINE input = “input”, ”:”, [expressions], [”,”, options], NEWLINE output = “output”, ”:”, [expressions], [”,”, options], NEWLINE depends = “depends”, ”:”, [expressions], [”,”, options], NEWLINE process = “process” | name, ”:”, [options] names = name, {”,”, name} workflow = name, [’_’, steps], {“+”, name, [’_’, steps} assignment = name, “=”, expression, NEWLINW expressions = expression, {”,”, expression} options = option, {”,” option} option = name, “=”, expression ‘‘‘

A SoS step consists of a header with one or more names, optional options. The body of a SoS step consists of optional comments, statements, input, output, or depends parameters, followed by step process.

This documentation is a comprehensive reference to all SoS features. Please refer to this [SoS quick start tutorial](https://github.com/bpeng2000/SOS/wiki/Quick-Start) to learn some basics of SoS before diving into the details.

Language (Python)

SoS uses [Python](http://www.python.org) expressions and statements. If you are unfamiliar with Python, you can learn some basics of Python, usually in less than half a day, by reading some Python tutorials (e.g. [the official python tutorial](https://docs.python.org/3/tutorial/)). This [short introduction](https://docs.python.org/3/tutorial/introduction.html) is good enough for you to use SoS.

SoS makes two modifications to standard Python syntax.

- String literals quoted by single triple quotes (‘” ‘’‘) are considered to be raw strings.
- String literals are format string that will be [interpolated](#string-interpolation) using SoS defined sigil.

These changes are made to make the inclusion of scripts easier in a SoS script and are only effective for Python statements in SoS scripts. Modules defined and imported in other python files are interpreted by a standard Python interpreter.

Raw multi-line strings

Whereas the following string literal in regular python

```
'python '''A multiline string with \n and newline''' '
```

would be treated as

```
'python 'A multiline string with \n\nand newline' '
```

SoS will treat it as

```
'python 'A multiline string with \\n\nand newline' '
```

so that it can be sent to the underlying interpreters as it is. Basically, SoS does not require the raw string prefix *r* for multiple line strings quoted by triple single quotation marks.

String interpolation

On top of python string manipulation functions and similar to recently introduced (Python 3.6) format string, SoS uses string interpolation to replace variables and expressions within a string with their values. For example, expressions *resource_path*, *sample_names[0]* and *sample_names* would be replaced by their values in the following string definitions.

‘‘‘python resource_path = ‘~/.sos/resources’ ref_genome = ‘${resource_path}/hg19/refGenome.fasta’ # ‘~/.sos/resources/hg19/refGenome.fasta’

sample_names = [’A’, ‘B’, ‘C’] title = ‘Sample ${sample_names[0]} results’ # ‘Sample A results’ all_names = ‘Samples ${sample_names}’ # ‘Samples A B C’ ‘‘‘

Depending on the return type of the expression

- String representations (*repr(obj)*) are returned for objects in simple Python types (e.g. string, True, False, None, numeric numbers)
- For objects with an iterator interface (e.g. Python *list*, *tuple*, *dict*, and *set*), SoS join the string representation of each item by a space or comma as specified by conversion flag *!,* (see [conversion and format](#conversion-and-format) for details). More specifically,
  - List of strings will be converted to a string by joining strings with a space or comma.
  - Dictionary of strings will be converted to a string by joining dictionary keys with no guarantee on the order of values.

If you are unhappy with the string conversion, you can always format your object using python expressions such as *${repr(obj)}* and *${‘, ‘.join(x for x in obj)}*.

SoS supports nested interpolation, for example

```
'python '${_input[${index}]}' '
```

would evaluate *${index}* and then *${_input[?]}* where *?* is the result of *${index}*.

Note that you can continue to use Python string functions such as

```
'python ref_genome = resource_path + '/hg19/refGenome.fasta' title = 'Result
for the first sample {} is invalid'.format(sample_names[0]) '
```

but string interpolation is recommended for multi-line scripts because it is easier to read.

Conversion and format

SoS interpolation also support all string format and conversion specification as in the [Python string format specifier](https://docs.python.org/2/library/string.html#formatspec), that is to say, you can use *: specifier* at the end of the expression to control the format of the output. For example

- *${1/3. :.2f}* in a string literal yields *0.33* instead of its long representation *0.3333333333333333*, and
- *${filename!r}* in a string literal yields *'Article by "Jane Eyre"'* (note the quotation mark) if *filename='Article by "Jane Eyre"'*.

The latter is interesting because SoS uses correct quotation marks for filenames with quotation marks, making

```
'python R(''' read.csv(${_input!r}) ''') '
```

a safer choice than

```
'python R(''' read.csv("${_input}") ''') '
```

because *${_input}* might contain quotation marks.

SoS supports *!q* (quoted) conversion that converts filenames to proper filenames that can be safely used in shell command. For example, the following command

```
'python run('cat ${_input!q}') '
```

would produce the correct command and execute

```
' cat 'Bon Jovi.txt' '
```

instead of

```
' cat Bon Jovi.txt '
```

for *_input=['Bon Jovi.txt']*.

SoS also supports a , conversion flag that can be appended to *!r*, *!q*, *!s* (for string representation of a Python object) or just *!*. This tells SoS to join items of sequences by comma instead of a space. For example, sequences results in the following example are joint by *','*:

```
'''python items = ['A', 'B', 'C'] inames = '${items}' # A B C inames = '${items!,}' # A,B,C inames = '${items!r,}' # 'A','B','C'

files = ['AB.txt', 'C D.txt'] fname = '${files!q,}' # AB.txt,CD.txt or AB.txt,'C D.txt' '''
```

If the default sigil conflicts with the script used, an alternative sigil can be used. Please see section [option *sigil*](#option-sigil) for details.

File header

A SoS script usually starts with lines

```
'python #!/usr/bin/env sos-runner #fileformat=SOS1.0 '
```

The first line allows the script to be executed by command *sos-runner* if it is executed as an executable script. The second line tells SoS the format of the script. The *#fileformat* line does not have to be the first or second line but should be in the first comment block. SOS format 1.0 is assumed if no format line is present.

Comments

Comments in a SoS script are mostly meaningful, meaning that they will be displayed in output of commands such as *sos show script*. More specifically

- **Script description**: Description of SoS script consists of comment blocks after the shabang block and before any workflow description or SoS statement.
- **Workflow descriptions**: Description of individual workflows consists of comment blocks after a comment line with only workflow name and before any SoS statement. Workflow descriptions can be put at the front of the script or between sections.

- **Step description**: Comments after section head and before the first SoS statement are step descriptions/
- **Parameter description**: Comments before each parameter in the [parameters section](#command-line-options) are parameter descriptions.

For example,

‘‘‘python #!/usr/bin/env sos-runner #fileformat=SOS1.0

This is a demo script for SoS comments.

human # align raw reads to human genome [parameters] # path to STAR star_path = ‘~/bin/STAR’

[*_1] # prepare index run:

    command1

[human_2] # align reads to human genome run:

    command2

mouse # align raw reads to mouse genome [mouse_2] # align reads to mouse genome run:

    command3

‘‘‘

defines two workflows (*human* and *mouse*). Assuming the sos script is named *test.sos*, the output of command *sos show test.sos* would be

‘‘‘ This is a demo script for SoS comments.

