Sunbeam Documentation

Release 2.1

Erik Clarke, Chunyu Zhao, Jesse Connell, Louis Taylor

Jan 30, 2020

Contents:

Sunbeam is a pipeline written in snakemake that simplifies and automates many of the steps in metagenomic sequencing analysis. Sunbeam requires a reasonably modern GNU/Linux computer with bash, Python 2.6+, internet access (to retrieve dependencies), 4Gb of RAM, and at least 3Gb of disk space. RAM and disk space requirements may increase depending on the databases and tasks you choose to run, and the size of your data. For more information, check out the Sunbeam paper in Microbiome.

Sunbeam currently automates the following tasks:

- Quality control, including adaptor trimming, host read removal, and quality filtering;
- Taxonomic assignment of reads to databases using Kraken;
- Assembly of reads into contigs using Megahit;
- Contig annotation using BLAST[n/p/x];
- Mapping of reads to target genomes; and
- ORF prediction using Prodigal

Sunbeam was designed to be modular and extensible. We have a few pre-built *Sunbeam Extensions* available that handle visualization tasks, including contig assembly graphs, read alignments, and taxonomic classifications.

To get started, see our *Quickstart Guide*!

If you use Sunbeam in your research, please cite:

EL Clarke, LJ Taylor, C Zhao *et al.* Sunbeam: an extensible pipeline for analyzing metagenomic sequencing experiments. *Microbiome* 7:46 (2019)

CHAPTER 1

Quickstart Guide

Contents

1.1 Installation

On a Linux machine, download a copy of Sunbeam from our GitHub repository, and install. We do not currently support non-Linux environments.

```
git clone -b stable https://github.com/sunbeam-labs/sunbeam sunbeam-stable
cd sunbeam-stable
./install.sh
tests/run_tests.bash -e sunbeam
```

This installs Sunbeam and all its dependencies, including the Conda environment manager, if required. It then runs some tests to make sure everything was installed correctly.

Tip: If you've never installed Conda before, you'll need to add it to your shell's path. If you're running Bash (the most common terminal shell), the following command will add it to your path: `echo 'export PATH=$PATH:$HOME/miniconda3/bin' > ~/.bashrc`

If you see "Tests failed", check out our troubleshooting section or file an issue on our GitHub page.

1.2 Setup

Let's say your sequencing reads live in a folder called `/sequencing/project/reads`, with one or two files per sample (for single- and paired-end sequencing, respectively). These files *must* be in gzipped FASTQ format.

Let's create a new Sunbeam project (we'll call it `my_project`):

```
source activate sunbeam
sunbeam init my_project --data_fp /sequencing/project/reads
```

Sunbeam will create a new folder called `my_project` and put two files there:

- `sunbeam_config.yml` contains all the configuration parameters for each step of the Sunbeam pipeline.
- `samples.csv` is a comma-separated list of samples that Sunbeam found the given data folder, along with absolute paths to their FASTQ files.

Right now we have everything we need to do basic quality-control and contig assembly. However, let's go ahead and set up contaminant filtering and some basic taxonomy databases to make things interesting.

1.2.1 Contaminant filtering

Sunbeam can align your reads to an arbitrary number of contaminant sequences or host genomes and remove reads that map above a given threshold.

To use this, make a folder containing all the target sequences in FASTA format. The filenames should end in "fasta" to be recognized by Sunbeam. In your `sunbeam_config.yml` file, edit the `host_fp:` line in the `qc` section to point to this folder.

1.2.2 Taxonomic classification

Sunbeam can use Kraken to assign putative taxonomic identities to your reads. While creating a Kraken database is beyond the scope of this guide, pre-built ones are available at the Kraken homepage. Download or build one, then add the path to the database under `classify:kraken_db_fp:`.

1.2.3 Contig annotation

Sunbeam can automatically BLAST your contigs against any number of nucleotide or protein databases and summarize the top hits. Download or create your BLAST databases, then add the paths to your config file, following the instructions on here: *blastdbs*. For some general advice on database building, check out the Sunbeam databases repository and for specific links please see the usage section: *Building Databases*.

1.2.4 Reference mapping

If you'd like to map the reads against a set of reference genomes of interest, follow the same method as for the host/contaminant sequences above. Make a folder containing FASTA files for each reference genome, then add the path to that folder in `mapping:genomes_fp:`.