Available workflows: human, mouse Workflow human: align raw reads to human genome

    Parameters: star_path path to STAR (default: ‘~/bin/STAR’)

    Step human_1: prepare index Step human_2: align reads to human genome

Workflow mouse: align raw reads to mouse genome

    Parameters: star_path path to STAR (default: ‘~/bin/STAR’)

    Step mouse_1: prepare index Step mouse_2: align reads to mouse genome

‘‘‘

Global definitions

Python functions, classes, variables can be defined or imported (using Python *import* statement) before any SoS step is defined. These definitions usually contains variables such as version and date of the script, paths to various resources, and utility functions that will be used by later steps. **They will be executed before each step process**. In addition, **variables defined globally are readonly** and change of these variables in later steps will trigger an error.

In addition to user-defined global variables, SoS defines the following variables before any variables are defined

- **‘SOS_VERSION‘**: version of SoS command.
- **‘CONFIG‘**: A dictionary of configurations specified by command line option *-c config_file*. The configuration file should be in the format of [*YAML*](http://yaml.org/) or its subset format ['JSON'](http://json-schema.org/implementations.html). Inside SoS script you can access these variables via, for example, either *CONFIG['gatk_path']* or *CONFIG.gatk_path*.

Command line options

Optional arguments

SoS looks for a *[parameters]* section for command line options and their default arguments. The format of each variable is

```
` # comment var_name = default_value `
```

The default value can be number, string, list of string, or expressions that return values of these types. Other types can be used as long as they can be converted to these types from user-provided values. For example

```
` # path to tool gatk gatk_path = '~/bin/GATK' `
```

defines a variable *gatk_path* with default value *'~/bin/GATK'*.

```
` # A list of sample names sample_names=[] `
```

defines a variable *sample_name* with default value *[]*. And

```
`python # path to gatk gatk_path=CONFIG['gatk_path'] `
```

uses *gatk_path* from a YAML/JSON-based configuration file (specified from command line using option *-c*) as default value. You can set default value for *CONFIG* inside SoS script via:

```
`python # path to gatk gatk_path=CONFIG.get('gatk_path', '/default/path/to/gatk')
`
```

so that if configuration files are not supplied the default value */default/path/to/gatk* will be used.

The default values not only determines the values of variable when they are not specified from command line or configuration files, but also determines the type of input these parameters accept. For example, with the above definitions for command arguments *–gatk_path* and *–sample_names*, you can pass values to these variables from command line,

```
`bash sos run myscript.sos --gatk_path /path/to/gatk --sample_names A1 A2 A3 `
```

A list will be passed to *sample_names* even if only a single value is provided (e.g. *sample_names=['A1']* for *–sample_name A1*). Attempts to pass more than one values (a list) to *gatk_path* (e.g. *–gatk_path /path1 /path2*) will trigger an error.

Note that bool values can be specified from command line as *yes*, *true*, *t*, *1* (case insensitive) for *True* and *no*, *false*, *f*, *0* for *False*.

Required arguments

In cases where there is no suitable default values and/or command line arguments are mandatary, you can list the type of arguments (e.g. *int*, *bool*, *str*, *list* of strings) in place of default values. For example, if an integer parameter *cutoff* is required, you can define it as

```
`python # cutoff value cutoff = int `
```

This will force the users to provide an integer to this parameter. You can do the same for lists but SoS assumes that you need a **list of strings**. For example, the following definition

```
`python # input bam files bam_files = list `
```

request a list of strings from command line. SoS will return a list even if only one value is provided.

Workflow definitions

A SoS script can specify one or more workflows. Each workflow consists of one or more numbered steps. The numbers (should be non-negative) specify the **logical order** by which the steps are executed, but a later step might be executed before the completion of previous steps if it does not depend on the output of these steps.

Single workflow

A single workflow can be specified without a name in a SoS script. For example, the following sections specify a workflow with four steps *5*, *10*, *20*, and *100*. As you can see, the workflow steps can be specified in any order and do not have to be consecutive (which is actually preferred because it allows easy insertion of extra steps).

```
` [5] [20] [10] [100] `
```

Workflows specified in this way is the *default* workflow and are actually called *default* in SoS output. If you want to give it a meaningful name, you can specify the steps as

```
` [mapping_5] [mapping_20] [mapping_10] [mapping_100] `
```

Because this SoS script defines only one workflow (*mapping*), you do not have to specify the name of workflow from SoS command

```
`bash sos run myscript.sos --input input1.fasta `
```

Unnumbered workflow steps are assumed to be the first (with index *0*) of a workflow unless they have other meanings (e.g. *parameters* or [auxiliary step](auxiliary-workflow-steps-and-makefile-style-dependency-rules)).

Multiple workflows

A SoS script can define multiple workflows. For example, the following sections of SoS script defines two workflows named `mouse` and `human`.

```
` [mouse_10] [mouse_20] [mouse_30] [human_10] [human_20] [human_30] `
```

You will have to specify which workflow to execute from the command line, e.g.

```
`bash sos run myscript mouse --input input1.fasta `
```

If you would like to define a `default` and a named workflow, you can define them as

```
` [10] [20] [30] [test_10] [test_20] [test_30] `
```

The *default* workflow will be executed by default using command

```
`bash sos run myscript.sos `
```

The *test* workflow will be executed if its name is specified from the command line

```
`bash sos run myscript.sos test `
```

Shared workflow steps

The most common motivation of defining multiple workflows in a single SoS script is that they share certain processing steps. If this is the case, you can define sections such as

```
` [mouse_10,human_10] [mouse_20] [human_20] [mouse_30,human_30] `
```

or

```
` [*_10] [mouse_20] [human_20] [*_30] `
```

or

```
` [*_10] [mouse_20,human_20] [fly_20] [*_30,fly_50] [fly_40] `
```

In the last case, step defined by *[*_30,fly_40]* will be expanded to `mouse_30`, `human_30`, `fly_30`, and `fly_50` and will be executed twice for the *fly* workflow. Note that workflow steps can use variable *step_name* to act (slightly) differently for different workflows. For example,

‘‘‘python [mouse_20,human_20] reference = mouse_reference if step_name.startswith(‘mouse’) else human_reference

input: fasta_files depends: reference

‘‘‘

Here the variable *step_name* is *mouse_20* or *human_20* depending on the workflow being executed, and expression *mouse_reference if step_name.startswith(‘mouse’) else human_reference* returns *mouse_reference* if the workflow *mouse* is executed, and *human_reference* otherwise.

Subworkflow

Although workflows are defined separately with all their steps, they do not have to be executed in their entirety. A *subworkflow* refers to a workflow that is defined from one or more steps of an existing workflows. It is specified using syntax *workflow[_steps]* where step can be *n* (step *n*), *-n* (up to *n*), *n-m* (step *n* to *m*) and *m-* (from *m*). For example

```
` A # complete workflow A A_5-10 # step 5 to 10 of A A_50- # step 50 up
A_-10 # up to step 10 of A A_10 # step 10 of workflow A can be considered a
subworkflow `
```

Combined workflow

You can also combine subworkflows to execute multiple workflows one after another. For example,

```
`python A + B # workflow A, followed by B A_0 + B # step 0 of A, followed by B
A_-50 + B + C # up to step 50 of workflow A, followed by B, and C `
```

This syntax can be used from the command line (option *workflow*, e.g. *sos-runner myscript.sos align+call*) or used to execute [nested workflows](#nested-workflow) inside the SoS script. Note that parameters steps of each subworkflow will be executed before it. For example,

```
`python A_0 + B `
```

will execute parameters step of *A* (with command line arguments), step *0* of *A*, parameters step of *B* (with command line argument), and all steps of *B*. Command line arguments might be used multiple times and SoS is not be able to identify misspecified (and thus unused) arguments in this case.

It is worth noting that combined workflow might work differently from when they are executed individually (e.g. default input of *B* is changed from empty to output of *A_0*), and it is up to the user to resolve conflicts between them (e.g. the same parameters with different types might be used in the workflows). Running a combined workflow in *dryrun* mode is always a good idea to test complex workflows.

Nested workflow

SoS also supports nested workflow in which a complete workflow is treated as part of a step process. The workflow is execute by SoS action *sos_run*, e.g.

```
` sos_run('A') # execute workflow A sos_run('A + B') # execute workflow B
after A sos_run('D:-10 + C') # execute up to step 10 of D and workflow C
sos_run('${aligner} + ${caller}') # execute user-specified aligner and caller
workflows `
```

In its simplest form, nested workflow allows you to define another workflow from existing ones. For example,

```
`python [default] sos_run('align+call') `
```

defines a nested workflow that combines workflows *align* and *call* so that the workflow will by default execute two workflows, but can also execute one of them as separate workflows *align* and *call*.

Nested workflow also allows you to define multiple mini-workflows and connect them freely. For example

```
`python [a_1] [a_2] [b] [c] [d_1] sos_run('a+b') [d_2] sos_run('a+c') `
```

defines workflows *d* that will execute steps *d_1*, *a_1*, *a_2*, *b_0*, *d_2*, *a_1*, *a_2*, and *c_0*. The parameters step for each workflow will be executed before any step in the workflow is executed.

Nested workflows, like other SoS actions, can be executed repeatedly, for example,

‘‘‘ [b_1] [b_2] [b_3]

[a] parameters = range(20) input: ‘some.txt’, for_each=’parameters’ output: ‘${input}_${_parameters}.res’ sos_run(‘b’) ‘‘‘

would execute the complete workflow *b* 20 times each with a different parameter. Similarly you can let the nested workflow process groups of input files.

Nested workflows can also be used to compose workflows from user-provided options through command line arguments, configuration files, and even results from previous steps. For example, the following example

‘‘‘ [parameters] # aligner steps to use to align the reads aligner = CONFIG.get(‘aligner’, ‘bwa’)

[bwa_1] [bwa_2] [novaalign_1] [novaalign_2]

[align] sos_run(aligner) ‘‘‘

defines workflows *bwa* and *novaalign* to align raw reads. The *align* workflow is a master workflow that executes *bwa* or *novaalign* determined by option *aligner* defined in a configuration file (command line option *-c*) and command line option *–aligner*.

Finally, if all or part of a nested work is defined in another script, you can specify it using the *source* step option. For example, the following step defines a *default* workflow that combined two workflows defined in *call.sos* and *align.sos*.

```
` [default] sos_run('call+align', source=["call.sos", "align.sos"]) `
```

The parameters steps of the current sos script and from *call.sos* and *align.sos* will be executed before *default*, *call* and *align* workflows, respectively.

The *source* option of the example essentially inserts the specified scripts into the existing script and might introduce conflicting workflows or workflow steps. A safer way to execute workflows from other scripts is to import them with an alias (namespace), using the dictionary form of option *source*. For example, if both *call.sos* and *align.sos* defines a *default* workflow, you will have to call them as follows:

```
` [default] sos_run('call.default + align.default', source={'call':
'call.sos', 'align':  'align.sos'}) `
```

Workflow steps

Although no item is required for a SoS step, a complete SoS step can have the following form

‘‘‘ [name_step: option1, option2, ...] # # description of the step #

statements input: input files, opt1=value1, opt2=value2

statements (executed in SoS) output: output files, opt1=value1, opt2=value2

statements (executed in SoS) depends: dependent files, opt1=value1, opt2=value2

statements (executed in SoS) process/action: opt1=value1, opt2=value2

statements or script (can executed in separate process)

action: script

‘‘‘

Basically, a step can have *input*, *output*, *depends*, *process* keywords with their options, with arbitrary Python statements before and in between, followed by a step process with one or more step actions.

Step options

Step options are specified after step name that assists the specification of workflows. SoS provides the following options

Option *skip*

Option *skip* takes two formats, the first format has no value

```
` [10:  skip] `
```

and is equivalent to

```
` [10:  skip=True] `
```

The whole step will be skipped as if it is not defined at all in the script. This option provides a quick method to disable a step.

The second format takes a value, which is usually an expression that will be evaluated when the step is executed. For example, in the following workflow,

‘‘‘ [parameters] # tool suite to align the reads quality_check = True

[10: skip= not quality_check] ‘‘‘

Step 10 will be skipped if option *–quality_check no* is specified from command line.

Option *sigil*

Because a SoS script can include scripts in different languages with different sigils (special characters used to enclose expression), **SoS allows the use of different sigils** in the script. Whereas the default sigil (*${ }*) has to be used for global variables, other sigils can be used for different sections of the script. For example

‘‘‘python [step_10: sigil=’%( )’] title = ‘Sample %(sample_names[0]) results’ run:

```
for file in *.fastq do
    echo Processing ${file} ... echo %(title)
done
```

‘‘‘

uses a different sigil style because the embedded shell script uses *${ }*. In this example *${file}* keeps its meaning in the shell script while *%(sample_names[0])* and *%(title)* are replaced by SoS to their values.

Option *sigil* accept only a constant string (not a variable or expression).

Option *alias*

SoS executes each step in a separate process and by default does not return any result to the master SoS process. If an option *alias* is specified with a name, a SoS step will return an object with specified name and attributes *name*, *input*, *depends*, *output*, and all variables defined in the step. The value of this option should be a quoted string (e.g. *alias=’align’*). For example, the following step assigns *index* (10), *input* (*fasta_files*), *output* (*[’accepted_reads.bam’, ‘summary.stat’]*) to variable *align* so that a later step can refer to it.

‘‘‘python [10: alias=’align’] input: fasta_files output: ‘accepted_reads.bam’, ‘summary.stat’

result=’success’ run(‘align ${input}’)

[20] input: align.output

if align.result == ‘success’: # do something

‘‘‘

Note that **the returned object is readonly** and cannot be changed in another step.

Option *target* (Not implemented)

This option turns the step into an [auxiliary step](Format-Specification#auxiliary-workflow-steps-and-makefile-style-dependency-rules) that is not part of the workflow but will be called if the target file is not required but does not exist.

Step input

Input files

The input of SoS step follows the following rules:

- **the input of a SoS step is by default the output of the previous step**, which is *[]* for the first step.

- **'input' option**, which could be **a list** of filenames (string literal, variables, or expressions that return filenames). Wildcard characters (*, which matches everything and *?*, which matches any single character) are always expanded. Nested lists are flattened.

Examples of input specification are as follows:

``` input: []

input: 'file1.fasta', 'file2.fasta'

input: 'file*.fasta', 'filename with space.fasta'

**input:** 'file*.txt', 'directory/file2.txt'

input: aligned_reads

input: aligned_reads, reference_genome

input: aligned_reads[2:]

input: 'data/*.fastq'

input: '*/GXT*.fastq'

input: func(parameter) ```

It is worth noting that

- The first examples shows that the step does not need any input file (so it does not depend on any other step).
- It does not matter if *aligned_reads* and *reference_genome* are strings or lists of strings because SoS will flatten nested lists to a single list of filenames.
- The *input* option tries to expand filenames with wildcard characters (* and *?*). This can be very useful for workflows that, for example, regularly scan a directory and process unprocessed files. However, because the value of this step depends on availability of files, the output of *sos show script* and the execution path will be unpredictable, and even wrong if there is no available file during the execution of *sos show script*. Option *dynamic* is required for such input files if they are meant to be determined at runtime.

The input files will be evaluated and form a list of input files. They are by default sent to the step process all at once as varible *_input*, but can also be sent in groups, each time with different *_input*. Here *_input* is a temporary variable that is available only within the step.

#### Option *filetype*

SoS allows the specification of input options, which are appended to input file list as comma separated *name=value* pairs.

Option *filetype* accepts one or more filetypes or a lambda function. For example,

``` [step] input:

    input_files, filetype='*.fastq'

```

passes only files with extension *\*.fastq*.