1.3 Running

After you've finished editing your config file, you're ready to run Sunbeam:

```
sunbeam run --configfile my_project/sunbeam_config.yml
```

By default, this will do a lot, including trimming and quality-controlling your reads, removing contaminant, host, and low-complexity sequences, assigning read-level taxonomy, assembling the reads in each sample into contigs, and then BLASTing those contigs against your databases. Each of these steps can also be run independently by adding arguments after the `sunbeam run` command. See *Running* for more info.

1.4 Viewing results

The output is stored by default under `my_project/sunbeam_output`. For more information on the output files and all of Sunbeam's different parts, see our full *User Guide*!

CHAPTER 2

User Guide

Contents

* *Taxonomic classification*
* *Alignment to genomes*
* *Quality control*

2.1 Requirements

- A relatively-recent Linux computer with more than 2Gb of RAM

We do not currently support Windows or Mac. (You may be able to run this on Windows using the [WSL](https://docs.microsoft.com/en-us/windows/wsl/about), but it has not been tested.

2.2 Installation

Clone the stable branch of Sunbeam and run the installation script:

```
git clone -b stable https://github.com/eclarke/sunbeam sunbeam-stable
cd sunbeam-stable
bash install.sh
```

The installer will check for and install the three components necessary for Sunbeam to work. The first is Conda, a system for downloading and managing software environments. The second is the Sunbeam environment, which will contain all the required software. The third is the Sunbeam library, which provides the necessary commands to run Sunbeam.

All of this is handled for you automatically. If Sunbeam is already installed, you can upgrade either or both the Sunbeam environment and library by passing `--upgrade [env/lib/all]` to the install script.

If you don't have Conda installed prior to this, you will need to add a line (displayed during install) to your config file (usually in `~/.bashrc` or `~/.profile`). Restart your terminal after installation for this to take effect.

2.2.1 Testing

We've included a test script that should verify all the dependencies are installed and Sunbeam can run properly. We strongly recommend running this after installing or updating Sunbeam:

```
bash tests/run_tests.bash
```

If the tests fail, you should either refer to our **troubleshooting_** guide or file an issue on our Github page.

2.2.2 Updating

Sunbeam follows semantic versioning practices. In short, this means that the version has three numbers: major, minor and patch. For instance, a version number of 1.2.1 has 1 as the major version, 2 as the minor, and 1 as the patch.

When we update Sunbeam, if your config files and environment will work between upgrades, we will increment the patch or minor numbers (e.g. 1.0.0 -> 1.1.0). All you need to do is the following:

```
git pull
./install.sh --upgrade all
```

If we make a change that affects your config file (such as renaming keys or adding a new section), we will increase the major number (e.g. 1.1.0 -> 2.0.0). When this occurs, you can use the same update procedure as before, and then update your config file to the new format:

```
git pull
./install.sh --upgrade all
source activate sunbeam
sunbeam config upgrade --in_place /path/to/my_config.yml
```

It's a good idea to re-run the tests after this to make sure everything is working.

2.2.3 Uninstalling or reinstalling

If things go awry and updating doesn't work, simply uninstall and reinstall Sunbeam.

```
source deactivate
conda env remove --name sunbeam
rm -rf sunbeam-stable
```

Then follow the *installation* instructions above.

2.2.4 Installing Sunbeam extensions

As of version 3.0, Sunbeam extensions can be installed by running `sunbeam extend` followed by the URL of the extension's GitHub repo:

```
sunbeam extend https://github.com/sunbeam-labs/sbx_kaiju/
```

For Sunbeam versions prior to 3.0, follow the instructions on the extension to install.

2.3 Setup

2.3.1 Activating Sunbeam

Almost all commands from this point forward require us to activate the Sunbeam conda environment:

```
source activate sunbeam
```

You should see '(sunbeam)' in your prompt when you're in the environment. To leave the environment, run `source deactivate` or close the terminal.

2.3.2 Creating a new project using local data

We provide a utility, `sunbeam init`, to create a new config file and sample list for a project. The utility takes one required argument: a path to your project folder. This folder will be created if it doesn't exist. You can also specify the path to your gzipped fastq files, and Sunbeam will try to guess how your samples are named, and whether they're paired.

```
sunbeam init --data_fp /path/to/fastq/files /path/to/my_project
```

In this directory, a new config file and a new sample list were created (by default named `sunbeam_config.yml` and `samplelist.csv`, respectively). Edit the config file in your favorite text editor- all the keys are described below.

Note: Sunbeam will do its best to determine how your samples are named in the `data_fp` you specify. It assumes they are named something regular, like `MP66_S109_L008_R1_001.fastq.gz` and `MP66_S109_L008_R2_001.fastq.gz`. In this case, the sample name would be 'MP66_S109_L008' and the read pair indicator would be '1' and '2'. Thus, the filename format would look like `{sample}_R{rp}_001.fastq.gz`, where {sample} defines the sample name and {rp} defines the 1 or 2 in the read pair.

If you have single-end reads, you can pass `--single_end` to `sunbeam init` and it will not try to identify read pairs.

If the guessing doesn't work as expected, you can manually specify the filename format after the `--format` option in `sunbeam init`.

Finally, if you don't have your data ready yet, simply omit the `--data_fp` option. You can create a sample list later with `sunbeam list_samples`.

If some config values are always the same for all projects (e.g. paths to shared databases), you can put these keys in a file and auto-populate your config file with them during initialization. For instance, if your Kraken databases are located at `/shared/kraken/standard`, you could have a file containing the following called `common_values.yml`:

```
classify:
  kraken_db_fp: "/shared/kraken/standard"
```

When you make a new Sunbeam project, use the `--defaults common_values.yml` as part of the init command.

If you have Sunbeam extensions installed, in Sunbeam >= 3.0, the extension config options will be automatically included in new config files generated by `sunbeam init`.

Further usage information is available by typing `sunbeam init --help`.

2.3.3 Creating a new project using data from SRA

If you'd like to analyze public data from NCBI SRA, we provide a feature in `sunbeam init` to use SRA as a data source instead of files already on local disk. Run `sunbeam init` with `--data_acc` instead of `--data_fp`, giving one or more SRA accession numbers.

```
sunbeam init /path/to/my_project --data_acc SRP#######
```

You can pass any number of SRA run identifiers (SRR/ERR - one sample), SRA project identifiers (SRP/ERP - one or more samples), or BioProject identifiers (PRJNA - one or more samples). For example, the below command will initialize a project for analyzing the 34 samples from `SRP159164` plus the single sample `ERR1759004`:

```
sunbeam init /path/to/my_project --data_acc SRP159164 ERR1759004
```

Sometimes, SRA projects contain both paired and unpaired reads. If this is the case, two config files and sample lists will be output–one prepended with "paired_"and one prepended with "unpaired_" (as Sunbeam runs on either paired or unpaired reads, but not both). Sunbeam uses the SRA metadata to call reads as paired- or single-end so it is only as accurate as the SRA metadata.

2.4 Configuration

Sunbeam has lots of configuration options, but most don't need individual attention. Below, each is described by section.

2.4.1 Sections

all

- `root`: The root project folder, used to resolve any relative paths in the rest of the config file.
- `output_fp`: Path to where the Sunbeam outputs will be stored.
- `samplelist_fp`: Path to a comma-separated file where each row contains a sample name and one or two paths (if single- or paired-end) to raw gzipped fastq files. This can be created for you by `sunbeam init` or `sunbeam list_samples`.
- `paired_end`: 'true' or 'false' depending on whether you are using paired- or single-end reads.
- `download_reads`: 'true' or 'false' depending on whether you are using reads from NCBI SRA.
- `version`: Automatically added for you by `sunbeam init`. Ensures compatibility with the right version of Sunbeam.

qc

- `suffix`: the name of the subfolder to hold outputs from the quality-control steps
- `threads`: the number of threads to use for rules in this section
- `seq_id_ending`: if your reads are named differently, a regular expression string defining the pattern of the suffix. For example, if your paired read ids are `@D00728:28:C9W1KANXX:0/1` and `@D00728:28:C9W1KANXX:0/2`, this entry of your config file would be: `seq_id_ending: "/[12]"`
- `java_heapsize`: the memory available to Trimmomatic
- `leading`: (trimmomatic) remove the leading bases of a read if below this quality
- `trailing`: (trimmomatic) remove the trailing bases of a read if below this quality
- `slidingwindow`: (trimmomatic) the [width, avg. quality] of the sliding window
- `minlength`: (trimmomatic) drop reads smaller than this length
- `adapter_template`: (trimmomatic) path to the Illumina paired-end adaptors (templated with `$CONDA_ENV`) (autofilled)
- `fwd_adaptors`: (cutadapt) custom forward adaptor sequences to remove using cutadapt. Replace with "" to skip.
- `rev_adaptors`: (cutadapt) custom reverse adaptor sequences to remove using cutadapt. Replace with "" to skip.
- `mask_low_complexity`: [true/false] mask low-complexity sequences with Ns
- `kz_threshold`: a value between 0 and 1 to determine the low-complexity boundary (1 is most stringent). Ignored if not masking low-complexity sequences.
- `kz_window`: window size to use (in bp) for local complexity assessment. Ignored if not masking low-complexity sequences.