``` [step] input:

    input_files, filetype=['*.fastq*', '.fastq.gz']

```

passes only files with extension *.fastq* or *.fastq.gz*. Under the hood SoS treats the pattern as Unix shell-stype wildcard pattern (with *, *?*, *[seq]* and *[!seq]*, see [doc](https://docs.python.org/2/library/fnmatch.html#module-fnmatch) for details) so
```

- **‘filetype=’.txt’‘ does not match ‘file.txt‘**
- *filetype=’*.fastq*’* matches *a.fastq*, *a.fastq.gz* and *a.fastq.zip*
- *filetype=’[!_]*.txt’* matches *file1.txt* but not *_file1.txt*

If you need more refined control over the selection of files, you can use lambda functions (a bit python knowledge is required). For example,

‘‘‘ [step] input:

input_files, filetype=lambda x: open(x).readline().startswith(‘##fileformat=VCF4.1’)

‘‘‘

passes only files with the first line starting with string *##fileformat=VCF4.1‘*.

Option *group_by*

SoS by default passes all input files to step process as a single list. Option *group_by* pass input files in groups, each time with a subset of input files named *_input*. SoS allows you to group input by individual file (*single*), *pairs*, *pairwise*, and *combinations*. For example, with the following sos script

‘‘‘ [parameters] # how to group input files group = ‘all’

[0] print(‘group_by=${group}’) input: ‘file1’, ‘file2’, ‘file3’, ‘file4’, group_by=group print(‘${_index}: ${_input}’) ‘‘‘

The output of commands are

‘‘‘ $ sos dryrun test.sos -v0 group_by=all 0: file1 file2 file3 file4

$ sos dryrun test.sos –group ‘single’ -v0 group_by=single 0: file1 1: file2 2: file3 3: file4

$ sos dryrun test.sos –group ‘pairwise’ -v0 group_by=pairwise 0: file1 file2 1: file2 file3 2: file3 file4

$ sos dryrun test.sos –group ‘pairs’ -v0 group_by=pairs 0: file1 file3 1: file2 file4

$ sos dryrun test.sos –group ‘combinations’ -v0 group_by=combinations 0: file1 file2 1: file1 file3 2: file1 file4 3: file2 file3 4: file2 file4 5: file3 file4 ‘‘‘

Obviously, the output of the *pairs* cases depends on the order of files. If you need to pair files in any particular order, you can control it in input. For example

‘‘‘ [step] input:

sorted([x for x in fastq_files if ‘_R1_’ in x]), sorted([x for x in fastq_files if ‘_R2_’ in x]), group_by=’pairs’

run(‘echo ${input}’) ‘‘‘

will take all input files and sort them by *_R1_* and *_R2_* and by filename, and pair them.

Option *for_each*

Option *for_each* allows you to repeat step process for each value of a variable. For example,

```
‘ [0] method = [’m1’, ’m2’] input:  ’file1’, ’file2’, for_each=’method’
print(’${_index}:  ${_input} ${_method}’) ‘
```

will repeat the step with each item of variable *method*

```
‘ $ sos dryrun test.sos -v0 0:  file1 file2 m1 1:  file1 file2 m2 ‘
```

Note that the SoS automatically creates a loop variable *_method* for variable *method*. If the variable is an attribute of an object (e.g. *aligned.output*), the iterator variable will be named without the attribute part.

Nested loops are also allowed. For example,

```
‘python [0] method = [’m1’, ’m2’] pars = [1, 2] input:  ’file1’,
’file2’, for_each=[’method’, ’pars’] print(’${_index}:  _input=${_input}
_method=${_method}, _pars=${_pars}’) ‘
```

would produce

```
‘ $ sos dryrun test.sos -v0 0:  _input=file1 file2 _method=m1, _pars=1 1:
_input=file1 file2 _method=m2, _pars=1 2:  _input=file1 file2 _method=m1,
_pars=2 3:  _input=file1 file2 _method=m2, _pars=2 ‘
```

If you would like to loop the process with several parameters, you can put them into the same level by ‘var1,var2’. For example,

```
‘python [0] method = [’m1’, ’m2’] pars = [1, 2] input:  ’file1’, ’file2’,
for_each=’method,pars’ print(’${_index}:  _input=${_input} _method=${_method},
_pars=${_pars}’) ‘
```

would produce

```
‘ $ sos dryrun test.sos -v0 0:  _input=file1 file2 _method=m1, _pars=1 1:
_input=file1 file2 _method=m2, _pars=2 ‘
```

Option *paired_with*

Input files might might come with additional information such as sample type, and sample name, and you can pair these information to input files using the *paired_with* option. For example, *bam_files* in the following example have matched *mutated* and *sample_name*, you can atteched these information to groups of input files *_input* with looped paired with variables.

‘‘‘ [0] bam_files = [’case/A1.bam’, ‘case/A2.bam’, ‘ctrl/A1.bam’, ‘ctrl/A2.bam’] mutated = [’case’, ‘case’, ‘ctrl’, ‘ctrl’] sample_name = [’A1’, ‘A2’, ‘A1’, ‘A2’]

input: bam_files, paired_with=[’mutated’, ‘sample_name’], group_by=’pairs’

print(‘${_index}: _input=${_input} _mutated=${_mutated}, _sample_name=${_sample_name}’) ‘‘‘

Output:

```
‘ $ sos dryrun test.sos -v0 0:  _input=case/A1.bam ctrl/A1.bam _mutated=case
ctrl, _sample_name=A1 A1 1:  _input=case/A2.bam ctrl/A2.bam _mutated=case
ctrl, _sample_name=A2 A2 ‘
```

Similar to option *for_each*, if the variable is an attribute of an object (e.g. *aligned.output*), the iterator variable will be named without the attribute part (e.g. *_aligned* for *paired_with=’aligned.output’*).

Option *pattern*

This option uses named wildcards to match pattern to all input files, and creates step variables for these wildcard objects. For example,

```
‘python [step] input:  ’a-20.txt’, ’b-10.txt’, pattern = ’{name}-{par}.txt’
output:  pattern = ’{name}-processed-{par}.txt’ run(’echo ${output}; touch
${output}’) ‘
```

will take all input files and extract *name* and *par* from each file name as variables *name* and *par*. It is then used to create output file names adding the word *processed* in between these wildcard objects. The outcome of the SoS script above is creation of files *a-processed-10.txt* and *b-processed-20.txt*.

When wildcard objects are accessed as step variables, both variable names with and without _ prefix is available, e.g. in this example, both *_name* and *name*, *_par* and *par* are avaiable and are the same. The two conventions will only differ when [*group_by*](Documentation#option-group_by) or [’for_each’](Documentation#option-for_each) is also used.

```
‘python [step] input:  ’a-20.txt’, ’b-10.txt’, pattern = ’{name}-{par}.txt’,
group_by=’single’ output:  pattern = ’{_name}-processed-{_par}.txt’ run(’echo
${_output}; touch ${_output}’) ‘
```

Option *pattern* supports multiple pattern matching with different wildcards. For example:

```
‘ [step] input:  ’a-20.txt’, ’b-10.txt’, pattern = [’{name}-{par}.txt’,
’{base}.{ext}’] output:  pattern = [’{name}-processed-{par}.txt’,
’{base}-{ext}.out’] run(’echo ${_output}; touch ${_output}’) ‘
```

will produce 4 files: *a-processed-20.txt*, *b-processed-10.txt*, *a-20-txt.out*, *b-10-txt.out*, due to multiple pattern specifications.

Please note that the sigil *{}* is exclusively used for named wildcards and you should not use *${}* because it will be interpreted as strings.

Option *skip*

Option *skip* takes either a constant (*True* or *False*) or a function. Option *skip=True* will make SoS skip the execution of the current step. Using *skip=True* is not very useful so this option is often used with a SoS variable. For example

“‘python [10] input:

fasta_files, skip=len(fasta_failes) == 1

output: ‘merged.fasta’

run(‘command to merge multiple fasta files.’)

“‘

Here the *skip* option gets the value of *True* if there is only one input file. The command to merge multiple input files would then be skipped.

Another use of option *skip* is to assign it a function. In this case, this function will be applied to each input group with *_input* as the first variable and all *paired_with*, *pattern*, *loop* variables (e.g. *_loopvar*) as keyword arguments. The input group will be skipped if this function returns *False*. For example,

“‘ [10] def filter_group(ifiles, **kwargs):

return all(os.path.getfile(x) > 0 for x in ifiles)

input: group_by=’combinations’, skip=filter_group “‘

will check all input groups and skip groups with one or two empty files (file size = 0).

Option *dynamic*

In order to determine the best execution strategy, SoS evaluates all expressions for all steps before the execution of a workflow to figure out input and output of steps. This works most of the time but sometimes the input of a step can only be determined at runtime. For example, if you would like your workflow to automatically scan an input directory and process all fasta files under it, or if a previous step produces files that cannot be determined beforehand, you can specify input files as follows,

```
‘python input:  ’input/*.fasta’ ‘
```

The problem is that no file or a wrong set files might exist during the planing stage so SoS might skip this step or start the step with a wrong set of files. To address this problem, you can declare the input of this step as **dynamic** by passing option *dynamic=True* to it,

```
‘python input:  ’input/*.fasta’, dynamic=True ‘
```

This tells SoS that the input of this step can only be determined at runtime and will execute the step only after all its previous steps have been completed.

Summary

Options of step *input* are evaluated in the following orders:

1. A list of input files, if specified, would replace *input*, which is by default output from the previous step.
2. Option *filetype* filters input files. **The output becomes 'input'**.
3. Option *group_by* groups the files into several groups, named *_input*
4. Option *for_each* repeat *_input* for each loop var, named *_loopvar* if *for_each='loopvar'*.
5. **Option *paired_with* pairs one or more variables with *input*, variable *paired* is paired with *input*** and variable *_paired* is paired with *_input* in each loop if *paired_with='paired'*
6. **Option *pattern* extract variables from filenames in *input*. Variable *extracted* is paired with *input*** and variable *_extracted* is paired with *_input* in each loop if *pattern='{extracted}_other_part'*.
7. Option *skip* optionally skip all or part of the input groups.

The differences between looped and non-loop steps are sumarized in the following figure

[[/media/step_loop.jpg]]

Step output

Output files

Output files of a step can be specified by step *output*. Whereas step *input* override or change SoS provided variable *input* to produce one or more variable *_input*, the step *output* specify *output* which should be *[]* if no output is generated (or of interest). Similar to *input*, step output accepts strings, variables, expressions, and allows wildcard characters. For example, the following are acceptable output files

```python output: []

output: 'accepted_hits.bam'

output: aligned_reads, bam_stats

output: 'aligned/*.bam' ```

In case of *input* loop, step *output* actually generates variables *_output* for each input loop. *output* is the sum of all *_output* with duplicated filenames removed. For example, the following step accepts one or more bam files and index them using command *samtools index*. The input files are passed one by one so generate multiple *output* files (*'${_input}.bai'*). The output of this step is then the collection of all such output files.

```python [10] input:
    bamfiles, group_by='single'
output: '${_input}.bai'
run(''samtools index ${_input} '') ```

Note that you can specify all output files, e.g.

```
`python output:  [x + '.bai' for x in bamfiles] `
```

but specifying *_output* for each *_input* is usually easier and allows better execution of the repeated steps.

Option *pattern*

Option pattern treats variables in *{}* in the passed patterns (more than one pattern is allowed) as wildcard variables and generate output filenames from items of these variables. For example, if *samples=['A', 'B', 'C']* and *temp=[10, 20, 20]*,

```
` pattern='{}-{}-result.txt' `
```

would produce three output files.

```
` A-10-result.txt B-20-result.txt C-20-result.txt `
```

This feature is usually used with [input pattern](Documentation#option-pattern) to derive output files from input files.

Option *dynamic*

Similar to the cases with [dynamic input files](#dynamically-determined-input-files-option-dynamic), the output of some steps could also not be determined beforehand. For example, with the following script that generates *html* files that cannot be determined during dry run,

‘‘‘python [0] run:

rm -f *.html

[10] input: [] output: ‘*.html’

import random for i in range(5):

run(‘touch result_${random.randint(1, 20)}.html’)

‘‘‘

SoS will determine that you do not have any output file (no *.*html* file) and produce the following output

```
`bash $ sos run test.sos INFO: Execute default_0:  INFO: input:  [] INFO:
output:  unspecified INFO: Execute default_10:  INFO: input:  [] WARNING:
*.html does not expand to any valid file.  INFO: output:  [] `
```

In this case, you will need to append option *dynamic=True* to the step output

‘‘‘ [0] run:

rm -f *.html

[10] input: [] output: ‘*.html’, dynamic=True

import random for i in range(5):

run(‘touch result_${random.randint(1, 20)}.html’)

‘‘‘

so that SoS knows that the output can only be determined after the completion of the step. Because of this, variable *output* is unavailable to the step process.

Step dependencies

This item specifies files that are required for the step. Although not required, it is a good practice to list resource files and other dependency files for a particular step. For example

‘‘‘python [10] input:

fasta_files

depends: reference_seq

‘‘‘

Similar to *output* options, dependent files can also be defined after *input* options and consist of dependent files determined from loop variables.

The following figure summarizes the effect of *input* and *output* options and input options *group_by* and *for_each* on the flow of input and output files and related variables.

[[/media/step_options.jpg]]

Step process

Step *process*

The *process* marks the beginning of step process, with optional runtime options to control its execution. For example,

‘‘‘ [10] input: group_by=’single’ process: concurrent=True

run(‘’’ samtools index {_input} ’’’) ‘‘‘

execute a shell script in parallel (with *concurrent=True*). The step process can consists of arbitrary python statements and execute multiple step actions. For example,

‘‘‘python process: try:

    action1()

except RuntimeError: action2()

‘‘‘

execute *action1* and *action2* if *action1* raises an error.

‘‘‘python process: for par in [’-4’, ‘-6’]:

    run(‘command with ${par}’)

‘‘‘

executes commands in a loop. This is similar to

```
` pars = ['-4', '-6'] input:  for_each=pars process:  run('command with
${_pars}') `
```

but the *for* loop version would not be able to be executed in parallel. Note that SoS actions can be used outside of *step process* but only statements specified after the *process* keyword can have runtime options and be executed in separate processes. That is to say,

```
` pars = ['-4', '-6'] input:  for_each=pars run('command with ${_pars}') `
```

is equivalent to

```
` pars = ['-4', '-6'] input:  for_each=pars process:  run('command with
${_pars}') `
```

but the latter can have additional runtime options to run commands in parallel

```
` pars = ['-4', '-6'] input:  for_each=pars process:  concurrent=True
run('command with ${_pars}') `
```

Script format

SoS allows you to write *process step* in a special script format. For example,

‘‘‘ [10] input: group_by=’single’

process: concurrent=True R(‘’’ pdf(‘${input}’) plot(0, 0) dev.off() ’’’) ‘‘‘

can be written as

```
` [10] input:  group_by='single' R: concurrent=True pdf('${_input}') plot(0,
0) dev.off() `
```

The script is a string without quotation marks and the normal string interpolation will take place. Also, SoS will dedent the text (remove all common leading white spaces) so you can write the example as

‘‘‘ [10] input:

    group_by=’single’

R: concurrent=True pdf(‘${_input}’) plot(0, 0) dev.off()

‘‘‘

if you prefer. Because all SoS keywords starts from the first column, indentation can help avoid trouble if your script contains strings such as *[1]* and *option:*.