- `pct_id`: (decontaminate) minimum percent identity to host genome to consider match
- `frac`: (decontaminate) minimum fraction of the read that must align to consider match
- `host_fp`: the path to the folder with host/contaminant genomes (ending in *.fasta)

classify

- `suffix`: the name of the subfolder to hold outputs from the taxonomic classification steps
- `threads`: threads to use for Kraken
- `kraken_db_fp`: path to Kraken database

assembly

- `suffix`: the name of the folder to hold outputs from the assembly steps
- `min_len`: the minimum contig length to keep
- `threads`: threads to use for the MEGAHIT assembler

annotation

- `suffix`: the name of the folder to hold contig annotation results
- `min_contig_length`: minimum length of contig to annotate (shorter contigs are skipped)
- `circular_kmin`: smallest length of kmers used to search for circularity
- `circular_kmax`: longest length of kmers used to search for circularity
- `circular_min_length`: smallest length of contig to check for circularity

blast

- `threads`: number of threads provided to all BLAST programs

blastdbs

- `root_fp`: path to a directory containing BLAST databases (if they're all in the same place)
- `nucleotide`: the section to define any nucleotide BLAST databases (see tip below for syntax)
- `protein`: the section to define any protein BLAST databases (see tip below)

Tip: The structure for this section allows you to specify arbitrary numbers of BLAST databases of either type. For example, if you had a local copy of nt and a couple of custom protein databases, your section here would look like this (assuming they're all in the same parent directory):

```
blastdbs:
  root_fp: "/local/blast_databases"
  nucleotide:
    nt: "nt/nt"
  protein:
```

(continues on next page)

(continued from previous page)

```
    vfdb: "virulence_factors/virdb"
    card: "/some/other/path/card_db/card"
```

This tells Sunbeam you have three BLAST databases, two of which live in `/local/blast_databases` and a third that lives in `/some/other/path`. It will run nucleotide blast on the nucleotide databases and BLASTX and BLASTP on the protein databases.

mapping

- `suffix`: the name of the subfolder to create for mapping output (bam files, etc)
- `genomes_fp`: path to a directory with an arbitrary number of target genomes upon which to map reads. Genomes should be in FASTA format, and Sunbeam will create the indexes if necessary.
- `threads`: number of threads to use for alignment to the target genomes
- `samtools_opts`: a string added to the `samtools view` command during mapping. This is a good place to add '-F4' to keep only mapped reads and decrease the space these files occupy.

download

- `suffix`: the name of the subfolder to create for download output (fastq.gz files)
- `threads`: number of threads to use for downloading (too many at once may make NCBI unhappy)

2.5 Building Databases

A detailed discussion on building databases for tools used by Sunbeam, while important, is beyond the scope of this document. Please see the following resources for more details:

- BLAST databases
- kraken databases
- kraken2 databases (used in Sunbeam v3.0 and higher)

2.6 Running

To run Sunbeam, make sure you've activated the sunbeam environment. Then run:

```
sunbeam run --configfile ~/path/to/config.yml
```

There are many options that you can use to determine which outputs you want. By default, if nothing is specified, this runs the entire pipeline. However, each section is broken up into subsections that can be called individually, and will only execute the steps necessary to get their outputs. These are specified after the command above and consist of the following:

- `all_qc`: basic quality control on all reads (no host read removal)
- `all_decontam`: quality control and host read removal on all samples
- `all_mapping`: align reads to target genomes

- `all_classify`: classify taxonomic provenance of all qc'd, decontaminated reads
- `all_assembly`: build contigs from all qc'd, decontaminated reads
- `all_annotate`: annotate contigs using defined BLAST databases

To use one of these options, simply run it like so:

```
sunbeam run -- --configfile ~/path/to/config.yml all_classify
```

In addition, since Sunbeam is really just a set of snakemake rules, all the (many) snakemake options apply here as well. Some useful ones are:

- `-n` performs a dry run, and will just list which rules are going to be executed without actually doing so.
- `-k` allows the workflow to continue with unrelated rules if one produces an error (useful for malformed samples, which can also be added to the `exclude` config option).
- `-p` prints the actual shell command executed for each rule, which is very helpful for debugging purposes.
- `--cores` specifies the total number of cores used by Sunbeam. For example, if you run Sunbeam with `--cores 100` and each rule/processing step uses 20 threads, it will run 5 rules at once.