More actions can be written in this way but runtime options can only be specified in the first action. For example, the following step calls two actions *run* and *R* in a step process.

‘‘‘ [10] input: group_by=’single’ run: concurrent=True

    samtools index {_input}

R: pdf(‘${_input}.pdf’) plot(0, 0) dev.off()

‘‘‘

Although the script format is more concise and easier to read, it is limited to actions that accept a single string as input and is less flexible (no flow control etc).

SoS accepts a list of runtime options that controls how the step process is executed. #### Option *workdir*

Default to current working directory.

Option *workdir* controls the working directory of the process. For example, the following step downloads a file to the *resource_dir* using command *wget*.

‘‘‘python [10]

run: workdir=resource_dir

    wget a_url -O filename

‘‘‘

Option *concurrent*

Default to *False*.

If the step process is repeated for different input files or parameters (using input options *group_by* or *for_each*), the loop process can be execute in parallel, up to the maximum number of concurrent jobs specified by command line option *-j*.

Option *docker_image*

If a docker image is specified (either a name, an Id, or a file), the action is assumed to be executed in the specified docker. The image will be automatically downloaded (pulled) or loaded (if a *.tar* or *.tar.gz* file is specified‘) if it is not available locally. Note that this option only affect script executing actions such as *run*, *python* and *perl* so other actions such as *check_command* will be executed on the host machine even if it is included in the step process.

For example, executing the following script

‘‘‘ [10] python3: docker_image=’python’

    set = {‘a’, ‘b’} print(set)

‘‘‘

under a docker terminal (that is connected to the docker daemon) will

1. Pull docker image *python*, which is the official docker image for Python 2 and 3.
2. Create a python script with the specified content
3. Run the docker container *python* and make the script available inside the container
4. Use the *python3* command inside the container to execute the script.

Additional *docker_run* parameters can be passed to actions when the action is executed in a docker image. These options include

- *name*: name of the container (option *–name*)
- *tty*: if a tty is attached (default to *True*, option *-t*)
- *stdin_open*: if stdin should be open (default to *False*, option *-i*)
- *user*: username (default o *root*, option *-u*)
- *environment*: Can be a string, a list of string or dictinary of environment variables for docker (option *-e*)
- *volumes*: string or list of string, extra volumes that need to be link, in addition to SoS mounted (*/tmp*, */Users* (if mac), */Volumes* (if [properly configured](https://github.com/bpeng2000/SOS/wiki/SoS-Docker-guide) under mac) and script file)
- *volumes_from*: container names or Ids to get volumes from
- *working_dir*: working directory (option *-w*), default working directory, or working directory set by runtime option *workdir*.
- *port*: port opened (option *-p*)
- *extra_args*: If there is any extra arguments you would like to pass to the *docker run* process (after you check the actual command of *docker run* of SoS

Option *docker_file*

This option allows you to import a docker from specified *docker_file*, which can be an archive file (*.tar*, *.tar.gz*, *.tgz*, *.bzip*, *.tar.xz*, *.txz*) or a URL to an archive file (e.g. *http://example.com/exampleimage.tgz*). SoS will use command *docker import* to import the *docker_file*. However, because SoS does not know the repository and tag names of the imported docker file, you will still need to use option *docker_image* to specify the image to use.

Option *blocking* (Not implemented)

Default to *False*.

The step process can only be executed by one instance of SoS. All other SoS instances will wait until one instance complete this step. Option *blocking=True* should be used for processes such as the creation of file index and downloading of resources.

Step action

Although arbitrary python functions can be used in SoS step process, SoS defines some **'actions'** (e.g. the *run* function in the aforementioned examples) that can be used in a SoS script. The only difference between a SoS action and a regular Python function is that they can behave differently in different *run_mode*. For example,

- Most SoS actions return 0 directly in *dryrun* mode.
- Some SoS actions such as *check_command* work in both *run* and *dryrun* mode so that you can check if you script can be executed by running it in dryrun mode.

Because SoS executes all step processes in dryrun mode for the planning of large workflows, **it is important to enclose data processing steps in SoS actions**. Otherwise these python statement might be executed multiple times.

It is easy to define your own actions. All you need to do is to define a function and decorate it with a *SoS_Action* decorator. For example

'''python

from pysos import SoS_Action

@SoS_Action(run_mode='run') def my_action(parameters):

    do_something_with_parameters return 1

‘‘‘

run_mode='run' can also be *run_mode=('dryrun', 'run')* or *run_mode='dryrun'* indicating in which mode the function will be executed (instead of returning 0 directly).

Action *sos_run*

Action *sos_run(workflow, source)* executes a specified workflow. The workflow can be a single workflow, a subworkflow (e.g. *A_-10*), or a combined workflow (e.g. *A + B*). Because the workflow is executed from a step, it takes step *_input* as the input of the nested workflow and it can access local step variables. For example,

‘‘‘ import random

[simulate] output: 'result_${_reps}.txt' run: simulate_experiment –seed=${seed} –output=${_output}

[10] reps = range(100) input: for_each='reps' outout: 'result_${_reps}.txt'

seed = random.randint(1, 2**32) sos_run('simulate') ‘‘‘

would run the nested pipeline *simulate* (which is a single step in this example) 100 times with their own *seed*, and *_reps*.

The workflow can be defined in the current script, or in other SoS scripts, in which case the name or full path of the SoS script should be provided to parameter *source*. For example,

```
` [myworkflow] sos_run('A+B', source="AB.sos") `
```

defines a nested workflow with workflow *A* and/or *B* defined in *AB.sos*. The nested workflow is a combination of two workflows *A* and *B* with their own parameters sections. SoS searches the specified files in the current working directory, the directory of the master script, and a search path specified by variable *sos_path* defined in the SoS global configuration file (*~/.sos/config.json*), and will produce an error if no file can be found.

Action *run*, *bash*, *sh*, *csh*, *tcsh*, *zsh*

Actions *run(script)* and *bash(script)* accepts a shell script and execute it using *bash*. *sh*, *csh*, *tcsh*, *zsh* uses respective shell to execute the provided script.

Action *python* and *python3*

Action *python(script)* and *python3(script)* accepts a Python script and execute it with python or python3, respectively.

Because SoS can include Python statements directly in a SoS script, it is important to note that embedded Python statements are interpreted by SoS and the *python* and *python3* actions are execute in separate processes without access to the SoS environment.

For example, the following SoS step execute some python statements **within** SoS with direct access to SoS variables such as *input*, and with *result* writing directly to the SoS environment,

‘‘‘python [10] for filename in input:

    with open(filename) as data:

‘‘‘

while

‘‘‘python [10] input: group_by='single'

python:

with open(${input!r}) as data: result = ${input!r} + '.res' ...

‘‘‘

composes a Python script for each input file and calls separate Python interpreters to execute them. Whereas the Python statement in the first example will always be executed, the statements in *python* will not be executed in *dryrun* mode.

Action *R* and *check_R_library*

Action *R(script)* exexute the passed script using *Rscript* command. Action *check_R_library* accept an R library and optionally can check for required versions. If the libraries are not available, it will try to install it from [CRAN](https://cran.r-project.org/), [bioconductor](https://www.bioconductor.org/), or [github](https://github.com/). Github package name should be formatted as *repo/pkg*. Action *check_R_library* is available in both *run* and *dryrun* mode.

For example, *check_R_library('edgeR')* will check and install (if necessary) *edgeR* from bioconductor. *check_R_library('stephens999/ashr')* will check and install *ashr* package from a github repository https://github.com/stephens999/ashr.

check_R_library can also be used to check for required version of packages. For example:

` check_R_library('edgeR', '3.12.0') ` will result in a warning if edgeR version is not 3.12.0, and

Multiple versions is allowed, for example: ` check_R_library('edgeR', ['3.12.0', '3.12.1']) ` Or, simply: ` check_R_library('edgeR', '3.12.0+') ` It is also possible to restrict package to old version, for example: ` check_R_library('ggplot2', '1.0.0-') `

Note that version mismatch triggers a warning message, not an error. If you would like to enforce certain version, use

```
` fail_if(check_R_library('ggplot2', '1.0.0+') != 0, 'Version 1.0.0 or newer
version of ggplot2 is required') `
```

Action *perl*, *ruby*

Action *perl(script)* execute the passed script using *perl* interpreter. Action *ruby(script)* execute the passed script using *ruby* interpreter.

Action *node*, *JavaScript*

Action *node(script)* and *JavaScript(script)* execute the passed script using *node* interpreter.

Action *docker_build*

Build a docker image from an inline Docker file. The inline version of the action currently does not support adding any file from local machine because the docker file will be saved to a random directory. You can walk around this problem by creating a *Dockerfile* and pass it to the action through option *path*. This action accepts all parameters as specified in https://docker-py.readthedocs.org/en/stable/api/#build because SoS simply pass additional parameters to the *build* function.

For example, the following step builds a docker container for [MISO](http://miso.readthedocs.org/en/fastmiso/) based on anaconda python 2.7.

‘‘‘ [build_1] # building miso from a Dockerfile docker_build: tag='mdabioinfo/miso:latest'

> # Dockerfile to build MISO container images # Based on Anaconda python ##
>
> # Set the base image to anaconda Python 2.7 (miso does not support python 3) FROM continuumio/anaconda
>
> # File Author / Maintainer MAINTAINER Bo Peng <bpeng@mdanderson.org>
>
> # Update the repository sources list RUN apt-get update
>
> # Install compiler and python stuff, samtools and git RUN apt-get install –yes
>
> > build-essential gcc-multilib gfortran apt-utils libblas3 liblapack3 libc6 cython samtools libbam-dev bedtools wget zlib1g-dev tar gzip
>
> WORKDIR /usr/local RUN pip install misopy

‘‘‘

Action *download*

Action *download(URLs, dest_dir='.', dest_file=None, decompress=False)* download files from specified URLs, which can be a list of URLs, or a string with tab, space or newline separated URLs.

- If *dest_file* is specified, only one URL is allowed and the URL can have any form.
- Otherwise all files will be downloaded to *dest_dir*. Filenames are determined from URLs so the URLs must have the last portion as the filename to save.
- If *decompress* is True, *.zip* file, compressed or plan *tar* (e.g. *.tar.gz*) files, and *.gz* files will be decompressed to the same directory as the downloaded file.

For example,

‘‘‘ [10] GATK_RESOURCE_DIR = ‘/path/to/resource’ GATK_URL = ‘ftp://gsapubftp-anonymous@ftp.broadinstitute.org/bundle/2.8/hg19/‘

download: dest=GATK_RESOURCE_DIR ${GATK_URL}/1000G_omni2.5.hg19.sites.vcf.gz ${GATK_URL}/1000G_omni2.5.hg19.sites.vcf.gz.md5 ${GATK_URL}/1000G_omni2.5.hg19.sites.vcf.idx.gz ${GATK_URL}/1000G_omni2.5.hg19.sites.vcf.idx.gz.md5 ${GATK_URL}/dbsnp_138.hg19.vcf.gz ${GATK_URL}/dbsnp_138.hg19.vcf.gz.md5 ${GATK_URL}/dbsnp_138.hg19.vcf.idx.gz ${GATK_URL}/dbsnp_138.hg19.vcf.idx.gz.md5 ${GATK_URL}/hapmap_3.3.hg19.sites.vcf.gz ${GATK_URL}/hapmap_3.3.hg19.sites.vcf.gz.md5 ${GATK_URL}/hapmap_3.3.hg19.sites.vcf.idx.gz ${GATK_URL}/hapmap_3.3.hg19.sites.vcf.idx.gz.md5

‘‘‘

download the specified files to *GATK_RESOURCE_DIR*. The *.md5* files will be automatically used to validate the content of the associated files. Note that

SoS automatically save signature of downloaded and decompressed files so the files will not be re-downloaded if the action is called multiple times. You can however still still specifies input and output of the step to use step signature

‘‘‘ [10] GATK_RESOURCE_DIR = ‘/path/to/resource’ GATK_URL = ‘ftp://gsapubftp-anonymous@ftp.broadinstitute.org/bundle/2.8/hg19/‘ RESOUCE_FILES = ‘‘‘1000G_omni2.5.hg19.sites.vcf.gz 1000G_omni2.5.hg19.sites.vcf.gz.md5 1000G_omni2.5.hg19.sites.vcf.idx.gz 1000G_omni2.5.hg19.sites.vcf.idx.gz.md5 dbsnp_138.hg19.vcf.gz dbsnp_138.hg19.vcf.gz.md5 dbsnp_138.hg19.vcf.idx.gz dbsnp_138.hg19.vcf.idx.gz.md5 hapmap_3.3.hg19.sites.vcf.gz hapmap_3.3.hg19.sites.vcf.gz.md5 hapmap_3.3.hg19.sites.vcf.idx.gz hapmap_3.3.hg19.sites.vcf.idx.gz.md5’‘’.split()

input: [] output: [os.path.join(GATK_RESOURCE_DIR, x) for x in GATK_RESOURCE_FILES] download([’${GATK_URL}/${x}’ for x in GATK_RESOURCE_FILES], dest=GATK_RESOURCE_DIR) ‘‘‘

Note that the *download* action uses up to 5 processes to download files. You can change this number by adjusting system configuration *sos_download_processes*.

Action *check_command*

Action *check_command(cmd, pattern)* check existence of command or the output of the command. This action works in both dyrun and run mode so it can be used to check the existence of command in dryrun mode.

For example, if a script contains step

```
‘python [100] check_command(’tophat’) run(’tophat ...’)  ‘
```

Command ‘sos run script -d’ would check the existence of command *tophat* without actually running the *tophat ...* command.

Action *check_command(cmd, pattern)* execute specified *cmd* and searches specified pattern in its output. *pattern* should be one or a list of python regular expressions (see [Python re module](https://docs.python.org/2/library/re.html), especially the *search* function for details). It raises an exception if the output does not match any of the patterns. This function is usually used to check the version of commands.

For example, action

` check_command('STAR --version', ['2.4.0', '2.5.0']) ` checks the output of command *STAR –version* and raises an error if it does not contain string *2.4.0* or *2.5.0*.

Note that

1. *check_command* will return a non-zero value if the command exist but returns non-zero value (e.g. because of incorrect command line argument). A warning message will also be printed.
2. *check_command* will timeout after 2 seconds because it is intended to check basic information of specified command. The action will produce a warning message and return 1 in this case.