2.6.1 Cluster options

Sunbeam inherits its cluster abilities from Snakemake. There's nothing special about installing Sunbeam on a cluster, but in order to distribute work to cluster nodes, you have to use the `--cluster` and `--jobs` flags. For example, if we wanted each rule to run on a 12-thread node, and a max of 100 rules executing in parallel, we would use the following command on our cluster:

```
sunbeam run -- --configfile ~/path/to/config.yml --cluster "bsub -n 12" -j 100 -w 90
```

The `-w 90` flag is provided to account for filesystem latency that often causes issues on clusters. It asks Snakemake to wait for 90 seconds before complaining that an expected output file is missing.

2.7 Outputs

This section describes all the outputs from Sunbeam. Here is an example output directory, where we had two samples (sample1 and sample2), two BLAST databases, one nucleotide ('bacteria') and one protein ('card').

```
sunbeam_output
├── annotation
│   ├── blastn
│   │   └── bacteria
│   │       └── contig
│   ├── blastp
│   │   └── card
│   │       └── prodigal
│   ├── blastx
│   │   └── card
│   │       └── prodigal
│   ├── genes
│   │   └── prodigal
│   │       └── log
│   └── summary
├── assembly
```

(continues on next page)

(continued from previous page)

```
      │   ├ contigs
      ├ classify
      │   └ kraken
      │       └ raw
      ├ mapping
      │   └ genome1
      └ qc
          ├ cleaned
          ├ decontam
          ├ log
          │   ├ decontam
          │   ├ cutadapt
          │   └ trimmomatic
          └ reports
```

In order of appearance, the folders contain the following:

2.7.1 Contig annotation

```
sunbeam_output
      ├ annotation
      │   ├ blastn
      │   │   └ bacteria
      │   │       └ contig
      │   ├ blastp
      │   │   └ card
      │   │       └ prodigal
      │   ├ blastx
      │   │   └ card
      │   │       └ prodigal
      │   ├ genes
      │   │   └ prodigal
      │   │       └ log
      │   └ summary
```

This contains the BLAST results in XML from the assembled contigs. `blastn` contains the results from directly BLASTing the contig nucleotide sequences against the nucleotide databases. `blastp` and `blastx` use genes identified by the ORF finding program Prodigal to search for hits in the protein databases.

The genes found from Prodigal are available in the `genes` folder.

Finally, the `summary` folder contains an aggregated report of the number and types of hits of each contig against the BLAST databases, as well as length and circularity.

2.7.2 Contig assembly

```
├ assembly
│   ├ contigs
```

This contains the assembled contigs for each sample under 'contigs'.

2.7.3 Taxonomic classification

```
├ classify
│   └ kraken
│       └ raw
```

This contains the taxonomic outputs from Kraken, both the raw output as well as summarized results. The primary output file is `all_samples.tsv`, which is a BIOM-style format with samples as columns and taxonomy IDs as rows, and number of reads assigned to each in each cell.

2.7.4 Alignment to genomes

```
├ mapping
│   └ genome1
```

Alignment files (in BAM format) to each target genome are contained in subfolders named for the genome, such as 'genome1'.

2.7.5 Quality control

```
└ qc
    ├ cleaned
    ├ decontam
    ├ log
    │   ├ decontam
    │   ├ cutadapt
    │   └ trimmomatic
    └ reports
```

This folder contains the trimmed, low-complexity filtered reads in `cleaned`. The `decontam` folder contains the cleaned reads that did not map to any contaminant or host genomes. In general, most downstream steps should reference the `decontam` reads.

CHAPTER 3

Sunbeam Extensions

Sunbeam extensions allow you to add new features to the pipeline. You might use an extension to carry out a new type of analysis, or to produce a report from Sunbeam output files. Extensions can be re-distributed and installed by other researchers to facilitate reproducible analysis.

A Sunbeam extension consists of a directory that contains a *rules* file. The *rules* file must have a name that ends with the file extension ".rules". The name of the extension is derived from the name of this file. It is customary for the name of the extension to start with "**sbx_**", which is shorthand for "Sunbeam extension." Technically, only the *rules* file is needed for a Sunbeam extension. In practice, almost all extensions include other files to facilitate the installation of software dependencies, to specify parameters in the configuration file, and to give instructions to users.

In Sunbeam version 3.0 and higher, extensions can be installed using the `sunbeam extend` command, followed by the GitHub URL of the extension you're installing. For example, to install an extension to run the kaiju classifier, you would run:

```
sunbeam extend https://github.com/sunbeam-labs/sbx_kaiju/
```

Sunbeam extensions are installed by placing the extension directory in the `extensions/` subdirectory of the Sunbeam software. Once the extension is in place, Sunbeam will find the *rules* file and incorporate the new functionality into the Sunbeam workflow.

3.1 Writing the rules file

The rules file contains the code for one or more rules in Snakemake format. These rules describe how to take files produced by Sunbeam and do something more with them. When Sunbeam is run, the Snakemake workflow management system determines how to put the rules together in the right order to produce the desired result.

3.1.1 Example: MetaSPAdes assembler

We'll use the sbx_metaspades_example extension to illustrate the basics. This extension runs the MetaSPAdes assembly program on each pair of host-filtered FASTQ files, producing a folder with contigs for each sample.

The rules file contains two rules: one describes how to run the MetaSPAdes program, and the other gives the full set of output folders we expect to make. Here is the rule that runs the program:

```
rule run_metaspades:
    input:
        r1 = str(QC_FP/'decontam'/'{sample}_1.fastq.gz'),
        r2 = str(QC_FP/'decontam'/'{sample}_2.fastq.gz')
    output:
        str(ASSEMBLY_FP/'metaspades'/'{sample}')
    shell:
        "metaspades.py -1 {input.r1} -2 {input.r2} -o {output}"
```

The first line indicates a new rule named `run_metaspades`. The `input` and `output` sections contain patterns for file paths that will be used by the command and produced after the command is run. The command itself is given in the `shell` section. Files from the input and output sections are indicated in curly braces inside the command. In the case of multiple inputs, you can assign them individual names, like `r1` and `r2`, then refer to these names inside the command.

In this example, the input and output filepaths are given as Python code. This is typical for rules in Sunbeam, because we are using info from the user's configuration to determine the full filepath. Let's take `str(QC_FP/'decontam'/'{sample}_1.fastq.gz')` and see how it's put together. `QC_FP` is a Python variable, which gives the absolute path to the directory containing quality control results in Sunbeam. This value is a special `Path` object in Python, which means that we can add subdirectories using the division symbol `/`. As punishment for this convenience, we have to convert this `Path` object back to an ordinary string before it can be used in the rule. The `str()` function accomplishes this. Here, `decontam` is the name of a subdirectory inside the main quality control directory. The last part, `{sample}_1.fastq.gz` looks like the name of a gzipped FASTQ file, but has something going on inside those curly braces.

The `{sample}` bit inside the input and output sections is the most important part of this rule. It's called a *wildcard*, and indicates that for any sample, we expect to see certain files before and after the command is run. If a sample is named `Abc`, Snakemake will look for the files `Abc_1.fastq.gz` and `Abc_2.fastq.gz` before running the command. If these files don't exist, that's an error. Likewise, Snakemake will check for the subdirectory `metaspades/Abc` after the command is run; if it's not there, Snakemake will stop and report the discrepancy. Wildcards allow you to write one rule that runs on all the samples.