Action *fail_if*

Action *fail_if(expr, msg='')* raises an exception with *msg* (and terminate the execution of the workflow if the exception is not caught) if *expr* returns True.

Action *warn_if*

Action *warn_if(expr, msg)* yields a warning message *msg* if *expr* is evaluate to be true.

Utility functions and *logger*

SoS exposes a few utility functions that can be helpful from time to time.

Function *get_output*

Function *get_output(cmd)* returns the output of command (decoded in *UTF-8*), which is a shortcut for *subprocess.check_output(cmd, shell=True).decode()*. It is worth noting that SoS kills any function call after 5 seconds in dryrun mode so you will need to put this function call inside a step process if it will take more than 5 seconds to execute.

Function *expand_pattern*

Function *expand_pattern* is the function alternative to output option *pattern*.

`'python output: expand_pattern('{a}_{b}.txt') `

is equivalent to

`'python output: pattern='{a}_{b}.txt' `

It can however be used if you would like to post-process outcome of the *pattern* option, for example,

` output: [x for x in expand_pattern('{a}_{b}.txt') if '200' not in x] `

removes output files if the output file contains *200*.

SoS *logger* object

The SoS logger object is a *logging* object used by SoS to produce various outputs. You can use this object to output error, warning, info, debug, and trace messages to terminal. For example,

` [0] logger.info('I am at ${step_name}') `

would print a logging message *I am at default_0* when the first step is execute.

`'bash $ sos run test.sos INFO: Execute default_0: INFO: input: [] INFO: I am at default_0 INFO: output: unspecified `

Auxiliary workflow steps (Not implemented)

Auxiliary steps are special steps that are used only when a target is needed but not available. Such steps are defined in the format of

‘‘‘python [step_name_without_index : target=pattern]

step_input = expression involving step_output

input: depends: action()

‘‘‘

where

- Step name does not have an index.
- An option *target* specifies that pattern of files that triggers the step.
- There is no *output* option.
- *step_input* should be explicitly calculated from a SoS provided *step_output*

For example,

‘‘‘python [index_bam : target=’*.bam.bai’] # # index bam file if its index is needed

input file should be filename.bam if the output is filename.bam.bai step_input = [x[:-4] for x in step_output]

input: group_by=’single’

create ${_input}.bai from ${_input} run:

samtools index ${_input}

‘‘‘

defines a step called *index_bam*. When a file with extension **.bam.bai* is required but does not exist, for example when the following step

‘‘‘python [align_100]

depends: input[0] + ‘.bai’ ...

‘‘‘

is executed with input *[’A1.bam’]*, SoS will check if there is an auxiliary step to produce it and call that step with *step_output=[’A1.bam.bai’]* but no *step_input*. In this example the *index_bam* step will figure out what the input files are needed (*step_input=...*) and execute the step as a regular SoS step if files specified by *step_input* exist, and execute other auxiliary steps to produce required files otherwise.

You might have already realized that an auxiliary step is a makefile style step and you can use this technique to build complete workflows in a way similar to [snakemake](https://bitbucket.org/johanneskoester/snakemake). That is to say, you can define multiple auxiliary steps (rules in snakemake’s term) and let SoS determine what steps to execute depending on what workflow target to produce. You can even mix the forward, input-oriented style with backward, output-oriented style in SoS. However, **auxiliary steps are designed to be helper steps that could be called multiple times during the execution of a workflow**. If you are strongly inclined to the makefile-like rule-based workflow system, make or snakemake should be better because they are specifically designed around that paradigm.

Execution of workflows

dryrun, *prepare*, and *run* mode

A sos script will be by default executed three times, in *dryrun*, *prepare* and *run* modes. Besically,

- The *dryrun* mode will check syntax errors of the script and execute actions that are executable in *dryrun* mode (e.g. actions such as *check_command* to check the existence of commands).

- The *prepare* mode will prepare the execution of the script by, for example, downloading required resources and execute actions that are executable in *prepare* mode (e.g. action *download* to download required resources).
- The *run* mode will execute all steps.

Syntax and basic execution errors are collected in *dryrun* and *prepare* mode to be reported altogether, whereas other runtime errors will terminate the execution of script in *run* mode. In addition, **all expressions and statements will be terminated after 5 seconds in 'dryrun' mode** so all time-consuming actions should be executed only in *run* mode. If you are on a particularly slow machine, you can relax this by setting an option *sos_dryrun_timeout* in *~/.sos/config.json* (see section [configuration file](#configuration-file) for details.

For example, with the following (incomplete) script

‘‘‘python [0] download('hg19 reference genome')

[10] check_command('fastqc') run:

    fastqc ...

[20] run: docker_image='ubuntu'

    script running in docker

[30] check_command('tophat') run:

    tophat ...

‘‘‘

Running this script will * execute the script in *dryrun* mode to check the existence of commands *fastqc* and *tophat* and terminate if any of the commands does not exist. * execute the script in *prepare* mode to execute *download* action to download hg19 reference genome and pull the *ubuntu* docker image. * execute the script in *run* mode to execute all steps.

You can execute a script in different mode using different subcommands `bash sos dryrun script sos prepare script sos run script `

or using options of the *run* subcommand

```
`bash # dryrun mode only sos run -d script # dryrun and prepare mode only sos run -p script # dryrun prepare and run mode (default) sos run script `
```

or using the *sos-runner* command

```
`bash sos-runner -d script sos-runner -p script sos-runner script `
```

Configuration file

SoS allows you to create a file *~/.sos/config.json* as the global configuration file for all SoS scripts. The content of this file will be read before the execution of any SoS script and be available in the *CONFIG* variable.

SoS currently support the following configuration variables

- *sos_path* (default to *[]*): A list of directories from which sos will try to locate a script if the script is not in the current directory. For example, if you set *sos_path* to *['~/scripts']*, *sos run myscript.sos* will execute *~/scripts/myscript.sos* if *myscript.sos* does not exist in the current directory.
- *sos_dryrun_timeout* (default to *5*): Time in seconds SoS waits for the completion of a SoS statement or action in *dryrun* mode before it terminates the execution
- *sos_download_processes* (default to *5*): Number of download processes to download files specified in action *download*.

The default value of 5 seconds for option *sos_dryrun_timeout* is usually good enough but you can set it to a longer time if you are working on a machine with, for example, slow disk access. To change this option, you will need to

- Create a file *~/.sos/config.json* if it does not exist.
- Replace the content of the file with the following or add the key to the file. Note that JSON requires double quote for strings so *'sos_dryrun_timeout'* is not allowed.

‘‘‘json # # global sos configuration file # {

“sos_dryrun_timeout”: 10

You can put machine-dependent settings in this file (e.g. password to a MySQL database) to be used by all SoS scripts but please keep in mind that your script will become less portable with these global settings.

Parallel execution Logically speaking, for a given script and workflow, SoS will

- Evaluate parameters with either command line input or their default values
- Evaluate the rest of steps in numeric order

Actual execution order depends on step actions, input output options of SoS steps, and option *-j*. In the sequential execution mode (by default), all steps are executed one by one with auxillary steps called when necessary. If a script is written without any step option and input and output files, it will be executed sequentially even in parallel execution mode (with *-j* option).

If the steps are described with necessary input and output information, steps in SoS workflows can be executed in parallel. The following figure illustrates the impact input/output options on the execution order of workflows. Note that a step with unknown *input* (no *input* at present step and no *output* at previous step) can only be executed after all its previous steps are completed can become bottlenecks of the workflow.

[[/media/workflow.jpg]]