We have a standard format for files that can be used as input to your extension. Here are all the patterns you can use:

Sequence data files	Target
Quality-controlled, non-decontaminated sequences	str(QC_FP/'cleaned'/'{sample}_{rp}.fastq.gz') str(QC_FP/'cleaned'/'{sample}_1.fastq.gz') str(QC_FP/'cleaned'/'{sample}_2.fastq.gz')
Quality-controlled, decontaminated sequences	str(QC_FP/'decontam'/'{sample}_{rp}.fastq.gz') str(QC_FP/'decontam'/'{sample}_1.fastq.gz') str(QC_FP/'decontam'/'{sample}_2.fastq.gz')
Contig sequences	str(ASSEMBLY_FP/'contigs'/'{sample}-contigs.fa')
Open reading frame nucleotide sequences	str(ANNOTATION_FP/'genes'/'prodigal'/'{sample}_genes_nucl.fa')
Open reading frame protein sequences	str(ANNOTATION_FP/'genes'/'prodigal'/'{sample}_genes_prot.fa')

Summary tables	Target
Attrition from decontamination and quality control	str(QC_FP/'reports'/'preprocess_summary.tsv')
Sequence quality scores	str(QC_FP/'reports'/'fastqc_quality.tsv')
Taxonomic assignments from Kraken	str(CLASSIFY_FP/'kraken'/'all_samples.tsv')

Very often, you will use one of these patterns directly for input to a rule. For output from a rule, you'll often use a pattern from the tables above and modify it to suit your command.

Our example extension has one more rule, representing the final set of files that should be produced after the extension is run. This rule has no output and no command; only input. After Python evaluates the code, the input to this rule is the full list of directories created by MetaSPAdes for every sample:

```
rule all_metaspades:
    input:
        expand(str(ASSEMBLY_FP/'metaspades'/'{sample}'),
               sample=Samples.keys())
```

This rule is critical for the `{sample}` pattern to work inside the Snakemake workflow management system. To determine the names of the samples, Snakemake *works backwards*, starting with the files you *would like to produce* at the end of the workflow. Snakemake does not work forward; you can't give it a list of samples or assume that it will match against input files already present. This may seem strange, but this way of working allows Snakemake to assemble a workflow containing only the steps that are needed to make a particular set of output files.

Fortunately, there is a basic pattern employed to write rules like this. Here, we take the output pattern from our other rule; this gives the pattern for the files we'd like to have at the end. Then, we use a function called `expand` to generate the full list of files. The `expand` function expects to get a list of all possible values for every wildcard in the filename. Sunbeam provides two variables for this purpose: `Samples.keys()` gives the full list of sample names, and `Pairs` gives the values used for the forward and reverse reads in the file. Here, we give `sample=Samples.keys()` as an additional argument to `expand()`, and the function produces a list of all the outputs we expect.

When the user runs the extension, they specify the rule name, `all_metaspades`. Using the full list of output directories, Snakemake figures out what sample files it needs to use, figures out what commands to run, runs the commands in parallel if possible, and lets you know if there were any problems.

3.1.2 Example: a reproducible report

As another example, we'll look at an extension that takes standard output from Sunbeam and produces a report. The extension sbx_shallowshotgun_pilot enables researchers to re-run the analysis for a small methods comaprison.

To make a report from Sunbeam output files, the extension needs only one rule.

```
rule make_shallowshotgun_report:
    input:
        kraken = str(CLASSIFY_FP/'kraken'/'all_samples.tsv'),
        preprocess = str(QC_FP/'preprocess_summary.tsv'),
        quality = str(QC_FP/'fastqc_quality.tsv'),
        sampleinfo = sunbeam_dir + '/extensions/sbx_shallowshotgun_pilot/data/
↪sampleinfo.tsv'
    output:
        str(Cfg['all']['output_fp']/'reports/ShallowShotgun_Pilot_Report.html')
    script:
        'shallowshotgun_pilot_report.Rmd'
```

Here, the output is a single file path, and the path does not contain any wildcards like `{sample}`. Therefore, Snakemake can work backwards from the output file and figure out everything it needs; we can use this rule as our final target when running Sunbeam.

The basic structure of the rule and most of the inputs should be familiar from the previous example. One of the inputs, `sampleinfo`, does not come from Sunbeam, but is distributed with the extension. We know the filepath inside the extension is `data/sampleinfo.tsv`, but we need to specify the entire path for Snakemake to find the file. To do this, we use the variable `sunbeam_dir`, which points to the Sunbeam installation directory. The extension must be located inside the `extensions/` subdirectory to run. From here, we know how to get to our file. Because the value of `sunbeam_dir` is an ordinary string, we use the `+` symbol to add on the `extensions/` subdirectory, the directory name for the extension, and the path to the file inside the extension directory. This example shows how to refer to files inside the Sunbeam installation directory.

In the output section, we need to specify a file path for the final report. Here, we use the configuration parameter `Cfg['all']['output_fp']` to get the base directory for output from Sunbeam. The value of this configuration parameter is a `Path` object, so we use the `/` symbol to add the rest of the filepath, and surround the whole thing with the `str()` function. Just as a note, Snakemake will create the `reports/` subdirectory if needed, so you don't have to worry about directories being present ahead of time to accommodate your output files.

At the bottom of the rule, we write `script` instead of `shell`, because we'd like Snakemake to run a script instead of a shell command. Here, we give the name of a script in R Markdown format. The file path of the script is given *relative to the rules file*, which is a little bit different from all the other file paths in the rules file, but convenient.

Inside the script, we need to access the input files given in the rule. Here is the part of the script that accesses the input file paths and saves them as ordinary variables in R:

```
sample_fp <- file.path(snakemake@input[["sampleinfo"]])
preprocess_fp <- file.path(snakemake@input[["preprocess"]])
quality_fp <- file.path(snakemake@input[["quality"]])
kraken_fp <- file.path(snakemake@input[["kraken"]])
```

The R Markdown tutorial and book are the best sources of information on the report format, whereas the R for data science book provides a good introduction to the R programming languageas you might use it in the report.

3.1.3 Variables provided by Sunbeam

Here is a table of all the Python variables provided by Sunbeam for use in your extensions:

Variable name	Type	Description
`QC_FP`	Path	Output directory for quality control files.
`ASSEMBLY_FP`	Path	Output directory for assembly files.
`ANNOTATION_FP`	Path	Output directory for gene annotation files.
`CLASSIFY_FP`	Path	Output directory for taxonomic classification files.
`Samples`	Dictionary	Key is the sample name, value is a dictionary with keys "1" and "2", values are the the gzipped FASTQ files at the start of the workflow. For unpaired reads the value for "2" is the empty string.
`Pairs`	List	For paired reads, ["1", "2"]. For unpaired reads, ["1"].
`Cfg`	Dictionary	Parameters found in the configuration file. For any parameter ending in "_fp", the value is converted to a Path object. The most commonly used parameter is `Cfg['all']['output_dir']`, which gives the base output directory.
`sunbeam_dir`	String	File path where Sunbeam is installed.

3.1.4 Further reading

We're only scratching the surface of what you can do with rules in Snakemake. The official Snakemake documentation gives excellent instructions with more examples.

3.2 Software dependencies

If your extension requires additional software to be installed, you can provide the names of Conda packages inside a file named `requirements.txt`. This file contains the package names, one per line. To install Conda packages in this file, users of your extension will run the `conda install` command with this file as an additonal argument:

```
conda install --file requirements.txt
```

3.3 Configuration

Your extension can include its own section in the configuration file. To take advantage of this, you would write an example configuration file named `config.yml`. This file should contain only one additional configuration section, specifying parameters for your extension. For example, the sbx_coassembly extension includes two parameters: the number of threads to use, and the path to a file with groups of samples to co-assemble.:

```
sbx_coassembly:
  threads: 4
    group_file: ''
```

As of version 3.0, config options from extensions are automatically included in config files made using `sunbeam init` and `sunbeam config update`. This functionality depends on the extension's configuration file being named `config.yml`.

In version <3.0, users can copy this example section to the end of their configuration file, using `cat`:

```
cat config.yml >> /path/to/user/sunbeam_config.yml
```

In your *rules* file, you can access parameters in the configuration like this: `Cfg['sbx_coassembly']['group_file']`.

3.4 The README file

We recommend that you include a README file in your extension. The contents of the file should be in Markdown format, and the file should be named `README.md`. Here's what you should cover in the README file:

1. A short summary of what your extension does
2. Any relevant citations
3. Instructions to install
4. Instructions to configure
5. Instructions to run

A good example to follow is the sbx_coassembly extension.

3.5 Publishing at sunbeam-labs.org

You are welcome to add your Sunbeam extensions to the directory at sunbeam-labs.org. To submit your extension to the directory, please go to the development page for sunbeam-labs.org and open an issue with the GitHub URL of your extension. If you know Javascript, you can edit the list at the top of the file `main.js` and send us a pull request.

CHAPTER 4

Citing Sunbeam

If you use Sunbeam in your research, please cite:

EL Clarke, LJ Taylor, C Zhao *et al.* Sunbeam: an extensible pipeline for analyzing metagenomic sequencing experiments. *Microbiome* 7:46 (2019)